Jolly Pochie
Technical Report RoboCup 2005

Hayato Kobayashi [1] Tsugutoyo Osaki [2] Akira Ishino [3]
Jun Inoue [1] Narumichi Sakai [1] Satoshi Abe [2]
Shuhei Yanagimachi [2] Tetsuro Okuyama [2] Akihiro Kamiya [2]
Kazuyuki Narisawa [1] Ayumi Shinohara [2]

[1] Department of Informatics, Kyushu University
[2] Department of System Information Sciences, GSIS, Tohoku University
[3] Office for Information of University Evaluation, Kyushu University

November 10, 2005

Kyushu University and Tohoku University

Contents

Chapter 1

Introduction

The team "Jolly Pochie [dʒóli·pótʃi:]" has participated in RoboCup Four-Legged League since 2003. In the last two years, the team consisted of the faculty staff and graduate/undergraduate students of department of informatics, Kyushu University [1]. This year it becomes a united team with Tohoku University.

Faculty members
Ayumi Shinohara and Akira Ishino

Graduate Students
Jun Inoue, Hayato Kobayashi, Narumichi Sakai, Kazuyuki Narisawa, Satoshi Abe, and Akihiro Kamiya

Undergraduate Students
Tsugutoyo Osaki, Tetsuro Okuyama, Shuhei Yanagimachi, and Yuki Matsumoto

Our research interests mainly include machine learning, machine discovery, data mining, image processing, string processing, software architecture, visualization, and so on. RoboCup is a suitable benchmark problem for these domains. For this year, we utilized an embedded scripting language in order to accelerate the development process, and established a simulator that can execute robot scripts without modifying the script. In addition, we are developing a new localization technique for the ball location.

The rest of this report is organized as follows. Chapter 2 introduces our original framework, into which we embedded scripting language Lua. Chapter 3, 4, and 5 describe motion modules, vision modules, and other modules, respectively, that we developed in this year. Chapter 6 describes our strategy system, Chapter 7 illustrates soccer strategies used in the system, and Chapter 8 shows shoot and pass motions used in the strategies. Chapter 9 shows the bots used for playing soccer, capturing images, and so on. Chapter 10 illustrates our tools that have been developed so far. Chapter 11

describes the results of the technical challenges in RoboCup 2005. Finally, Chapter 12 presents the conclusion of this report.

Chapter 2

Framework

This chapter describes the architecture of Jolly Pochie. We present the original framework for AIBO programming. It consists of a base system and many modules which are easily exchangeable with other modules.

2.1 JPObject

In the operating system Aperios, which was developed by SONY and runs on AIBO, multiple objects are processed concurrently, communicating with each other. The software development kit OPEN-R for Aperios defines a special object OVirtualRobot. OVirtualRobot is a kind of proxy object for AIBO. By communicating with OVirtualRobot, we are able to move joints, to capture images from the camera, to send and receive data with other AIBO's from wireless LAN, and so on.

We use several Aperios objects, which are sub-classes of OObject, in our soccer robot. PowerMonitor and TinyFTPD are come from the OPEN-R distribution, GameController is from RoboCup community, and TCPServer, UDPServer, and JPObject are implemented by us.

JPObject plays a main role in Jolly Pochie, which has no function in itself, but manages many modules and communicates with other objects (Fig. 2.1). Most of events on a robot, *e.g.* action events, camera events, sensor events, network events, and so on, gather to JPObject. Then JPObject assigns these events to appropriate modules. On the other hand, actions which occurred on modules are translated by JPObject and communicate to OVirtualRobot.

2.2 JPModule

Modules are managed by JPObject and provide various functions for moving joints, capturing camera images, detecting objects, localizing self-position, and so on. Each module is designed for an atomic function and is developed

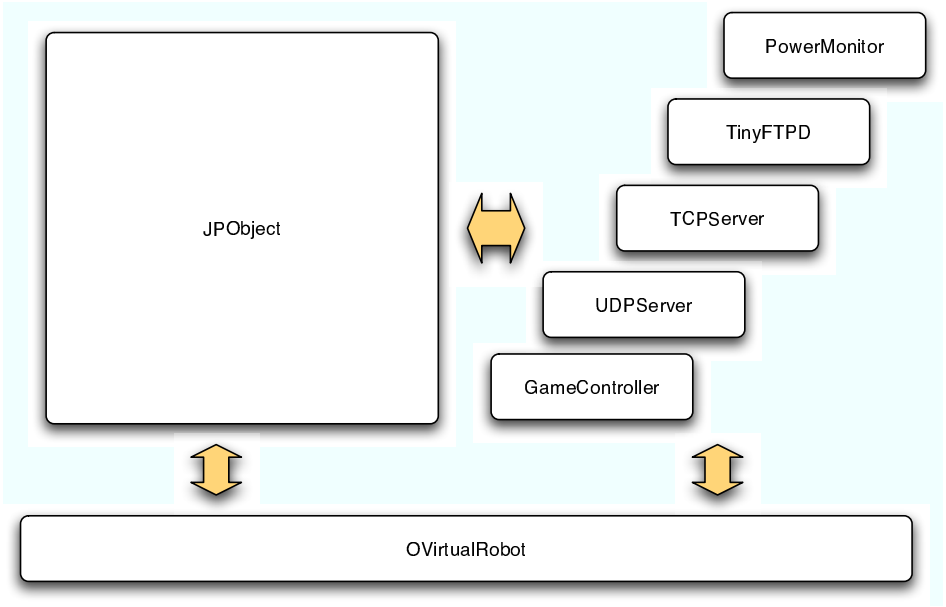


Figure 2.1: JPObject and other programs.

independently. Combining suitable modules, we can get robots for various purpose easily. Someone improves a certain module, others receive the benefits automatically. When an experimental module is developed by modifying from another module, the original module is left as it was, and any other developer need not to concern about the modifications. The module system of Jolly Pochie makes the development of robot programming easy. In this year 2005, more than 200 modules were created. (Unfortunately, about half of them were failed ones.)

The base class of modules is JPModule. JPModule registers itself in JPObject by the constructor. The only thing which a module has to do is to define a default constructor in which the constructor of JPModule is called with the module name. Once the modules are registered, JPObject call these modules when an event has occurred. For such a purpose, JPModule have four methods: init(), start(), stop(), and destroy() which correspond to JPObject::DoInit(), JPObject::DoStart(), JPObject::DoStop() and JPObject::DoDestroy(), respectively.

The following program is a simple example of the module. The HelloWorld module says “Hello world!” in the monitor console at start-up, and says “bye” at end.

```
---HelloWorldJPM.h
```

```
#include "JPModule.h"

class HelloWorldJPM : public JPModule {
public:
  HelloWorldJPM();
  virtual ~HelloWorldJPM() {}
  virtual void start();
  virtual void stop();

};

---HelloWorldJPM.cc
#include "HelloWorldJPM.h"
#include "JPSyslog.h"

HelloWorldJPM::HelloWorldJPM() : JPModule("HelloWorldJPM") {}

void
HelloWorldJPM::start() {
  JPSysDebug(("Hello world!\n"));
}

void
HelloWorldJPM::stop() {
  JPSysDebug(("bye"));
}
```

The above example also shows that no OPEN-R functions and data types are used in the module directly. We hides the functions and types in OPEN-R. Instead of these, we provide many useful functions which use standard C++ types, *e.g.* string, vector, and so on. Excluding OPEN-R functions and type from module programs makes it possible to debug and test in the ordinal environment, with unit test libraries, and without AIBO's.

There exist some specific base modules for events of OVirtualRobot (Fig. 2.2). JPActionModule is the base module for action and has a method actionReady called every 80ms for transferring joint angles to OVirtualRobot. JPSpeakerModule is for speaker and has a method speakerReady. JPSensorModule is for sensor and has a method sensorNotify. JPCameraModule is for camera and has a method cameraNotify called every 40ms for transferring data of CCD-camera from OVirtualRobot. JPTCPModule is for tcp communication and has methods tcpNotify and tcpReady. JPUDPModule is for udp communications and has methods udpNotify and udpReady. JPMindModule is a slightly special module, which does not cor-

respond with any events of OVirtualRobot. JPMindModule has methods mindNotify which is called after cameraNotify and gameNotify for events of GameController.

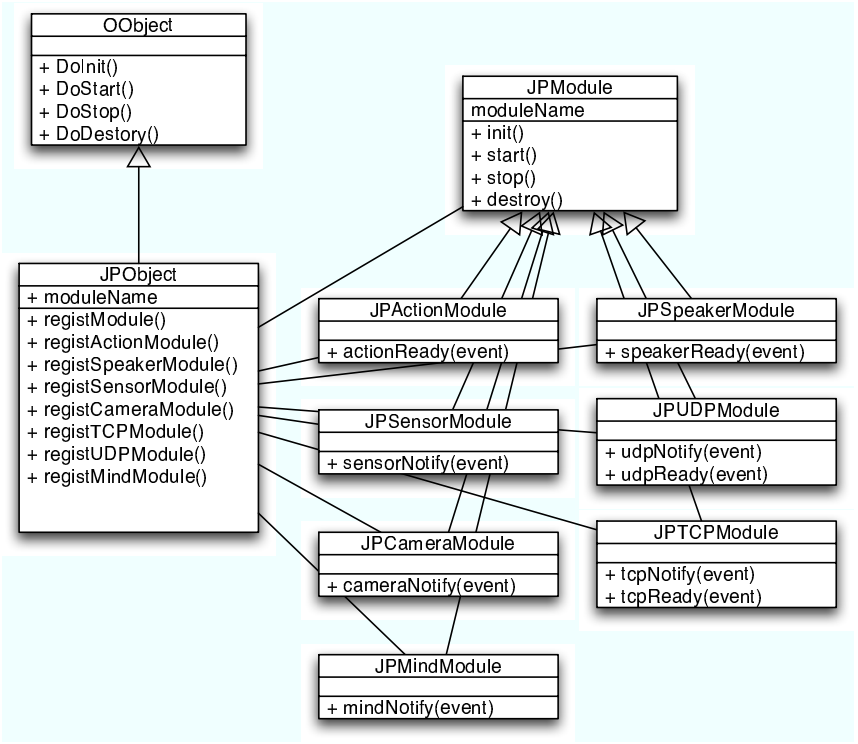


Figure 2.2: JPObject and JPModules.

By inheriting one of the base module class and implementing the specific methods, we can create modules which use various functions of AIBO (Fig. 2.3). In other words, JPObject does nothing but manages modules, and everything is processed by the modules.

2.3 Embedding Lua

Using compiled languages such as C++, it takes an awfully long time to adjust the parameters of complex programs designed for robots to play soccer. The reason is that both sensor input and actuator output can hardly be precisely predicted, since our robots act in the real world. It is very difficult to make our robots behave as we expect without a lot of trial and error. Thus, many advanced teams have adopted scripting languages into their systems. Embedding scripting language dramatically improves the efficiency of development, because we do not have to recompile the source code, and because we can update the scripts at runtime without rebooting the robot.

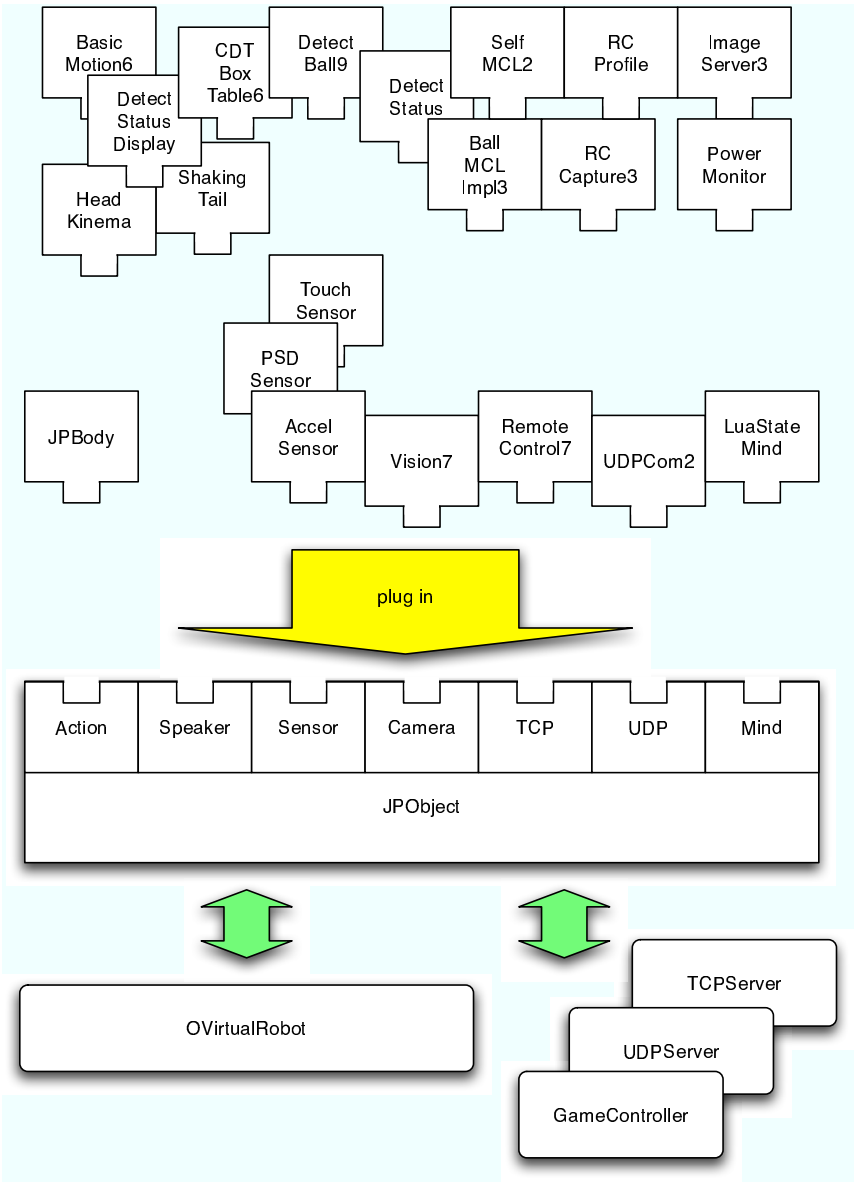


Figure 2.3: JPModules are plugged in the JPObject.

When we try to embed a scripting language into our framework, a problem often arises to bind C++ modules (*i.e.* to register C++ classes in order to use member functions of it in Lua scripts). If we bind in the common base program of our framework, we need likely to rewrite the code of binding whenever we replace the modules. In addition, if we bind in each module, we might have to rewrite the scripts, or it might become difficult to create a new module, whenever we replace the modules.

In this section, we show how to embed scripting language Lua [7] in our framework. Thanks to the excellent mechanisms supplied by Luabind [5], it becomes quite easy to bind C++ modules in this system. Thus, our framework enables us to create many modules easily and quickly without the difficulty of the bindings. As a matter of course, the scripts can always access to the methods of the modules. Therefore, we can develop each low-level module separately and independently, while some other teammates are writing and adjusting high-level strategetic scripts. This is quite advantageous to the development for RoboCup competition, because even a new member of the team, who is not familiar with practical C++ programming nor understand the whole complicated system developed so far, can contribute to adjust the parameters, and to examine another strategy, and so on. Through these experiences, she/he will understand the whole system gradually, and be prepared to join the development of the core parts of it.

2.3.1 Lua

Lua [7] is a scripting language designed to embed into C/C++. It has a nice, simple, powerful syntax, and a small scripting engine (parser, compiler, and interpreter), as well as being easily embedded. Therefore, we could easily embed Lua in the Jolly Pochie Framework. we could get used to write Lua scripts a short time later.

The syntax of Lua is similar to that of Pascal. The following is an example of the function `sum()` that takes a variable number of integers as arguments, and returns a summation of the arguments. We had better execute local declaration for the variable `s`, because variables are normally regarded as global variables.

```
function sum(...)
   local s = 0
   for i=1, arg.n do
      sum = s + arg[i]
   end
   return s
end
```

Lua has a flexible table structure that can represent ordinary arrays, symbol tables, sets, records, graphs, trees, and so on. Since functions are

first class values in Lua, we can represent the structure that is like a class in C++. In addition, we can make a subclass that derives the class by using a metatable. The metatable is an ordinary table that defines the behavior of a table.

The following is an example of expression of classes, and inheritance of the classes using the metatable. In the example, we create the class `AIBO`, and the subclass `ERS7` of the class `AIBO`. The function `setmetatable(tbl, mtbl)` sets exploring not only the field of the table `ERS7`, but also the field of the table `AIBO`, when accessing the index of the table `ERS7`. Thus, the member function `ERS7:getColor()` returns the value `"white"`. `ERS7:getColor()` is syntax sugar for `ERS7.getColor(ERS7)`, and `ERS7.getColor` is also syntax sugar for `ERS7["getColor"]`.

```
AIBO = {
   color = "",
   getColor = function(self)
                 return self.color
              end
}
ERS7 = {
   color = "white"
}
setmetatable(ERS7, {__index = AIBO})
```

2.3.2 Luabind

We use Luabind [5] to embed Lua in our framework, although Lua can be embedded using only Lua API. The reason is that it is difficult to bind C++ classes using only Lua API. We will have to define global wrapper functions for the member functions of the classes, and allow instances of the classes to access the member functions by using metatables.

Luabind is a library that helps you create bindings between C++ and Lua. We do not have to write the annoying processes to bind Lua, because It is implemented utilizing template metaprogramming to automate the processes in compiling time. Luabind supports class inheritance, overloaded functions, overridden virtual functions, and so on.

Now, we show an example of binding a class in C++ side, and using the class in Lua side. For example, we create the sample class `AIBO` in C++ as follows.

```
class AIBO {
  int speed;
public:
  AIBO(int s) { speed = s; }
```

```
  void walk() { cout << "walk " << speed << endl; }
};
```

We can easily bind the class `AIBO` defined above.

```
module(L) [
  class_<AIBO>("AIBO")
  .def(constructor<int>())
  .def("walk", &AIBO::walk)
]
```

After binding, we can instantiate the class `AIBO` in Lua side as follows. We can treat Lua classes in a similar way to C++ classes, but we must use the operator colon (`:`) instead of the operator dot (`.`) or arrow (`->`) to access member functions and variables of the classes. Since our framework needs to instantiate modules (classes) in C++ side, we must register the instances for Lua to operate the modules. This method will be taken up at length in the following section.

```
aibo = AIBO(100)
aibo:walk()
doping_aibo = AIBO(200)
doping_aibo:walk()
```

2.3.3 Calling Lua Functions

Lua scripts should be called in mind modules, because mind modules create strategies to play soccer in the RoboCup competition. We embed Lua in our framework so that a mind module can load a Lua script, and call Lua functions in the script. If we specify the script name as *Foo* in a configuration file, the mind module loads Lua script file *start.lua* in directory *Foo* when the mind module is instantiate. There are two special functions `init()` and `mindNotify()` in our robot scripts. The Lua function `init()` is called only once after the mind module are instantiated. We can initialize a strategy in the function `init()`. The Lua function `mindNotify()` is called in the C++ member function `mindNotify()` in the mind module. We can write a strategy in the function `mindNotify()`, using C++ functions in other modules.

We can quite easily call a Lua function using Luabind. In order to call the Lua function `mindNotify()` that has no argument, and returns nothing, we have only to describe as below. The variable `JPLua::L` is a pointer of the structure that register Lua status.

```
luabind::call_function<void>(JPLua::L, "mindNotify");
```

2.3.4 Binding C++ Modules

In order for Lua scripts to call some methods in other C++ modules, it is necessary to bind the modules on the C++ side. It means to register an instance of the modules, as well as information of the classes and member functions. For instance, in order to register a class `ExampleModule` with public member functions `method1(const char* str)` and `method2(int x, int y)`, we have only to add the description as follows, in the function `init()` in C++ modules.

```
module(JPLua::L) [
  class_<ExampleModule>("ExampleModule")
  .def("method1", &ExampleModule::method1)
  .def("method2", &ExampleModule::method2)
];
get_globals(JPLua::L)["exampleModule"] = this;
```

The bottom line is to register the instance, assigning the `this` pointer of the class `ExampleModule` to the variable `"exampleModule"` on the Lua side. After the registration, we can call the function in Lua scripts as follows.

```
exampleModule:method1("hello")
exampleModule:method2(10, 20)
```

Luabind can register information of class inheritance. We need not bind in the class `AdvancedExampleModule` inheriting the class `ExampleModule` as follows. Therefore, we need not rewrite the script even if we change those modules. The same is true for the compatible modules in terms of binding (*e.g.* `ExampleModule2` having the same methods that `ExampleModule` has).

```
module(JPLua::L) [
  class_<AdvancedExampleModule,
        ExampleModule>("AdvancedExampleModule")
];
get_globals(JPLua::L)["exampleModule"] = this;
```

2.3.5 Specification of Our Robot Scripts

After embedding Lua, high-level process can be described in Lua scripts. When creating scripts, we have to understand the following two functions `init()` and `mindNotify()` on the Lua side. The function `init()`, where we can initialize variables, is called only once at the beginning. The function `mindNotify()` is called by the member function `mindNotify()` in mind modules, that is to say, called every 40 ms. Therefore, a strategy can be changed depending on input from CCD-camera.

The following example is a simple script that enumerates nonnegative integers, starting with 0 and incrementing by 1.

```
function init()
  i = 0
end

function mindNotify()
  print(i)
  i = i + 1
end
```

Of course, we can use C++ functions and variables that were binded by the method of the previous section. Table 2.1 shows most of the available C++ functions and variables in Lua scripts, which we have been developed so far. There are also convenient libraries using these C++ functions and variables in Lua side. We can create robot scripts by good use of the Lua library and the C++ functions and variables. The following is another example of a robot script that make our robot move toward a ball, slow down if it is near to 500 mm, and come to a stop if it is near to 50 mm.

```
require "Vision.lua"
require "CMotion.lua"

function init()
end

function mindNotify()
  visionLib:detectBall()
  cmotion:watchBall()

  local dist = visionLib:getBallDistance()
  if dist > 500 then
    cmotion:walk(1.0, 0, 0)
  elseif dist > 50 then
    cmotion:walk(dist/100, 0, 0)
  else
  end
end
```

The function `visionLib:detectBall()` performs the process that recognizes the ball, and the function `cmotion:watchBall()` responds to swing the head to keep watching the ball. The function `cmotion:walk(`f`,`l`,`r`)` performs the parametric movement of our robot specified by three parameters ranging between -1 and 1 : forward direction f, leftward direction l, and clock-wise rotation r.

Table 2.1: Most of the available C++ functions and variables in Lua scripts.

instance	available functions and variables
faceLED	setState(bitvector)
headLED	setState(bitVector)
tailLED	setState(bitVector)
touchSensor	clickedBackFront(), clickedBackMiddle(), clickedBackRear(), clickedHead(), clickedBody(), clickedBasic(t), pressedBackFront(t), pressedBackMiddle(t), pressedBackRear(t), pressedHead(t), pressedBack(t), pressedChin()
soundPlayer	registWavFile(soundname, filename), playSoundOnce(soundname), changeVolume(volume), playSoundRepeat(soundname), playSoundStop()
visionBase	landmarksDetect(), ballDetect(), getLeftLineSlant(), getRightLineSlant(), getLeftLineIntercept(), getRightLineIntercept()
basicMotion	loadMotion(motionname, filename), playMotion(motionname, state), playMotionLoop(motionname, state), stopAction(), cancelAction(), swingHead(tilt1, pan, tilt2, nextState), setHeadDeltaTilt1(deltaTilt1), setHeadDeltaPan(deltaPan), setHeadDeltaTilt2(deltaTilt2), setHeadDelta(deltaTilt1, deltaPan, deltaTilt2), stopSwingHead(), cancelSwingHead(), loadGait(gaitname, gaitfile, odparamfile), playGait(), playGaitLoop(), clearGaitsource(gaitname, rate), setGaitSource(), setGaitFirstFlag(flag)

instance	available functions and variables
detectBall	getL(), getPan(), getTilt(), getAdvisablePan(), getAdvisableTilt(), getInSightTilt1(), getInSightPan(), getInSightTilt2()
ballMCL	getX(), getY(), getPan(), getDistance(), isValid(), addXYV(x, y, dx, dy), shiftXYT(x, y, theta), update(), nupdate(n)
mcLocalization	getX(), getY(), getTheta(), update(), nupdate(n)
udpCom	udpSay(words)
psdSensor	getHeadPsdValue(), getHeadNearPsdValue(), getHeadFarPsdValue(), getBodyPsdValue()
detectStatusDisplay	update()
gameControlData	firstHalf, kickOffTeam, secsRemaining, dropInTeam, dropInTime, myTeam.teamColour, myTeam.score, myTeam.player1.penalty, myTeam.player1.secsTillUnpenalised
detectStatus	ballDistance, ballPan, ballTilt
accelSensor	getAccelX(), getAccelY(), getAccelZ(), getValueX(i), getValueY(i), getValueZ(i), getPosture()
headKinema	setHeight(frontHeight, rearHight), getHeight(), setHeadAnglesToWatchThePosition(X, Y, Z, tilt1required)

Chapter 3

Motion

3.1 Action

In OPEN-R SDK, we can move joints by sending data, for example, 0, 10, 20, 15, 5, -5, ..., to OVirtualRobot. The number of the joint data is not fixed but we decided that the number is 10 for smooth moving and double buffering is also used. One frame of the data corresponds with 8ms. Accordingly, the time cycle of actionReady is 80ms, which is twice of the one of cameraNotify and mindNotify.

LEDs are also treated as joints in OPEN-R SDK. There are three kinds of LEDs in AIBO: face LEDs, back LEDs, and ear LEDs. We decided the number of frames for face and back LEDs is the same as joints, 10, but the number of frames for ear LEDs is 1.

However, strategic routines should not care about these low-level mechanisms. The strategic routines do not indicate the next angles of joints but decide a next action such as walk, turn, and shoot. We therefore define an action is the sequence of joint data, which are given preliminarily or calculated as needed.

3.1.1 JPAction

The base class of actions is JPAction. There exist seven subclasses of JPAction for each part of the AIBO's body. JPLegsAction is for legs. JPHeadAction is for a head. JPEarAction is for ears. JPTailAction is for a tail. JPFaceLEDAction is for face LEDs of ERS-7. JPHeadLEDAction is for ear LEDs of ERS-7 and face LEDs of ERS-210/220. JPTailLEDAction is for back LEDs of ERS-7 and tail LEDs of ERS-210/220. Finally, JPHeadLegsAction is a synchronous action for a head and legs such as shoot. These classes are abstract and, of course, do nothing. Concrete actions, walking, shooting, and so on, are described later.

Actions are processed by JPBody, which is the subclass of JPActionModule (Fig. 3.1). JPBody treats the above actions without JPHeadLegsActions

independently. Basically, once an action starts to be processed, another actions for the same type are kept waiting until the current action has finished. JPBody keep only one action for each type of actions. When an action A is waiting, another action B for the same type of action is coming, JPBody discards the action A and now keeps the action B waiting. On the other hand, let C be an action of different type from the action A. In this case, the action A is not discarded and the action C is processed at once or also is kept waiting.

JPHeadLegsAction is processed in slightly different way. JPHeadLegsAction, of course, waits for finishing the previous JPHeadLegsAction. Moreover, JPHeadLegsAction uses a head and legs and consequently waits for finishing JPHeadAction and JPLegsAction. We give a priority for JPHeadLegsAction over JPHeadAction and JPLegsAction. If JPHeadLegsAction, JPHeadAction, and JPLegsAction is waiting at the same time, then the previous JPHeadAction has finished but the next JPHeadAction does not start because JPHeadLegsAction has a priority.

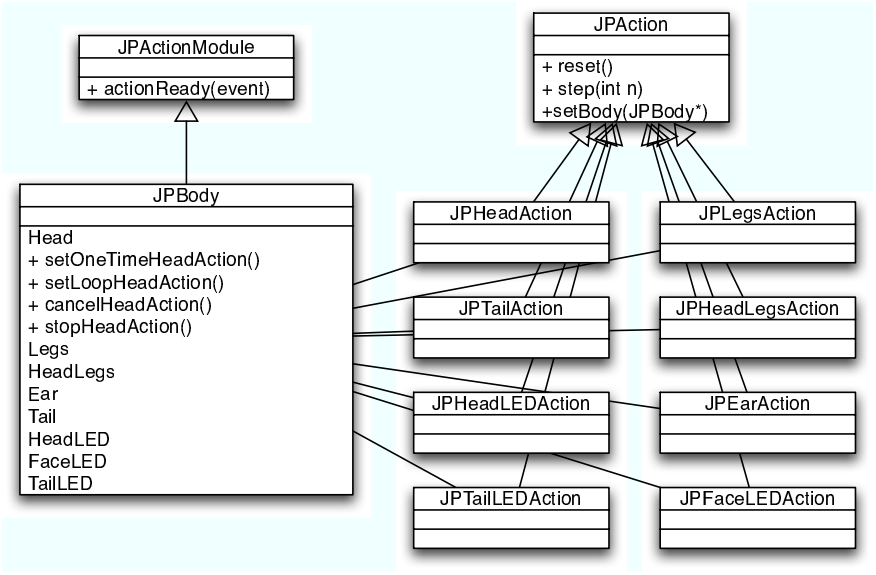


Figure 3.1: JPBody and JPAction's.

3.1.2 One Time and Loop Process

There are two ways to process actions by JPBody. One is that an action is processed once and is discarded when it has finished. The other is that an action is processed more than once until another action comes. We call the former *one time process* and the latter *loop process*. We define that one time process has a priority over loop process. In addition, when the one

time process finished, the previous loop process is processed again. In this way, we keep AIBO walking, sometimes turning, and shooting (Fig. 3.2).

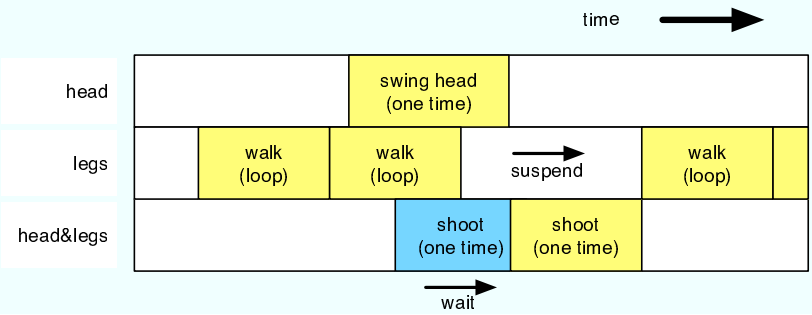


Figure 3.2: An example of timetable for processing actions.

3.1.3 Basic Actions

This section introduces and describes several basic actions. There exist actions for all parts of AIBO. The actions are classified into three categories: (static) changing angles or states into the specified one and keeping it, (sequential) changing angles or states continuously on a scenario, and (dynamic) changing angles or states whose values are calculated dynamically. Table. 3.1 shows the categories of actions.

HeadJointAngleAction

HeadJointAngleAction moves the head to the specified angles in the specified frames. If the current angle of the head is (0, 0, 60) that is the triple of tilt1, tilt2, and pan, and HeadJointAngleAction(0, 0, -60, 100) starts to be processed. The head will move from left to right at a rate of 1.2 ° a frame.

```
class HeadJointAngleAction : public JPHeadAction {
public:
  HeadJointAngleAction(JPHeadJointAngle& angle, int frame);
  void setAngle(JPHeadJointAngle& angle);
};
```

HeadJointAngleAction2

HeadJointAngleAction2 also moves the head to the specified angles but the frame is not given and calculated from dividing the difference of angles by the constant δ.

```
class HeadJointAngleAction2 : public JPHeadAction {
```

Table 3.1: The categories of actions.

	static	**sequential**	**dynamic**
Head	HeadJointAngleAction HeadJointAngleAction2	HeadMotionAction	
Legs	LegsJointAngleAction	LegsMotionAction LegsSequentialAction	CompositeWalkOdAction PositioningWalkAction
HeadLegs **HeadLegs**		HeadLegsMotionAction HeadLegsSequentialAction	
Ear	EarStateAction		
Tail			ShakingTailAction
HeadLED	HeadLEDStateAction	HeadLEDMotionAction	
TailLED	TailLEDStateAction		
FaceLED **FaceLED**	FaceLEDStateAction FaceLEDFadeStateAction	FaceLEDMotionAction	

```
public:
  HeadJointAngleAction2(JPHeadJointAngle& angle);
  void setAngle(JPHeadJointAngle& angle);
  void setTiltDelta(uradian delta);
  void setPanDelta(uradian delta);
  void setRollDelta(uradian delta);
};
```

HeadMotionAction

HeadMotionAction processes HeadMotion which is the sequence of angles of the head and frame. However, this action is merely used. The head is used to search and track the ball, or to pass and shoot the ball. In the first case, HeadJointAngleAction2 is used. In the latter case, HeadLegsMotionAction is more useful because the action can move the head and legs synchronously.

```
class HeadMotionAction : public JPHeadAction {
public:
  HeadMotionAction(HeadMotion& motion);
};
```

LegsJointAngleAction

LegsJointAngleAction moves the legs to the specified angle in the specified frames.

```
class LegsJointAngleAction : public JPLegsAction {
```

```
public:
  LegsJointAngleAction(JPLegsJointAngle& angle, int frame);
};
```

LegsMotionAction

LegsMotionAction processes LegsMotion which is the sequence of twelve angles of the legs and frame. However, instead of this action, HeadLegsMotionAction is used.

```
class LegsMotionAction : public JPLegsAction {
public:
  LegsMotionAction(LegsMotion& motion);
  void setMotion(LegsMotion& motion);
};
```

LegsSequentialAction

LegsSequentialAction processes the sequence of JPLegsActions from head to tail. This action is useful for packing some actions.

```
class LegsSequentialAction : public JPLegsAction {
public:
  LegsSequentialAction();
  void addAction(JPLegsAction* action);
};
```

CompositeWalkOdAction

CompositeWalkOdAction uses composite walk module, which generates walking motions, and odometer module. Three parameters for walking forward, sideways, and rotating are specified for the action. For example, the parameters (1.0, 0, 0) means running at full speed, (0.5, 0.5, 0) means walking diagonally, (0, -0.5, 0.5) means turning around, and so on.

```
class CompositeWalkOdAction : public JPLegsAction {
public:
  CompositeWalkOdAction(OdometerJPM* odometer);
  void setParameter(double f, double s, double r);
};
```

PositioningWalkAction

By using PositioningWalkAction, we can easily make AIBO walk to the specified position. This action uses CompositeWalkOdAction.

```
class PositioningWalkAction : public JPLegsAction {
public:
  PositioningWalkAction(OdometerJPM* odometer);
  void setTarget(int x, int y, int theta);
  void setTarget(int x, int y, int theta, int lookX, int lookY);
};
```

HeadLegsMotionAction

HeadLegsMotionAction processes HeadLegsMotion which is the sequence of twelve angles of the legs, three angles of the head and frame. This action is used for shooting, getting up, and several emotional motions.

```
class HeadLegsMotionAction : public JPLegsAction {
public:
  HeadLegsMotionAction(HeadLegsMotion& motion);
  void setMotion(HeadLegsMotion& motion);
};
```

HeadLegsSequentialAction

HeadLegsSequentialAction processes the sequence of JPHeadLegsActions. This action is useful for packing some actions.

```
class HeadLegsSequentialAction : public JPLegsAction {
public:
  HeadLegsSequentialAction();
  void addAction(JPHeadLegsAction* action);
};
```

EarStateAction

EarStateAction flicks ears with the specified states. However, we skipped processing ear actions due to the limit of CPU performance, in the games.

```
class EarStateAction : public JPEarAction {
public:
  EarStateAction();
  void setState(JPEarState& state);
};
```

ShakingTailAction

ShakingTailAction shakes the tail. This action was used for SLAM Challenge.

```
class ShakingTailAction : public JPTailAction {
public:
  ShakingTailAction();
  void startShaking();
  void stopShaking();
  void setPanDelta(uradian delta);
  void setTiltDelta(uradian delta);
};
```

HeadLEDStateAction

HeadLEDStateAction lights up head LEDs with a specified pattern.

```
class HeadLEDStateAction : public JPHeadLEDAction {
public:
  HeadLEDStateAction();
  void setState(JPHeadLEDState& state);
};
```

HeadLEDMotionAction

HeadLEDMotionAction processes HeadLEDMotion which is a sequence of illumination patterns of Head LEDs.

```
class HeadLEDMotionAction : public JPHeadLEDAction {
public:
  HeadLEDMotionAction(HeadLEDMotion& motion);
};
```

TailLEDStateAction

TailLEDStateAction lights up tail LEDs with a specified pattern.

```
class TailLEDStateAction : public JPTailLEDAction {
public:
  TailLEDStateAction();
  void setState(JPTailLEDState& state);
};
```

FaceLEDStateAction

FaceLEDStateAction changes the states of LEDs with a specified patterns.

```
class FaceLEDStateAction : public JPFaceLEDAction {
public:
  FaceLEDStateAction();
```

```
  void setState(JPFaceLEDState& state);
};
```

FaceLEDFadeStateAction

FaceLEDFadeStateAction gradually changes the states of LEDs with a specified patterns. A parameter delta specifies the rate of changing states.

```
class FaceLEDFadeStateAction : public JPFaceLEDAction {
public:
  FaceLEDFadeStateAction();
  void setState(JPFaceLEDState& state);
  void setDelta(int delta);
};
```

FaceLEDMotionAction

FaceLEDMotionAction processes FaceLEDMotion which is a sequence of illumination patterns of Face LEDs. Unfortunately, FaceLEDMotionAction is not used yet due to the lack of the face LED motion editor.

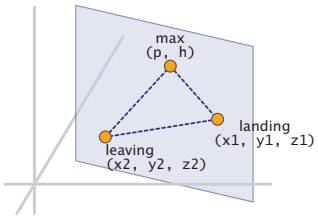
```
class FaceLEDMotionAction : public JPFaceLEDAction {
public:
  FaceLEDMotionAction(FaceLEDMotion& motion);
};
```

3.2 Gait

Here, we will introduce two processes, which are necessary to create smooth and fast walking. One is a manual tuning and the other is an automatic optimization. We first explain the locus of gait and its creation process with tools, and then we show the optimization process for better walking.

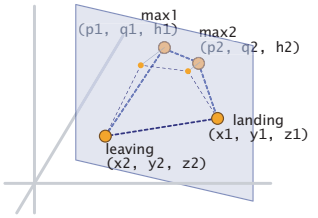
3.2.1 Locus of Gaits

Aiming for better gait on the game field, we changed the locus of gaits just before RoboCup 2005 in Osaka. Before that, we had used triangle locus; it was enabled to walk fast, while it was difficult to reduce vertical oscillation. The problem arose from the friction between feet and the field, because it was difficult to keep landing and leaving angles of the feet. In order to solve this problem, we changed the locus to quadrangle. Fig. 3.3 and Fig. 3.4 show the difference.



`(x1, y1, z1)`	:	landing point
`(x2, y2, z2)`	:	leaving point
`(p, h)`	:	max point
`n`	:	total frame number
`m`	:	grounding frame number

Figure 3.3: Locus (old version)



`(x1, y1, z1)`	:	landing point
`(x2, y2, z2)`	:	leaving point
`(p1, q1, h1)`	:	max point(1)
`(p2, q2, h2)`	:	max point(2)
`n`	:	total frame number
`m1`	:	grounding frame number
`m2`	:	floating frame number

Figure 3.4: Locus (new version)

3.2.2 Creating Gaits Manually

When we create some gaits, we do the following two processes. The first step is a manual process using a gaits mixing tool, and the second step is an optimization process using ceiling camera with Genetic Algorithm. Through the first step, we can find good locus roughly, so that the optimization step costs less times. On the first step, we create locus by setting each parameters that have given in Fig. 3.4. When creating locus, we have to take it into consideration that the gaits are well mixable.

Fig. 3.5 shows a screen shot of the gaits mixing tool, which we developed. The tool enables us the followings.

- Create some basic gaits.
- Mix some basic gaits and check if it works well.
- Mix some basic gaits and define it as a basic gait (if it can be often used on the game field).
- Examine the gaits by sending them to robot via wireless connection.

With this tool, we created some basic gaits (forward, backward, leftward, rightward, clockwise rotation, anticlockwise rotation, etc.), which are mixed

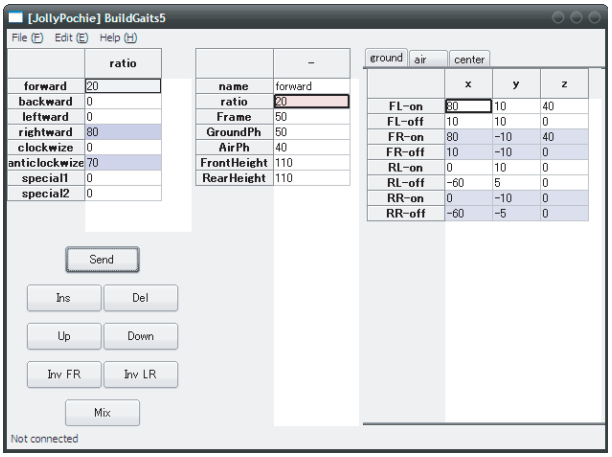


Figure 3.5: Gaits Mixing Tool

each other on the game field. Moreover, we can define a mixed gait as a basic gait if it can be often used on the game field. For example, it is easy to define the following gait.

$$\begin{aligned} \text{anticlockwise warp around} \quad = \quad & \text{forward} \times 0.2 \\ + \quad & \text{rightward} \times 0.8 \\ + \quad & \text{anticlockwise} \times 0.7 \end{aligned}$$

3.2.3 Optimization of Gaits

Genetic Algorithm

In our genetic algorithm, we set the size of populations to be 50, and the mutation probability be 10%. In the initial population, each gene has random values as parameters. For each gene, the robot moves actually and evaluates the fitness score. Among 50 genes, the best 20 genes will be alive in the next generation, and the rests are exchanged with new ones. The crossover operation is executed for these 20 genes, at any points. The fitness score is measured by the distance between the starting position and the end position for a fixed period. By this system, we could develop fast gaits automatically.

Measurement Environment

In order to measure the position of a robot from the ceiling camera, we use two colored balls equipped on the back of the robot, as makers. By these makers, the ceiling camera can determine the location and the direction of the robot easily and accurately. The robot can receive these information

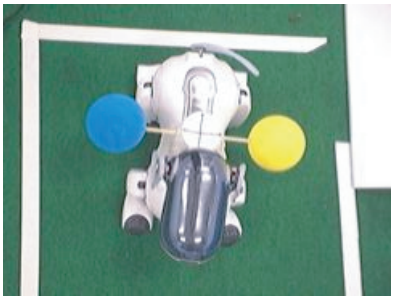


Figure 3.6: Marker AIBO

from the camera via wireless network. Fig. 3.6 shows a robot with the fmarkers.

3.2.4 Mixing Gaits

Gaits which were passed the optimization process are saved as a basic gaits. Actual gaits on the field are created by mixing those basic gaits.

In the past, we have enabled gaits on the game field mixing six basic gaits (forward, backward, leftward, rightward, clockwise, anticlockwise) with some ratio. However, we have felt some shortage of this method, because a full speed walking and an approach walking toward the ball is hardly equals to a normal walking.

We have extended the mixing method and got flexibility; mixability with not only six basic gaits, but an arbitrary number of gaits. The mixed gait $\vec{G}$ can be described as

$$\vec{G} = \sum_{i=1}^{n} \frac{p_i|p_i|}{\sqrt{\sum_{j=1}^{n} p_j^2}} \vec{g_i},$$

and it can be obtained simply from the weighted sum of n factors. Here, $\vec{g_i}$ is the ith basic gait, and p_i is the ratio for the weight of $\vec{g_i}$. In the previous method, n was fixed to $n = 3$: forward-backward, leftward-rightward, and clockwise-anticlockwise. By increasing the number n, any kind of gaits had come to be mixed each other.

3.2.5 Odometer

For each step of walking, the robot has to calibrate its own position especially when no beacons are in its view. For self-positioning, we prepared position calibration curve (Fig. 3.7) as a file. Each gait has its own curve. When some gaits are mixed, those curves are also mixed and the position calibration is done.

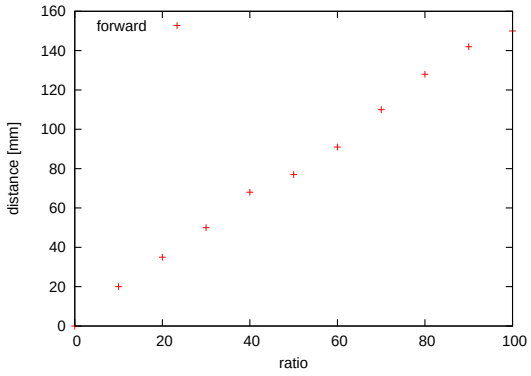


Figure 3.7: Calibration Curve

We realized that the calibration value estimated from the mixed curve does not necessarily correspond to the actual distance of the movement. It will be better to prepare calibration curves by actual measurements even for the mix gaits, instead of simply mixing these curves. It remains to our future work.

3.3 Locomotion

In Section 3.2, locus of gaits and its change were shown. Each gait has landing point, leaving point, and two top points as a locus. To convert this locus (four points) into actual joint angles for walking, we have to interpolate these points. Here, we introduce our interpolate method and the details of gait locus.

3.3.1 Interpolation of Locus

Interpolation step needs the number of interpolate points. Additionally, it is natural for the distance between adjacent interpolated points of an actual gait to be non-uniform; the first half and the last half of the gait motion speed may have some difference. To generate natural walk, we divided a gait into "ground phase", "air phase", and two "other phase"s as Fig. 3.8.

We use HERMITE interpolation to make locus motion to be smooth. In case the locus was interpolated by LINEAR method, its locomotion could be rough. There are many interpolation methods that makes smooth walking: SPLINE (FIRST, SECOND, THIRD...), LAGRANGIAN, HERMITE, and so on. In these methods, SPLINE-THIRD and LAGRANGIAN require more computing costs, whereas SPLINE-FIRST and SPLINE-SECOND lack approximate accuracy. We think that HERMITE interpolation takes a balance of accuracy and cost.

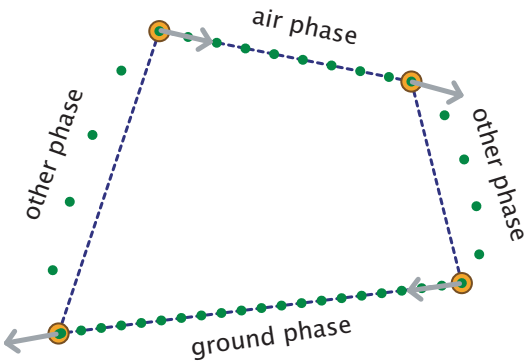


Figure 3.8: Four phases of the Gait

Chapter 4

Vision

In RoboCup, image processing is one of the most essential problems that make a difference of the performance of soccer robots. Last year, our robots consumed most time for image processing. This year, we shaped up our system at first, and introduced new approaches. Our vision system consists of three modules, CDTBox, Vision and DetectBall. CDTBox module detects 8 specific colors from the original 24-bit color image. Vision module recognizes landmark objects, and DetectBall module recognize a orange ball. Figure 4.1 shows the flowchart of image processing. The detail of each module is as follows.

4.1 Color Detection Table

The role of CDTBox is to classify 24-bit colors into specific colors with a color detection table. Therefore, making the table is important for our vision system.

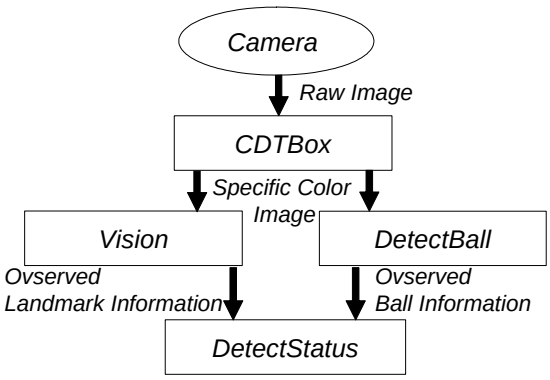


Figure 4.1: The flowchart of Image Processing

Last year, we used the table which consists of 6 threshold values for each specific color to classify colors. It was on the HSV color space. The table was made by deciding H-max, H-min, S-max, S-min, V-max, and V-min values by hands. Making the table needed a long time and much experience because it required a sense of human. Therefore, it was so difficult that we decided each max-min value by hands.

This year, the format of the table is changed. The table consists of the 3D-Matrix which size is $64 \times 64 \times 64$, and it is on the YUV space. The construction of the table takes much more time compared to the above method and is required the sence of human similarly. Hence, we develope a learning tool for resolving these problems, (show Figure 4.2) This learning tool reduces operations required the sense of human, and make the construction of the table easily. The usage of this tool is as follows. First, choose a color in the group (Red Ellipse in Figure 4.2) and click a point in images on the tool (Yellow Boxes). Then the color of the point is learned.Eight specific color, red, blue, yellow, cyan, pink, green, white, and orange, are learned. Those are the color of uniform of robots, goal, pole, field, line and ball. Colors that does not relate with a game is learned as negative samples.

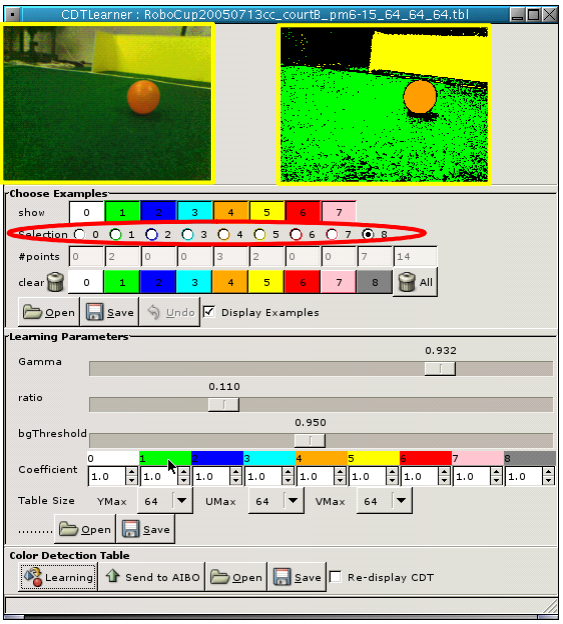


Figure 4.2: Color Table Making Tool

This tool enables us to make the table without the sense of human, because of only choosing pixels of objects in sample images. The problem of this method is which image we choose for the subject of the learning.

Last year, we made the table with the still images taken by standing

AIBO. These are so available for learning the color of specific cases. However, the image that AIBO sees in the game is actually shaky. The colors of objects in shaky images are different from those in still images. Therefore, this year, we make the table with images that taken by moving AIBOs. Figure 4.3 is the tool for taking the images. This tool can capture both still images and the shaky images that are taken by moving AIBOs actually. Hence, it is enable to learn the colors in shaky images, especially the colors of the edges of objects with blurring.

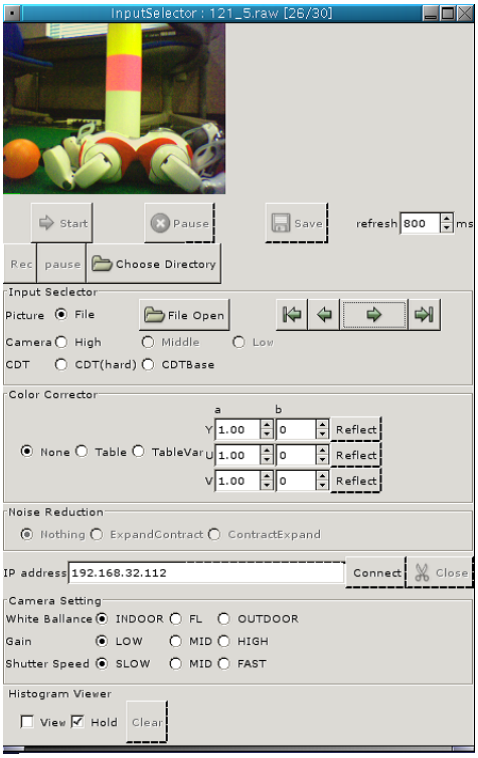


Figure 4.3: Image Capture Tool

4.2 Ball Recognition

DetectBall module recognizes the ball from a image received the CDTBox module. A naive algorithm which counts the orange pixels in the image to estimate the distance does not work well, since the ball is often hidden by other robots, and the ball may be on the corner of the view as shown in Figure 4.4. We show our algorithm to recognize the ball and estimate the position relative to the robot, from a image.

In the image, the biggest component colored by *orange*, satisfying the

following heuristic conditions, is recognized as the ball.

1. The edge length of the component has to be more than 10 pixels, in order to exclude too small components.

2. The ratio of the *orange* pixels to the area of the bounding box must exceed 40%.

3. If the component touches to the edge of the image, the length of longer side of the bounding box must be over 20 pixels.

We use the diameter of the ball to estimate the distance of it. However, the ball is often hidden by other robots partially, and when the robot approaches to the ball, only a part of the ball is visible at the corner of the camera view. Thus the size of the bounding box and the total number of the pixels are not enough to estimate the distance accurately.

Figure 4.4 shows several cases of the diameter estimation. When the ball is inside the view completely (case 1), we regard the length of longer side of the bounding box as the diameter of the ball. When it touches to the edges of the image, we use three points of the components since any three points of the edge of a circle uniquely determine the center and the diameter of it. We choose the these three points as follows. If the centroid of the component exists upper side (case 2), we choose the leftmost and rightmost points on the edges of the bounding box, and calculate the median point on it as the referring points to reconstruct the circle. In the same way, we use top, bottom, and median for left case 3 or 4 (left or right side), and leftmost, rightmost, and median for case 5 (lower side).

4.3 Landmark Recognition

Vision module recognizes landmark objects, for instance, beacons, goals, and lines. It receives specific color images from CDTBox module and outputs the information of the objects to DetectStatus module. DetectStatus module keeps various information of landmarks and ball, and other modules get those information from it. Last year, we use only connected components to recognize objects in the image. This year, we use a hybrid method that consists of connected components and scan lines. That hybrid method reduces false recognitions and calculates accurate distance to the objects. Figure 4.5 shows the flowchart of the object recognition. The detail of each recognition algorithm is shown as follows.

4.3.1 Compute Connected Components

At first, Vision module computes connected components for specific color image that received for CDTBox. Last year, we computed connected components for three colors (*Cyan*, *Pink*, and *Yellow*) in whole region of the

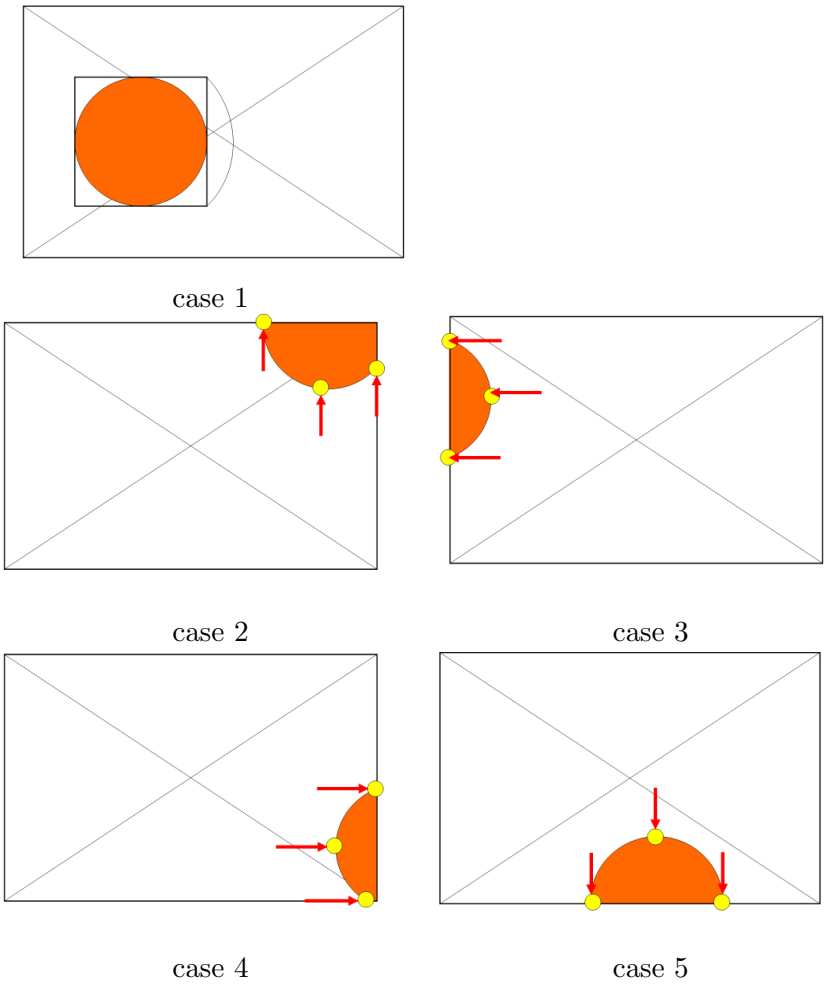


Figure 4.4: Cases of Ball Recognition. Yellow circles on the edge show vertexes of a inscribed triangle. Red arrow presents the direction of the search of vertexes. The diameter of the ball is calculated by using these vertexes.

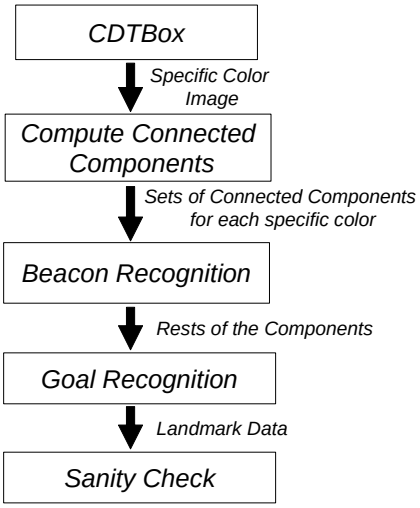


Figure 4.5: The flowchart of the Object Recognition

image, although the most region that was filled any other colors did not relate to the recognition process. It consumed too much computing time. This year, we calculate clipping areas in advance when CDTBox converted the image, and reduce the cost of the computation of connected components. Those calculated components are sorted by the total number of pixels of each color. We store coordinates of the bounding box, the total number of pixels, and the centroid of each component. Those information are used below process.

4.3.2 Beacon Recognition

All beacons consist of a reasonable sized *Pink* section. It is easy for robots to distinguish *Pink* section under the various lighting conditions. Therefore, beacons are the first objects to be recognized. To find beacons, we check in turn whether *Yellow* or *Cyan* components exist near the *Pink* components. The details of the beacon recognition are as follows.

1. The first indication is that the length of the edge of each pink component is longer than 4 pixels.

2. For each pink component, the total number of pixels must over 10 pixels.

3. We calculate a $height$ and a centroid (cx, cy) for each pink component, $height = ((y1 - y0 + 1) + (x1 - x0 + 1))/2$. If a pink component forms a beacon, a cyan or yellow component exists upper or lower side of

the pink component. The estimation point of centroid of the component is calculated according to the $headSlant$ and $height$, $cx_{upper} = cx + height \times sin(headSlant), cy_{upper} = cy - height \times cos(headSlant)$, $headSlant$ is horizontal gradient of the head of the robot. It is calculated by the Kinematics routine. The neighborhood of the estimated centroid is searched. If a cyan or yellow component is found, we go on the next step.

4. We check the ratio of pink pixels to cyan or yellow pixels. If the ratio falls in the range of two thresholds,$maxRatio$,$minRatio$, we store the coordinates of bounding box, the total number of pixels, and the centroids of two components. In addition, delete those components in the memory. Figure 4.6 shows an example of beacon recognition.

5. If the ratio is over or under the thresholds, we check the upper or lower color by the line scanning. This scanning is executed according to the $headSlant$. In fact, It is executed vertically in the view. For example, we find some cyan pixels above and white pixels below, we recognized it as a $CyanPinkPole$. Figure 4.7 shows an example of beacon recognition.

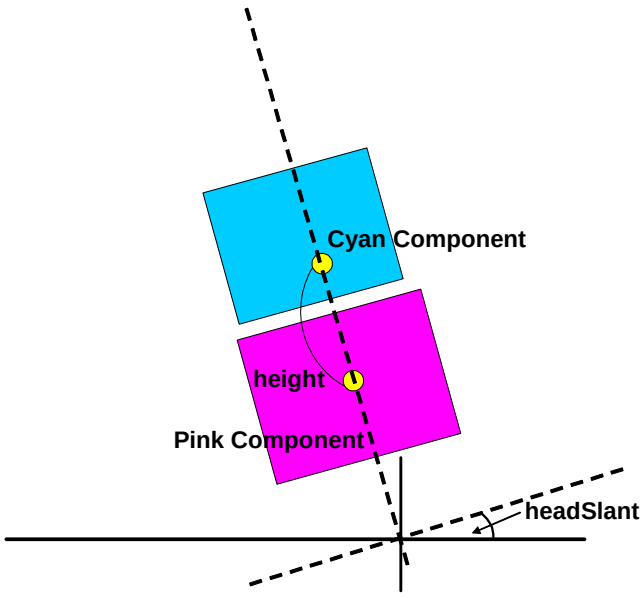


Figure 4.6: An example of beacon recognition (reasonable ratio)

We calculate the distance to a beacon with the below formula.

$$Distance = BeaconConst/DistanceBetweenCentroids + BeaconIntercept$$

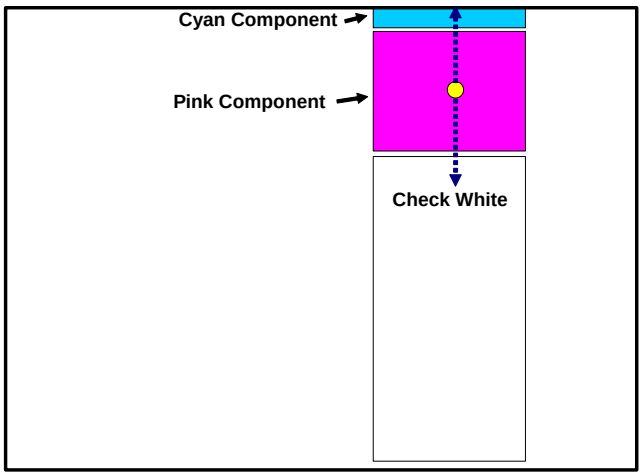


Figure 4.7: An example of beacon recognition (unreasonable ratio)

BeaconConst and *BeaconIntercept* are calibrated by measuring the length between centroids of the two components that form the beacon. we use that length for estimating the distance to a beacon, because it is more accurate than using the total number of pixels of components for estimating. Where the ratio of two components is unreasonable, we use the height of bigger component instead of the length, because the length of centroids of two components equal to the height of a component by rights.

4.3.3 Goal Recognition

As is the case with the beacon recognition, we use a hybrid method for recognizing Goals. Goals consist of the component that has same colors as beacons. It is difficult to distinguish whether that colored component belongs to the goal or a beacon. Hence we executed the beacon recognition before the goal recognition. Since components that belong to beacons are picked out in advance, remaining components become targets of the goal recognition process. The detail of the recognition algorithms is shown as follows.

1. At first, we check the total number of pixels of each remaining component colored *Cyan* or *Yellow*. it must exceed 150 pixels because of excluding too small components.

2. The components that have pink components just above are also excepted. These components have possibilities to be beacons yet, because the beacon recognition that has already executed may fail to recognize some beacons.

3. Above step excludes the components that have some possibilities as a beacon, but in the case of closing to the beacon, robots face a big *Cyan* or *Yellow* component. To except those components, we check below colors of the components using the scan line method vertically. If the components have enough white pixels below, these components are excluded from the candidates. It may also remove correct components where other robots close in front of the robot. In this case, since the visibility of the robot is poor, the result of the recognition is trustless.

4. We estimate a distance to the goal according to the height or width of the goal. Six scan lines calculate a length of the width or height of the goal horizontally and vertically. Figure 4.8 shows an example of the scan lines.

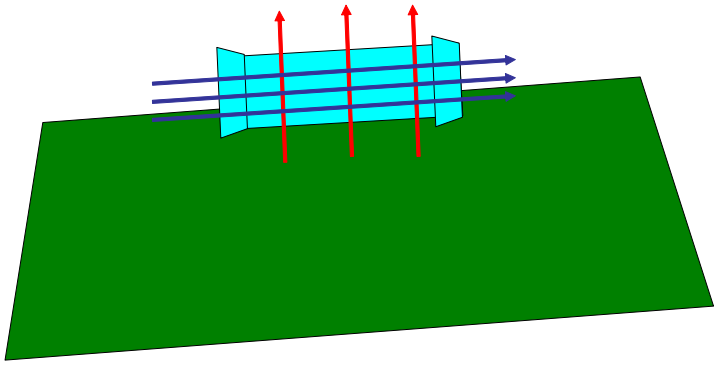


Figure 4.8: Line Scanning for the goal

5. The biggest component that satisfies above conditions is recognized as a goal. We store coordinates of the bounding box and the total number of pixels.

4.3.4 Sanity Check

In the above processes to recognize landmarks, some landmarks may be misunderstood. We check the accuracy of landmarks according to their location. For instance, if we recognize the beacon which is located 1000 mm high, it is removed for recognized landmarks. We set two thresholds (one is a minimum height and another is a maximum height) for each landmark and exclude landmarks that have illegal heights.

4.4 Line Recognition

We developed two methods for recognizing filed lines. Those are used for helping self-localization. Both methods detect the border between field and field line and are based on line scanning. Each scan line searches the point of the edge of a line for a image from bottom to top. The difference of two methods are input images. One receives raw images, another receives CDT images. In first method, we define the point, in which the value of Y increases significantly, as a field line border. In second method, the point that has white pixel above and green pixel below defines the border. We present a result of our line recognition method in Figure 4.9 to Figure 4.11.

Both of them have been inadequate. they cannot recognize the unique attribute of line , for instance, a center circle and corners of penalty areas.

Figure 4.9: The image that captured by the camera

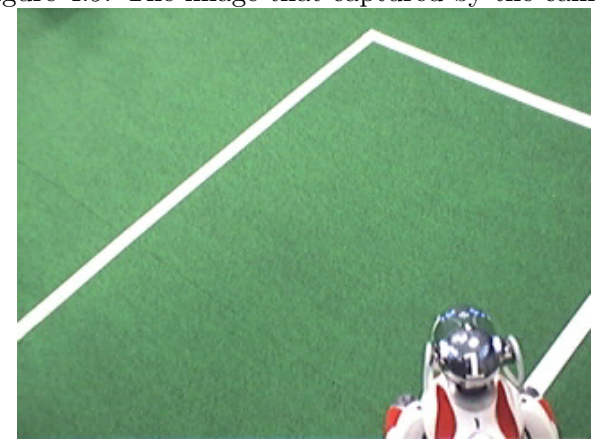


Figure 4.10: The position of a robot and filed lines

Figure 4.11: The result of line recognition

Chapter 5

Other Modules

5.1 Localization

Localization is an important task for automatic positioning, shooting and team coordination. We use Monte-Carlo localization technique for self localization and ball localization. Kalman filters are another well-known methods to localize a position of robots and a ball, and are more efficient than the Monte-Carlo localization. Because of the size of the field being bigger than last year, no more than one or two landmarks are observed at one time in AIBO's sight. In this condition Monte-Carlo localization is more suitable than Kalman filters.

5.1.1 Self Localization

Self localization module is a program which takes as input any of distances and directions of landmarks, e.g. poles and goals, and calculates as output the location of a robot. In this year, however, the change of the field reduces the number of landmarks in AIBO's sight. It is hard to determine precisely where a robot is placed from only one or two landmarks. Then we use the white lines as landmarks in addition to poles and goals.

Our Monte-Carlo self localization algorithm is patterned after German Team 2004 [2]. First we implemented the algorithm by using Processing which is an open source programming language and environment based on Java [3]. Processing have an editor window and allow a programmer to develop programs in a try and test manner like scripting languages and to display results graphically by just clicking the "run" button. We tested several implementations of the Monte-Carlo localization as shown in Figure 5.1, and then translated it into a C++ version as module SelfMCL.

Calculation of the Monte-Carlo localization takes time proportional to the number of locators which consist of a position (x, y) and a rotation θ. We decided the number of locators was 1,000.

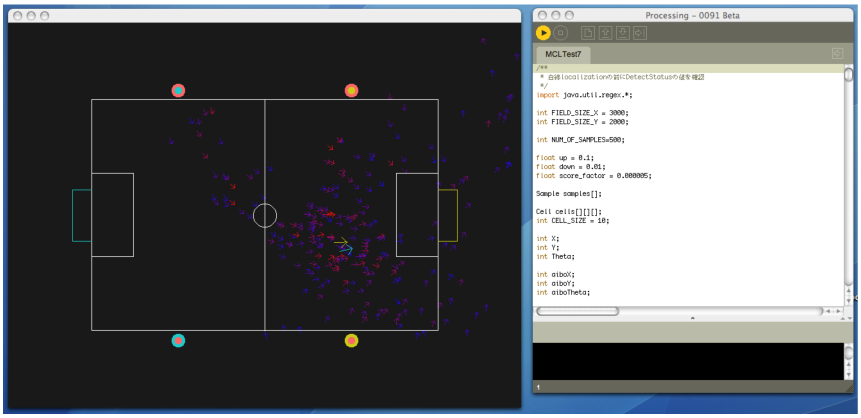


Figure 5.1: Monte-Carlo self localization test environment on Processing.

Figure 5.2 shows classes for the Monte-Carlo self localization. Monte-Carlo Localization module is a subclass of Odometer which is used by GaitsMixerOd and updates the position of a robot whenever the robot walks. We developed three kind of implementations for Monte-Carlo localization: SelfMCL without using white line, SelfMCL2 and SelfMCL3 with using white line.

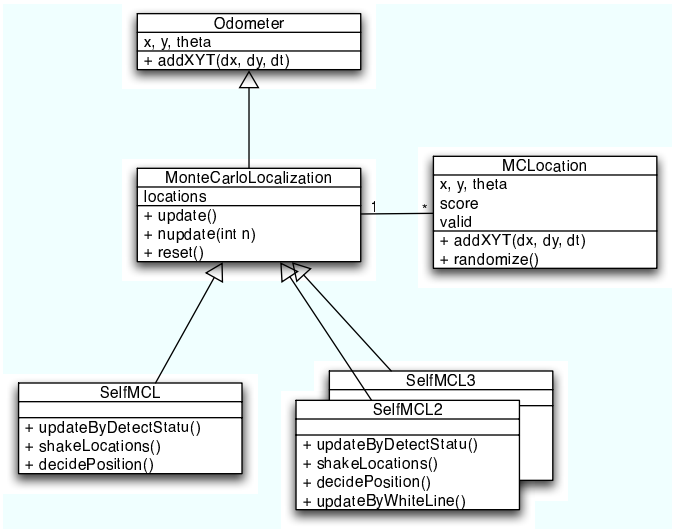


Figure 5.2: Monte-Carlo self localization classes.

5.1.2 Ball Localization

Last year, our robots play soccer based on the location of the ball which is observed directly. However, in soccer game, the position of the ball is fluctuating. It is difficult for the robots to keep the correct position of the ball only based on the direct observation because the robot has narrow view so that it frequently lost the ball. Then we are trying to develop a system to infer the position of the ball even when it is invisible, based on Monte-Carlo method.

A Monte-Carlo ball localization algorithm is similar to a self localization one and is very simple, which takes as input only the position of the ball. However, it needs great accuracy for locating the position of the ball to kick the ball. If the amount of an error for locating the position of the ball was more than 2 cm, robots cannot even touch the ball, much less kick it. Therefore we gave precedence to accuracy over stability for adjusting parameters of ball localization.

For self localization the scores of locators are not changed when none of landmarks are observed. However, for ball localization the scores of locators are decreased even when the ball is not observed. The ball localization works like the spectrum of human eyes. When robots lost the ball temporarily, Monte-Carlo ball localization keeps the position for a little while. This function of the ball localization had a great effect especially for chasing the ball with other robots because in the situation ball is often hidden.

We implemented ball localization by using Processing at first as for self localization, and translated it into C++ version after testing and adjusting parameters (Fig. 5.3).

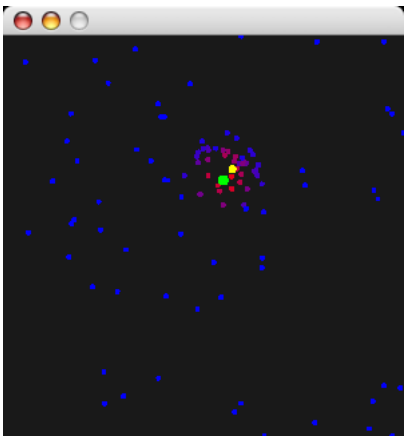


Figure 5.3: Monte-Carlo ball localization test on Processing.

5.2 Sensor

AIBO have many kind of sensors, for example, sensors for acceleration, the angles of joints, touching on the back, and so on. For the sensors we implemented many kind of modules which manipulates data structures of sensors in OPEN-R. The modules allow us to easily measure the values of sensors without OPEN-R programming knowledge only by calling get methods.

5.2.1 AccelSensor

AccelSensor module supports acceleration sensors of x-axis, y-axis, and z-axis. In addition to the sensor values, AccelSensor decides the posture of the robot, which is normal, upside-down, left-down, and so on.

5.2.2 HeadJointAngleSensor

HeadJointAngleSensor supports the values of joint angles of the head, *e.g.* the tilt of the neck, the pan of the head, and the tilt of the chin (for ERS-7) / the roll of the head (for ERS-210/220).

5.2.3 TouchSensor3

TouchSensor3 supports touch sensors of head, chin, back, and toes. The value of touch sensors are whether it is being touched or not, and if touched, the continuous time of it. Moreover, when it is touched and released, a clicked flag is set. Once the clicked flag is read, the flag is reset.

5.2.4 PSDSensor

PSDSensor supports positioning sensing detectors of far head, near head, and body on ERS-7 (only head on ERS-210/220). It reports the distance from an object in front of the robot.

However, the PSD of body does not work because of our low and forward-bend standing posture, it always reports the minimum value.

5.2.5 LegsSensor

LegsSensor supports the load and voltage values of motors of legs. The values may be useful for detecting a collision. Unfortunately, we did not use the values in soccer robots, but used these in some experiments for optimization of locomotions.

5.3 Network

This section describes modules for networking. As mentioned in Chapter 1, networking is not processed JPObject, that is the main program of Jolly Pochie, but by outside programs TCPServer, UDPServer, and GameController. Networking modules are the proxy of the programs.

5.3.1 RemoteControl7

RemoteControl7 is a generic remote control module for a mainly debugging purpose. When the module receives a message, the message is transferred to an appropriate remote control module selected by a first character of the message.

A lot of remote control modules has been developed: RCAccelStatus, RCBallMCL2Monitor, RCCapture, RCCheckDistance, RCCheckOrient, RCDetectBall, RCDetectStatus, RCFixedMotion, RCGaitsMixer, RCJointAngle, RCLuaScript, RCMonteCarloMonitor, RCMotion, RCParaWalk, RCPositioningWalk, RCProfile, RCRecognize, RCSlant, RCSonarData, RCWalk.

Especially RCLuaScript is the most useful. It is able to process all the thing that is able to process in Lua. New remote control modules are merely created hereafter.

5.3.2 UDPCom2

UDPCom2 is a simple module for communicating through a UDP channel with other robots. Messages from GameController are not processed by this module, UDPCom2 is used for team play.

All messages are processed by Lua, *i.e.* the messages are Lua scripts. Therefore certain robot can totally control other robots. In this year, we use the message for changing the values of variables which represent the status of other robots. However, this mechanism seems to be week against accidentally troubles. We are plan to change the mechanism into more robust one in next year.

Chapter 6

Strategy System

6.1 Action Callback

In our framework, we can call a callback function soon after a certain action of a robot is finished. We call this callback "action callback". Using action callback functions, we can reserve processes executed after actions in advance. Action callback is mainly used when a robot swings its head from side to side, and performs a shoot motion. The following is a script that a robot swings its head to an angle 80, and to an angle -80. The function `basicMotion:swingHead(tilt1, pan, tilt2)` makes a robot swing its head to a position specified by the three angle `tilt1`, `pan`, and `tilt2`. If we give a certain function to the fourth argument, we can set the function as an action callback function. If the fourth argument is a string, a transition to the state, specified by the string, is occurred at the time of action callback. We use this *state-transition-callback* method when we write a script by using state tree in the next section.

```
require "JPLib/Units.lua"

function init()
   basicMotion:swingHead(0, d2ur(80), 0,
                         function()
                           basicMotion:swingHead(0, d2ur(-80), 0)
                         end)
end

function mindNotify()
end
```

Table 6.1: An example of hierarchical states in four legged robot league.

playing	search	swing	left1
			right1
			...
		turn	left1
			right1
			...
		walk	area1
			...
		...	
	approach	near	
		far	
		...	
	shoot	forward	
		left	
		...	
	support		
	...		
ready			
set			
penalized			
finished			

6.2 State Tree Machine

We had used a normal state machine to create strategies so far, because we could treat strategies as simple models using a state transition method. However, As a robot program became complex, it was difficult to understand all of the states of the state machine. Therefore, this year we use a state machine with a tree structure in order that we can break a large program into small chunks that we can easily understand. We call the tree structure "state tree", and call the state machine using the state tree "state tree machine". The state tree is suitable for soccer strategies in RoboCup, because we may think hierarchical states as shown Table 6.1. This hierarchical states should be represent as a tree structure.

Now, let us show the outline of the implementation of a state tree machine. First, we create a state tree as shown Figure 6.1 that represent the states as shown Table 6.1. Secondly, we create functions that execute processes corresponding to each node. We call this function "node function". a state of the state tree is expressed in the path from the root to a certain node in the tree. Finally, we call each node function from the root to the node in order to execute the process for the state.

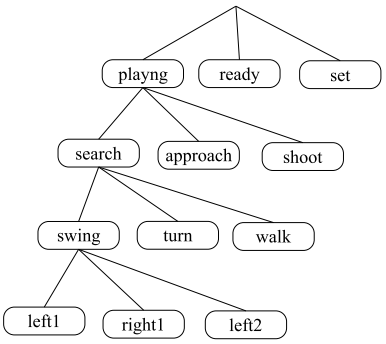


Figure 6.1: A state tree that represent the states as shown tab. 6.1

A state tree machine is simply implemented in Lua, because we can easily make a tree structure by using a table structure in Lua. Additionally, it is expected to be able to create a state tree, and define node functions at the same time as follows, because the functions in Lua are first class values.

```
playing = function() ... end
playing.search = function() ... end
playing.search.swing = function() ... end
playing.search.walk = function() ... end
...
```

However, a syntax error obviously occurs in the second line from the top, because the function `playing` is treated as a table. In order to remove this error, we use the metatable method in Lua. Using a metatable, we can create a table that can call a node function at the time of function call. The following procedures show how to use metatables.

1. We define the node function with node name `playing`. The node function is always appended to leaf in the state tree. The following code is equal to the first line from the top in the above code.

   ```
   function playing()
     ...
   end
   ```

2. The value of the function `playing` is copied to temporary variable `tmp_func`.

   ```
   tmp_func = playing
   ```

3. The function `playing` is initialized as a table.

   ```
   playing = {}
   ```

4. We set a metatable so that the table `playing` can call the function `tmp_func` at the time of the function call of the table `playing`.

```
setmetatable(playing,{__call = tmp_func})
```

In the same way, we must execute the above procedures for all internal nodes. Lastly, if the state is "playing.search.swing", we have only to call the table `playing()`, the table `playing.search()`, and the function `playing.search.swing()` in order from the front. The following functions were actually implemented for the state tree machine.

- `stree:setState(state)` allows the state of a state tree to transit from current state to the state `state`.
- `stree:doAction()` parses the state, which is a path in the state tree, and call all node functions in order from the nearest node to the root.
- `stree:new(func)` replaces the function `func` with the table that can call the function `func`.

The following script is an example for a robot to search a ball, moving from side to side, and swinging its head. The function `mindNotify()` calls only node functions described in the state. If the fourth argument `func` of the function `basicMotion:swingHead(tilt1, pan, tilt2, func)` is not a function but a string, the function `stree:setState(func)` is called as a callback function.

```
require "JPLib/Units.lua"
require "JPLib/STree.lua"
require "JPLib/Vision.lua"
require "JPLib/CMotion.lua"

function init()
  count = 0
  ds = 1.0
  stree:setState("walk.swingLeft")
end

function mindNotify()
  stree:doAction()
end

function walk()
  count = count + 1
  if count > 100 then -- it is approximately 4000 ms.
    count = 0
```

```
    ds = ds * -1.0
  end

  cmotion:walk(0, ds, 0)
end

stree:new("walk")

function walk.swingLeft()
  stree:setState("walk.swingWait")
  basicMotion:swingHead(0,d2ur(80),0, "walk.swingRight")
end

function walk.swingRight()
  stree:setState("walk.swingWait")
  basicMotion:swingHead(0,d2ur(-80),0, "walk.swingLeft")
end

function walk.swingWait()
  visionLib:detectBall()
end
```

We can describe processes for the game states (e.g. to change LED colors, to play sounds, and to check whether a robot has rolled upside down or not) in node functions of nodes whose depth is one. We can easily write robot scripts that follows the directions of the game controller, by loading the game controller library. If we want to write a strategy in the *playing* state, we have only to write in function `playing.run()`. The following script prints the game states.

```
require "JPLib/Syslog.lua"
require "JPLib/STree.lua"
require "JPLib/GameCtrl.lua"

function init()
end

function mindNotify()
  stree:doAction()
end

function initial.run()
  print("it is initial state.")
end
```

```
function ready.run()
  print("it is ready state.")
end

function set.run()
  print("it is set state.")
end

function playing.run()
  print("it is playing state.")
end

function finished.run()
  print("it is finished state.")
end

function penalized.run()
  print("it is penalized state.")
end
```

The *set* state, the *ready* state, and the *finished* state may be common in all robot scripts for soccer players. Moreover, we may mostly develop strategies of the *playing* state. Therefore, we write independently the processes for each state, and marge these scripts into a script for the *playing* state, as if to graft subtrees onto a main tree. In order that the robot script in which we only described a process of the *playing* state becomes a soccer player script following the directions of the game controller, we have only to load the scripts for other states, as follows.

```
require "JPLib/Syslog.lua"
require "JPLib/STree.lua"
require "JPLib/GameCtrl.lua"

-- graft subtrees
require "Ready6/ready.lua"
require "Set3/set.lua"
require "Final3/finished.lua"

function init()
end

function mindNotify()
  stree:doAction()
```

```
end

function playing.run()
  print("the game strategy is written here.")
end
```

In conclusion, by using a state tree machine, we could more easily create huge complex programs than using a normal state machine. The reason is that we could divide between low-level layer and high-level layer in a strategy, that in high-level layer we did not need to write the processes written in low-level layer, and that we could create small subtrees, and graft each subtree onto a main tree. However, the number of nodes in a state tree machine may be larger than in a normal state machine, if we describe a certain strategy by each method. As a result, it seemed that the program written by using state tree machine easily caused logical errors. The most common error is a transition to an unknown state. If this error occurs, a robot will stop working even during a game. Therefore, in the coming years, we may have to use a state tree machine to describe the outline of a strategy, and use logic base or rule base system to describe the detail.

6.3 Behavior System

It is difficult for programmers to describe a whole strategy with a state tree machine, because the number of the nodes in a state tree machine increases rapidly even if a state tree machine is more convenient than a normal state machine. Therefore, we put some processes into a medium-scale procedure to make a whole strategy easy to understand. We call this medium-scale procedure "behavior". A behavior is like a subtree in a state tree as shown Figure 6.2. Behavior system is developed based on a state tree machine, but a behavior differs from a state tree machine on the point that a behavior has an independent state of a state tree machine.

We implement a behavior as a function that can use another state tree. As a matter of convenience, we call the state tree to describe a whole strategy "global state tree", and call a state tree used by each behavior "local state tree". By using local state trees (behaviors) in the global state tree, it is easier for programmer to describe a strategy than before, because both the global state tree and the local state trees will have a small number of the nodes.

Every time we call a behavior in the global state tree, the behavior works changing the state in the local state tree. Since a behavior is completely independent of the other behaviors, we do not have to know the state of the local state tree in the behavior. Also, a certain transition from one state to another state does not result in an unexpected drawback. If a behavior is

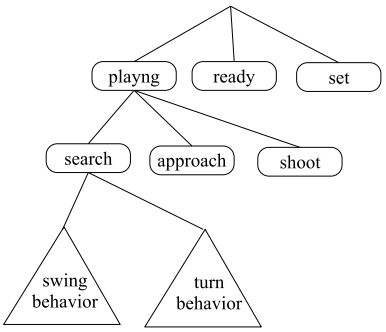


Figure 6.2: The outline of the behavior.

normally finished, or causes an error, we can notice it by a returned value of the behavior.

Behaviors are implemented by using a metatable in a similar way to the global state tree. Using a metatable, we can treat a behavior as both a function and a table. A behavior, in fact, is a table under which we can create node functions. This table means the root of a local state tree. When we call a behavior as a function, the local state tree under the behavior internally works changing its own state. Although a behavior is similar to a state tree machine, we must use `setState(str)` in the behavior instead of that in the state tree, and tell the main program the end or error of the behavior if necessary.

The following script is a behavior that a robot swings its head from side to side. The function `behavior:newBehavior(str)` creates and returns a behavior named `str`. In the following example, The variable `b` means `behavior.swingHead`, and The function-definition`b:init()` means `behavior.swingHead.init(self)`. When we call `behavior.swingHead:init()`, the global variable `behavior.swingHead` is assigned the local variable `self`.

```
require "JPLib/Behavior.lua"

b = behavior:newBehavior("swingHead")

function b:init() -- means behavior.swingHead.init(self)
  -- self means behavior.swingHead
  self:setState("swingLeft")
end

function b:swingLeft()
  basicMotion:swingHead(0, d2ur(80), 0,
                        function()
                          self:setState("swingLeft")
```

```
                          end)
end

function b:swingRight()
  basicMotion:swingHead(0, d2ur(-80), 0,
                          function()
                            self:setState("swingCenter")
                          end)
end

function b:swingCenter()
  basicMotion:swingHead(0, 0, 0,
                          function()
                            self:setState("finish")
                          end)
end

function b:finish()
  return BEHAVIOR_STOP
end
```

Next, We show how to call behaviors. The following is an example that call the behavior `behavior:swingHead()` defined above. Because the behavior returns the constant value `BEHAVIOR_STOP` at the finish time, we can change the global state from `"swing"` to `"swingFinish"` in that time.

```
function swing()
  if behavior:swingHead() == BEHAVIOR_STOP then
    stree:setState("swingFinish")
  end
end
```

In conclusion, behaviors are easily created by state tree method, and simply used by function call method. Since each behavior should be not so large, the difficulty using only state tree machine does not very often occurs. However, behaviors cause another difficulty of where behaviors begin and end. Should a search behavior include the process of checking whether a robot found a ball? Should the process of approaching a ball be separated to several behaviors? Should a one-robot behavior be different from a multi-robots behavior? Frankly, we do not swear which way is best. A continuous examination of the behavior system may teach us the answer of that.

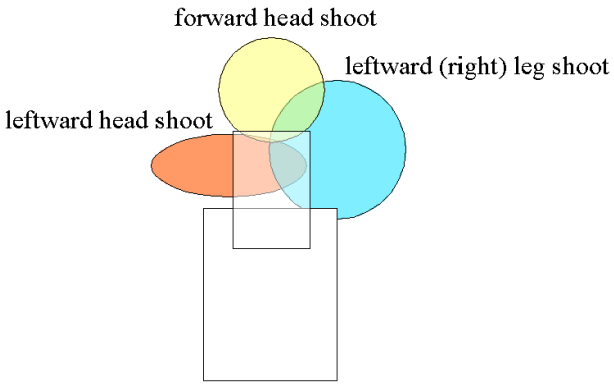


Figure 6.3: The shootable areas of three shoot.

6.4 Electric Field Approach

We tried using *Electric Field Approach* (EFA) [4] proposed by Team Chaos [6] in hopes of creating a model of a complex strategy. The EFA is a kind of potential field approaches. A robot nicely plays soccer automatically, only by placing positive and negative artificial charges on strategically important locations including robots themselves, and by working the robots to increase the potential at the ball position.

Using EFA, we can simply implement a complex strategy, and easily extend the strategy by adding actions. As regards shoot actions, we do not have to consider the shootable area, even if the shootable area of each shoot is different as shown Figure 6.3. If the shootable condition is defined so that a robot can kick or head a ball in the shootable area, the robot can select the best available shoot without fail. However, it is difficult to confirm which action is being executed, if we use many actions. That is to say, a strategy program becomes almost a "black box". Hence, we can not adjust the program in detail. Moreover, an EFA strategy has a possibility of shooting to wrong direction in big situation, because locational errors near opponents' goal is not negligible as shown Figure 6.4. As a result, we did not use the EFA method in the RoboCup 2005 competition. Nevertheless, there may be several possible solutions.

6.5 Team Play System

In this year, we have developed a team play system that enables robots to work cooperatively. Using this system, robots can dynamically change its own role. For example, while one robot is chasing the ball, another robot can move to a good position for a passed ball, and wait for the opportunity of own shot. Moreover, robots can tell each other about the position of the

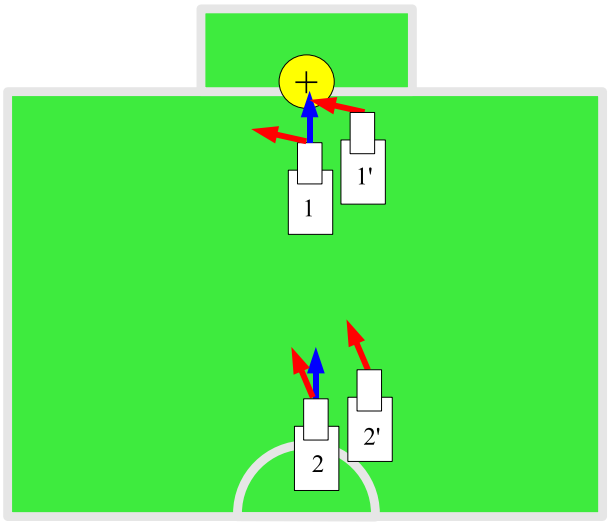


Figure 6.4: The shoot direction error of the robot 1 is larger than that of the robot 2, although the position errors of both robots are same. The robots 1 and 2 show actual positions, while the robots 1' and 2' show virtual positions by localization method. A blue arrow shows a ideal shoot direction . A red arrow shows a calculated shoot direction.

ball. The ball must be found more easily by all robots than by one robot.

teamPlay:init()

This function has some variable. "teamPlay.Info_ball" receives the ball information. "teamPlay.udp_count" is counter. When it more than "teamPlay.UDP_WAIT", a player sends the information. "teamPlay.current_roll" is a situation of a player. If the player is nearest to the ball in our team, we call this situation of the player "Active". And if the player is not nearest to the ball in our team, we call this situation of the player "Passive". "teamPlay.current_roll" is given "Active" or "Passive". "teamPlay.num" is the number of the nearest player to the ball in the team. "teamPlay.my_dist" is a distance of the ball. Like this the function has these informations.

```
function teamPlay:init()
   teamPlay.ball_Info = x=0, y=0
   teamPlay.udp_count = 0
   teamPlay.current_roll = "Active"
   teamPlay.num = -1
   teamPlay.my_dist = -1
end
teamPlay:init()
```

set*()

A metatable makes this function be called every time after Play() is called. The function has a function that send player's information and a function that receive other player's information.

```
function teamPlay:set()
   teamPlay.playing_func = getmetatable(playing).__call
   teamPlay.playing_derived = function()
 teamPlay.playing_func()
 teamPlay:sendMessage()
 teamPlay:getInfo()
 return teamPlay:changeRoll()
      end
   setmetatable(playing, __call  = teamPlay.playing_derived )
end
```

sendMessege()

This function send a player's information and has a counter for adjusting an interval. If the interval is not enough, all players do not send an information in one cycle. The function checks an information. It erases an old information which is sent by a penalized or down player.

```
 teamPlay.udp_count = teamPlay.udp_count + 1
   if teamPlay.udp_count > teamPlay.UDP_WAIT then
   teamPlay.udp_count = 0
   teamInfo:sendMyInfo()
   teamInfo:checkInfo()
end
```

getInfo()

This function calculates a position of the ball. If a player do not watch the ball, it receives the information and calculates a position of the ball from the ball information. If all players do not watch, the ball information is initialized.

changeRoll()

This function works to judge whether a player is nearest to the ball. If the player is nearest to the ball in our team, we call this situation of the player "Active". If the player is not nearest to the ball in our team, we call this situation of the player "Passive". When a player's situation changes, the function gives the player "Active" or "Passive" (Fig. 6.5).

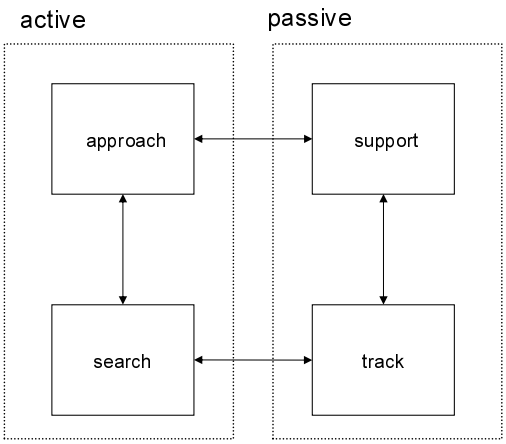


Figure 6.5: Active and Passsive.

setStateActive()

This function changes a player's state when it is nearest to the ball in our team.

setStatePassive()

This function changes a player's state while another player in our team is nearer to the ball than it.

Chapter 7

Strategies for Jolly Pochie 2005

7.1 Attacker

Attackers can change its own state depending on the game situation. There are four states "Approach", "Support", "Track", and "Search" (Fig. 7.1). Each state is selected by an attacker corresponding to whether two conditions is true or not. One is that the attacker is seeing a ball. The other is that the attacker is the nearest one to a ball in our team players.

Before, the attacker in state of "Approach" judges whether it watches the ball at all time, so it makes an useless state transition. Now, it judges this when we want its state to change, so we can control the state transition. And then if the attacker loses sight of the ball a moment, it can hold or chase the ball. However that method has a disadvantage that its response becomes slow when it loses sight of the ball.

As the attacker sometimes localize, it clashes with the other player and its localization has an error. So we think an approach speed is more important than a localization. If the attacker entirely localizes, it wonders in the field. Then it swings its neck and checks the direction of the goal while approaching the ball. And it often localizes when it does not watch the ball.

Our system does not have the player recognition. The moving attacker clashes with the other player. For preventing the attacker from clashing with the other, we made the communication system. If there is an approaching player which is the nearest to the ball, the other player does not approach the ball and does not disturb it. Now the attacker in state of "Support" only stops there and watches the ball at a position a little away. Because the communication system has a problem that two player approaching the ball send the information the different time, then comparing two players' ball distance does not work well and they approach the ball.

On the other hand the player's localization often has an error, so the in-

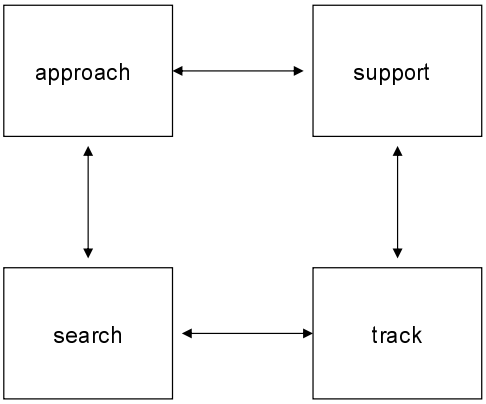


Figure 7.1: Changing states in Attacker.

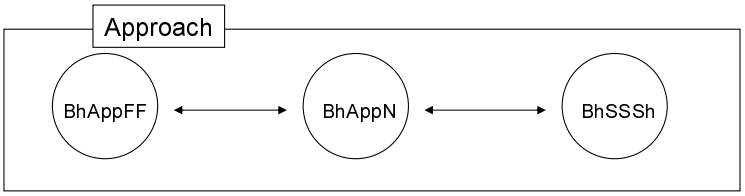


Figure 7.2: Behaviors in approach.

formation is inaccurate. The error of the ball information makes the player's searching less efficient. The player in our team takes much time for finding the ball than in other team. We have to improve behaviors in the "Search" and the communication system. Like this the attacker in our team has many problems. A correct localization, the player recognition and improving behaviors makes the player good.

We explain 4 states which the attacker have as follow.

7.1.1 Approach

When the player watches the ball and it is the nearest in our team's players, a player's state is "Approach" (Fig. 7.2). It is the purpose of the state that the player approaches speedy and shoots the ball in the direction of the goal accurately. When the ball distance is far, BhAppFF makes it approach the ball. When the ball distance is near, BhAppN makes it stop. When the player stops in front of the ball, BhShSS makes it shoot the ball.

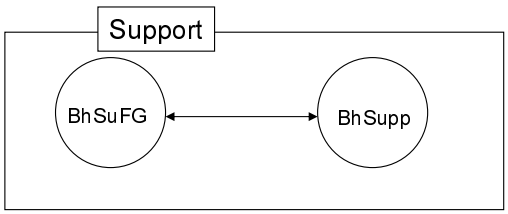


Figure 7.3: Behaviors in support.

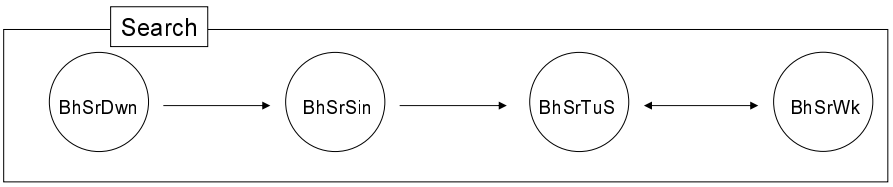


Figure 7.4: Behaviors in search.

7.1.2 Support

When the player watches the ball and is not the nearest in our team's players, a player's state is "Support" (Fig. 7.3). It is the purpose of the state that the player supports and does not disturb the other player which is the nearest to the ball. When the ball distance is far, BhSuFG makes it approach the ball. When the ball distance is near, BhSupp makes it walk for the supporting position.

7.1.3 Search

When the player does not watch the ball and other player is nearer than it, a player's state is "Search" (Fig. 7.4). It is the purpose of the state that the player finds the ball speedily and efficiently. First BhSrDwn makes the player search for a near place. Next BhSrSin makes the player search for a far place. If the player does not find the ball, BhSrTuS and BhSrWk work alternately. So the player walks for a searching position and turns. And it walks for a next searching position and turns until one player in our team finds the ball.

7.1.4 Track

When the player does not watch the ball and is not the nearest in our team's players, a player's state is "Track" (Fig. 7.5). It is the purpose of the state that the player receives the ball information and finds the ball. As it does not watch the ball, it receives the ball information and searches the ball.

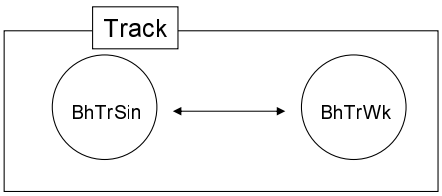


Figure 7.5: Behaviors in track.

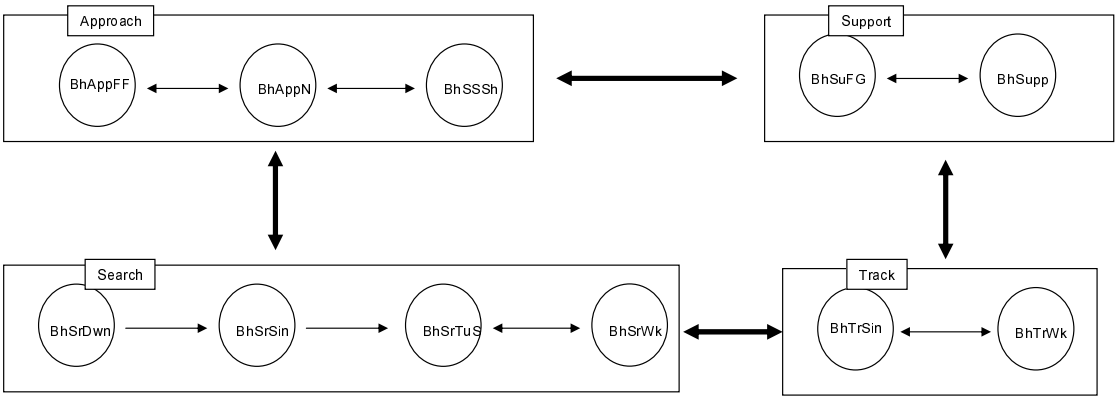


Figure 7.6: Changing states in Attacker.

First it localizes and turns in the direction of the ball. Second BhTrSin makes it swing its neck. Next it walks for the ball.

When these detailed each state and state transition are shown, it becomes the following(Fig. 7.6).

7.2 Old Attackers

We made many version of attackers, and improve attackers little by little. In this section, We show several old attackers scripts that have been developed before making the final attacker script.

7.2.1 At7

This attacker is made at RoboCup in Japan. It did not have communication system and it did not use behavior. So program is written in the state, but its state is so complicated that our programming is difficult. It had only two states; Its state is approach when it watches the ball and its state is search when it does not watch the ball. It in the state of approach turns to the ball and approaches the ball in a top speed. On the other hand it in the state

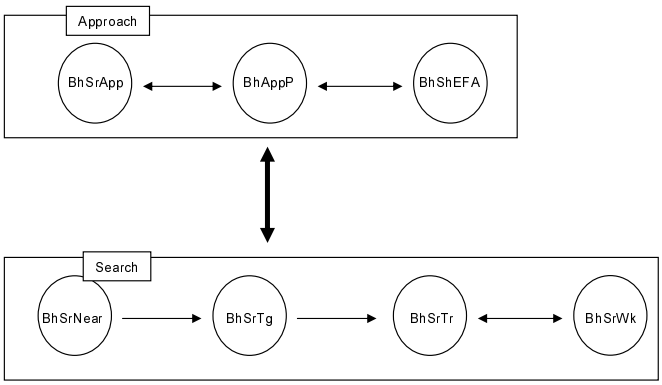


Figure 7.7: Changing states in AtHk.

of search swings its head and moves backward. If it does not find the ball, it turns around and moves the searching position.

7.2.2 AtHK

We imported behavior which was written a part of programs into this attacker(Fig. 7.7). So we could check behavior and our programming was easier than before.

7.2.3 AtHK7T

We improved behaviors and made behaviors. Our team's communication system was imported into this attacker, and then the attacker's state became four states(Fig. 7.8). But JPLib/TeamPlay.lua does not exist and a program for communication system is written in a function playing_derived which is called next to the function playing. In this time the state of support and the state of track are simply and equality. The player in this two states moves between the ball and the searching position whether it watches the ball or not. If it does not watch the ball, it calculates the ball position from the information which is sent by the our team's player. For checking self localization the player in the state of the approach swings its head right and left while approaching the ball. So swinging it's head takes waste time and makes it's moving unstable.

7.2.4 At9ts

This attacker which was made when we arrived at Osaka used JPLib/TeamPlay.lua and TeamPlay was equality with AtFinal0. It differs AtFinal0 in "Support" and "Track" (Fig. 7.9). The attacker in the state of "Support" does not

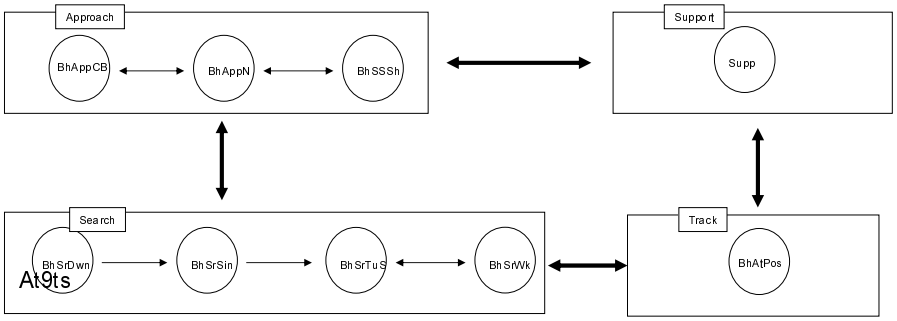


Figure 7.8: Changing states in AtHk7T.

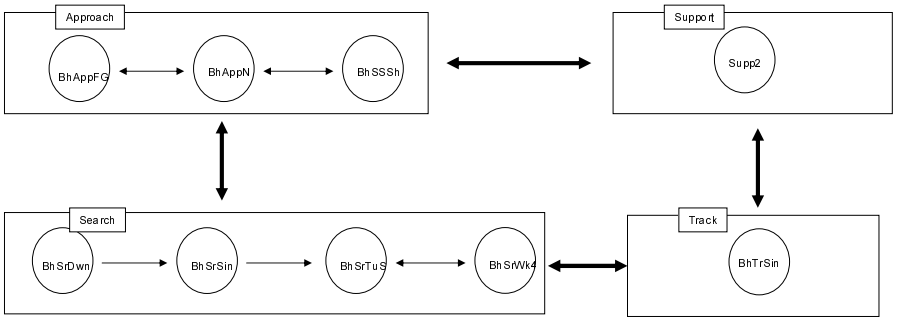


Figure 7.9: Changing states in At9ts.

approach. It moves slowly the position which is between the ball and the searching position and wait at there. But in our team only one player approaches and the other players watch the ball on a far place, so it is disturbed by the opposing player and does not approaches the ball.

7.3 Defender

As is the case of attackers, defenders can also change its own state depending on the game situation the process of each state of defenders is only different from that of attackers. By necessity, defenders should be more difensive than attackers.

The defender always does not attack the target goal and it stays in our team's territory. It stops and walks for the guarding position that is between own goal and the ball, when the ball is out of our team's territory and in own goal's area. And if it arrives at the guarding position, it localizes. So its localization is more accurate than the attacker.

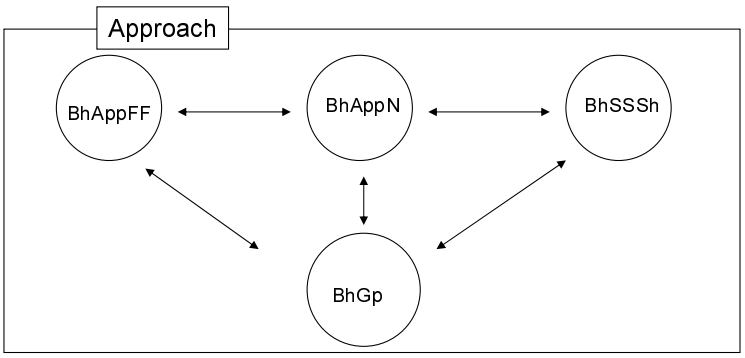


Figure 7.10: Behaviors in approach.

7.3.1 Approach

When the player watches the ball and is the nearest in our team's players, a player's state is "Approach" (Fig. 7.10). When the ball distance is far, BhAppFF makes it approach the ball. When the ball distance is near, BhAppN makes it stop. When the player stops in front of the ball, BhShSS makes it shoot the ball. These is much the same to the attacker, but the defender chases the ball only when the ball is in our team territory. The defender does not go out of our team territory, as it always defends and walks for a good position which is between the own goal and the ball.

We explain 4 states which the attacker have as follow.

7.3.2 Support

When the player watches the ball and is not the nearest in our team's players, a player's state is "Support" (Fig. 7.11). The state is based on "Approach". It is the purpose of the state that the defender approaches and shoots the ball, whether other player approaches the ball. That has the risk that it disturbs other player, but it approaches the ball.

7.3.3 Search

When the player does not watch the ball and no player is nearer than it, a player's state is "Search" (Fig. 7.12). It is a purpose of the state that the player finds the ball speedily and efficiently. First BhSrDwn makes the player search for a near place. Next BhSrSin makes the player search for a far place. If the player does not find the ball, BhSrTuS and BhSrWk work alternately. So the player walks for a searching position and turns. It walks for a next searching position and turn until one player in our team finds the ball.

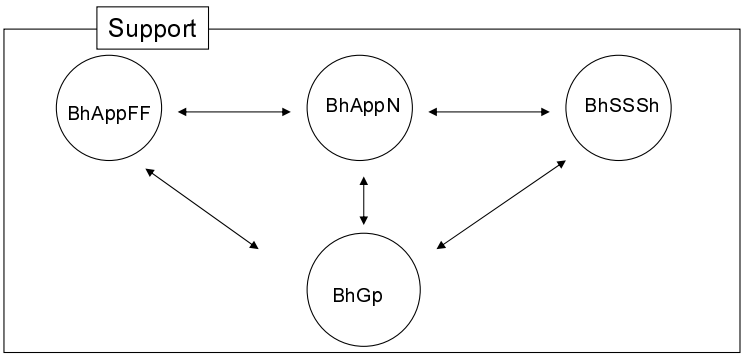


Figure 7.11: Behaviors in support.

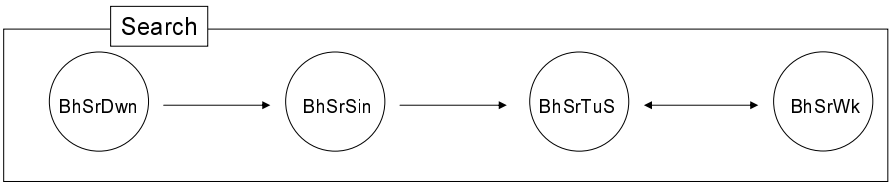


Figure 7.12: Behaviors in search.

7.3.4 Track

When the player does not watch the ball and is not the nearest in our team's players, a player's state is "Track" (Fig. 7.13). It is the purpose of the state that the player receives the ball information and finds the ball. As it does not watch the ball, it receives the ball information and searches the ball. First it localizes and turns in the direction of the ball. Second BhTrSin makes it swing its neck.

When these detailed each state and state transition are shown, it becomes the following(Fig. 7.14).

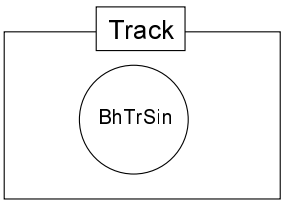


Figure 7.13: Behaviors in track.

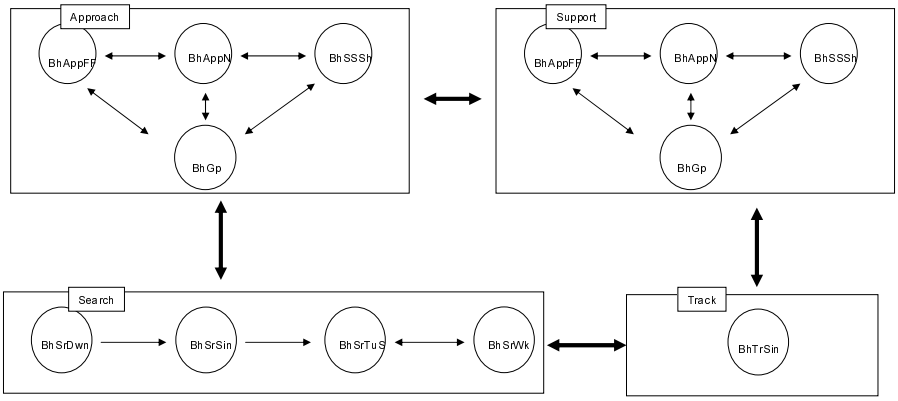


Figure 7.14: Changing states in Defender.

7.4 Goalie

Our goalie could save a number of shoot. It played a vital role in many games. The goalie spent almost all the playing time in guard. This design was quite effective to decrease shoots given up.

Our goalie have generally five states – position, search, guard, DFposit, clear. The most important the goalie is wanted is stay in front of own goal.So the most important state is position. Position is called several times as well as starting in games. When the goalie is in front of own goal, search is called first. What the goalie finds the ball which is target of the guard is directly connected to decrease shoots given up.

When goalie found the ball, it chooses one of two states. In short, the goalie should be either guard or clear. It try to make a fine judgment which is based on a distance and an angle from it to the ball.

Although the goalie is in guard, its localization may become out of order. Because, for example, it is pushed by other player or unfortunately missed the former localization. DFposit can reset its own localization with guard position. Naturally, if it judges it is wrong position, it sets itself position again.

Our goalie has many advantages and, however, has many disadvantages too. We had had serious problem till halfway. Our localization was incorrect enough for goalie because it needs more exact localization than other position. It could not have correct localization and ball position. Under this condition, it was not competitive. Moreover, we were confused by starting white line detection which supports players localization. So we needed much time for work of an improvement. But our localization had kept improving for RoboCup.

Because we did not care speed of the ball, we could not defence based on

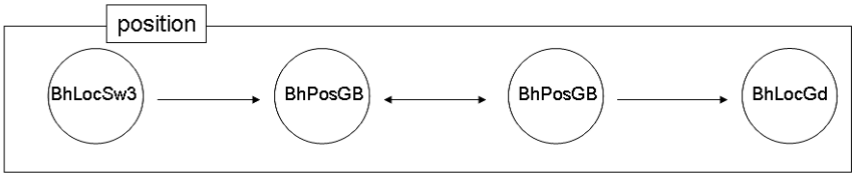


Figure 7.15: Behaviors in position.

predicting the ball. Especially it is very effective that the goalie recognizes the shoots which become outside to own goal. The goalie several became bad posture by reacting such shoots, so we have to improve that early.

In addition to these, it makes setting conditions of states not well-defined that we cannot recognize the enemies.

The next needs the localization which is correctly, is speedy and do not have to change its position.

7.4.1 Position

This state which is most important for the keeper is called first and several time when it is in other states. When the keeper is in this state, BhLocSw3 is called and the keeper checks the beacon's locations. It moves to its right position by BhPosGB with a right information which based on this information. In case it is caught by anything, we take measures to. If some time is passed without changing the goalie location, the process runs from the beginning. We use timer for this.

We explain transition to another state. If the ball is near the goalie or the own goal, it calls clear. This transition is the only transition without over position. When its move was over, its next state depends on the ball angle. If the goalie can watch ball and the ball is in thirty degree in front of it, the next is guard. In other case, the next is search. This transition is designed with why the goalie deals the situation that is in probability order to give goals.

7.4.2 Search

This state is called when the goalie is in position and able not to find a ball. If this state is transited, the goalie is in guard position and searches ball. It is because BhSrGd runs. If it can find the ball, its state is changed into guard. Or if it recognize to be in wrong position, its state is changed into position. It is designed very simple. Because if the goalie finds the ball, its state is changed into guard and decide its next action at guard in almost all situation.

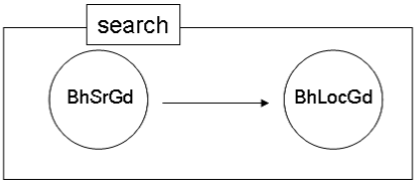


Figure 7.16: Behaviors in search.

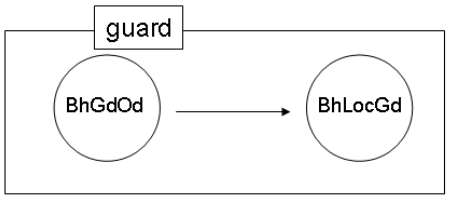


Figure 7.17: Behaviors in guard.

7.4.3 Guard

The guard is the key of state transition when the goalie is in front of own goal. Because it is able to change all other states. If the goalie can watch the ball for a long time or near the own goal, its state is changed into clear. If it is in wrong position, its state is changed into position. Or if it gets lost the ball, its state is changed into search.

And its state is changed DFposit for localization every some time in guard position. This is controlled by the timer.

7.4.4 DFposit

If the ball is far from the own goal, we cannot make shoots given up. So we designed the goalie spend this time for localization. This state is similar to the above state – position. When this state is called, it localizes with guard position. And if it had the wrong position, the position is called. What is

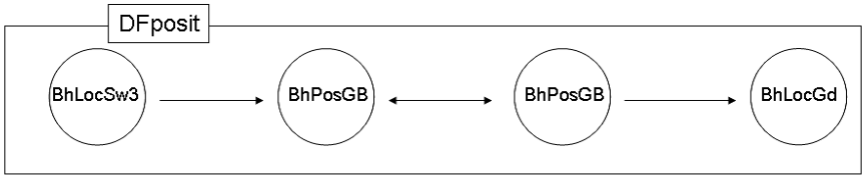


Figure 7.18: Behaviors in DFposit.

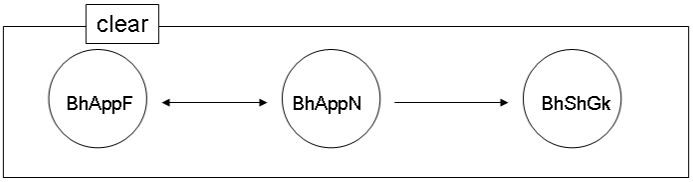


Figure 7.19: Behaviors in clear.

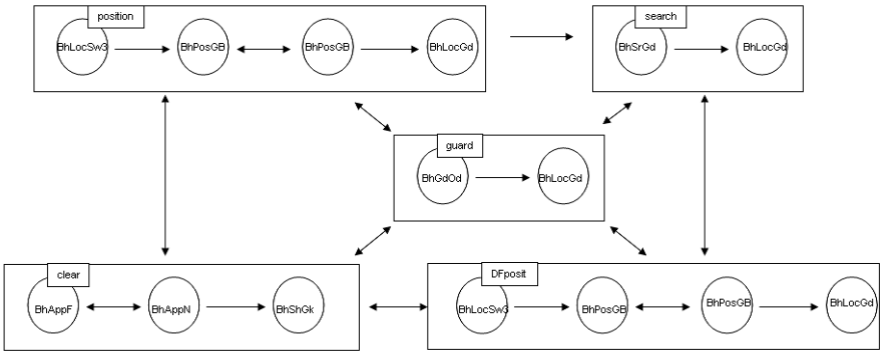


Figure 7.20: Changing states in goalie.

different from the position is positioning with guard position.(The position is positioning without guard position.) This is comes from that we estimate the goalie is almost in front of the own goal when it changes this state.

7.4.5 Clear

At almost all of the above state, the goalie does anything -positioning or localization. This is why it is very effective to decrease goals given up by long shoots to be in guard position. But the goalie has necessity to clear the ball when it is near the own goal because friend players cannot help it. Clear is the primal state to change in almost all of the situation when the ball is near the goalie or the own goal. When this state is called, the goalie approaches the ball by BhAppF which is suited for approaching from middle-far range.(But BhAppF is hardly called.) Following this, the ball distance from goalie become short, it approaches by BhAppN which is suited from near. The end of approaching, the goalie clear the ball by BhShGk.

The goalie is often far from the own goal with wrong localization after clearing the ball. So it changes state to position and readies to the following attack.

7.5 Behaviors

Since filenames must be 8 letters or less in Aperios, the filenames of behaviors is represented as an abbreviation of several words that show the acts of those.

7.5.1 BhAppFF

The behavior works when a player approaches the ball on a far distance. It is a purpose of the behavior that the player approaches speedy and accurately. The relative ball angle gives a player one thing of three actions. While the ball distance is more than $400mm$, the behavior is working and the player is doing one thing of three actions. If the ball distance is less than $400mm$, the behavior stops after the player brakes. First, action which is going ahead when the relative ball angle is less than $20°$. Second, action which is curving when the relative ball angle is more than $20°$ and less than $40°$. Last, action which is turning when the relative ball angle is more than $40°$. It explain the three actions as follow.

Going ahead is managed by the function b:far(). When a player finds the ball and stops in the front of the ball in a moment, the function in the behavior works and the relative ball angle gives the player one of the action. If the relative ball angle is less than $20°$ and the function is working, the player goes ahead as follow. The player gradually accelerates until its speed is a top speed. If the player suddenly accelerates, the slipping player makes an error in the localization. While the player is going ahead, it swings its neck and checks goal. Out team's system does not have an ability to localize in a moment, but the player checks the goal and remembers in a direction of the goal and shoots the ball in the direction. When it swings its head, it returns its head in a moment which it finds a goal. When the ball distance is less than $400mm$, the function stops and the function b:near() starts

Curving is managed by the function b:curve(). When the relative ball angle is more than $20°$ and less than $40°$, the function works. The function is next to b:far(), so a player's speed is a top speed. Walking is unstable when the player moves forward and side direction in a top speed. So when the player walks sideways and forward, it has to control its speed.

Turning is managed by the function b:turn(). When the relative ball angle is more than $40°$,the function works. While the function is working, the player stops there and turns in the direction of the ball.

These functions is working ,while the ball distance is more than $400mm$. When the ball distance is less than $400mm$, the function b:near() starts. In the function the player's speed slow down and the behavior stops.

Figure 7.21: The player's moving when Attacker is approaching.

7.5.2 BhAppN

This behavior works next to "BhAppFF" before a player shoot the ball(Fig. 7.21). In other words, BhAppN is between "BhAppFF" and "BhShSS". It is important that a player accurately and speedy approaches the ball for shooting. When the ball moved by a player goes away from the player moving in low speed, the behavior stops and "BhAppFF" starts. In this case it is that the ball distance suddenly increases that the ball goes away. If it is that the ball distance increases that the ball goes away, the player does a wrong thing. Because the ball distance which is involved in the vision and the condition is unstable. However it is most important to adjust parameter which our team can not automatically set in the behavior. That has two reasons. One is that the ball distance is involved in the vision. Second is the ball distance is involved in a parameter of "BhAppFF" when the behavior starts. If a player has a bad parameter, it has some problem that it hits the ball or slowly approaches.

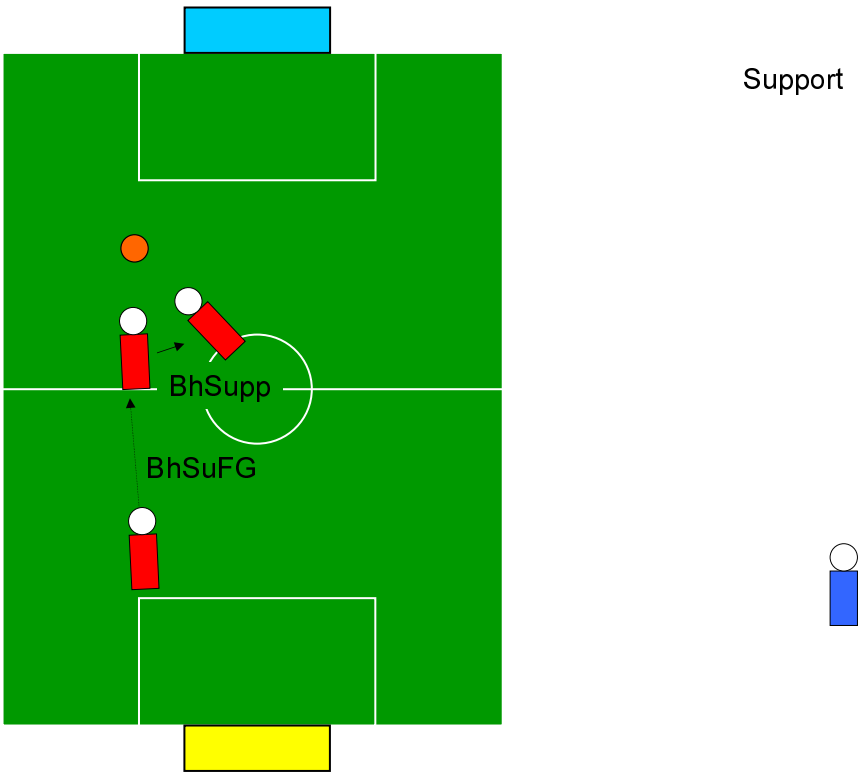


Figure 7.22: The player's moving when Attacker support.

7.5.3 BhSuFG

This behavior works when an attacker in state of "Support" approaches the ball. This script is based on BhAppFF, but it stops earlier than BhAppFF. The attacker in state of "Support" supports the other player which is near than it. So it has to stop more far than the other player and support.

7.5.4 BhSupp

This behavior works when an attacker in state of "Support" approaches ball. The attacker stops not to disturb the other player which is nearest to the ball in my team, but to support it(Fig. 7.22). There are some ways to support , but we only make the player stop. Because the information which an other player sent is not accurate, the player receiving information do not know an accurate position of the ball. If it does something to an information of the ball, it wander. But if a self localization and the information will be more accurate than now, we will be able to do various based on the information.

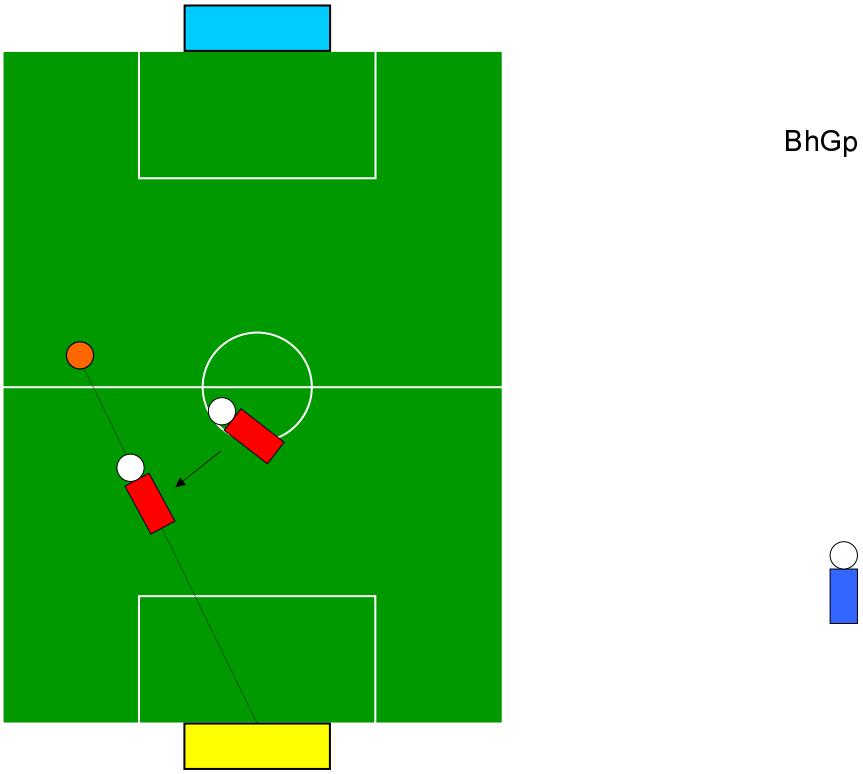


Figure 7.23: The player's moving when BhGp works.

7.5.5 BhGp

This behavior works when a defender in state of "Approach" and "Support" watches the ball which is out of our team's territory and in own goal's area. The behavior makes a defender watch the ball and move between the ball and the own goal(Fig. 7.23). Because many team's player shoot the ball to the goal, it is a good result that a defender does so. Then the defender efficiently finds the ball which is shot by the opposing player. While the defender is moving, it swings its head and checks the beacons for localizing.

7.5.6 BhSrDwn

This behavior works when players in state of "Search" do not watch the ball. A flag memorizes in a direction of the ball which a player lost sight of. First the player turns the neck in the direction memorized by the flag. If it does not find the ball, it turns the neck in another direction. The behavior makes it search for a near place.

7.5.7 BhSrSin

This behavior works next to BhSrDwn. It makes a player search the ball on a far place when the player finished searching the ball on a near place. It starts when BhSrSin finished. It makes the player's neck turn to the opposite direction.

7.5.8 BhSrTuS

This behavior works next to BhSrSin. If a player does not find the ball forward, it makes the player search backward. The neck is inclined a little in the direction where the ball disappeared. And the player turns around for searching backward. Then doing so comes to see the more distance. Moreover, there is an advantage that it becomes easy to see the ball when other players hides the ball.

7.5.9 BhSrWk

This behavior works when a player in state of "Search" finished searching around. An attacker searches the ball in opposing team's territory and a defender searches the ball in our team's territory. The behavior gives the player the searching position. The player turns in the direction of searching position. It walks for searching position(Fig. 7.24). If it only walks, its localization becomes inaccurate. Then it checks the beacons and the goals when it walks for searching position. The behavior stops, if the player arrives at the searching position. The searching position of each player is different, it is written in "JPLib/Env.lua".

7.5.10 BhTrSin

This behavior works when a player in state of "Track" searches the ball which based on the ball information sent by another player. First a player receives the ball information sent by another inaccurate localized player. And the player makes tilt2 an angle calculated from the ball information. Next it swings its head and searches the ball.

7.5.11 BhTrWk

This behavior is next to BhTrSin. A player swings its neck and walks for the position of the ball information(Fig. 7.25). The function is input an absolute position of the ball information into and makes the player walk for the position of the ball information. But it has a disadvantage that the ball information sent by the other player which has an error in localization is inaccurate.

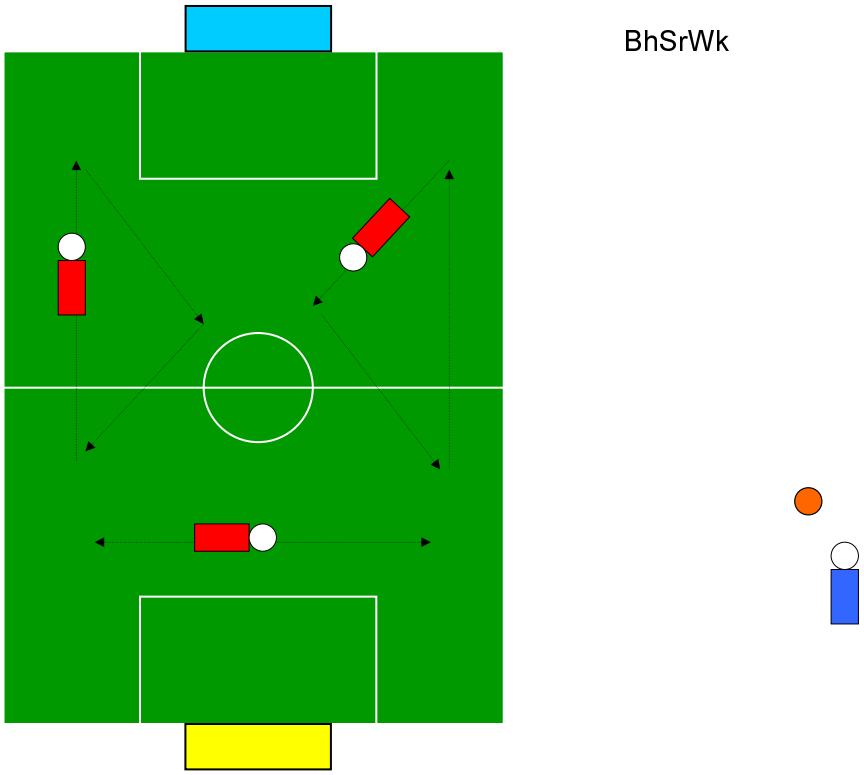


Figure 7.24: Each player's moving when they are searching the ball.

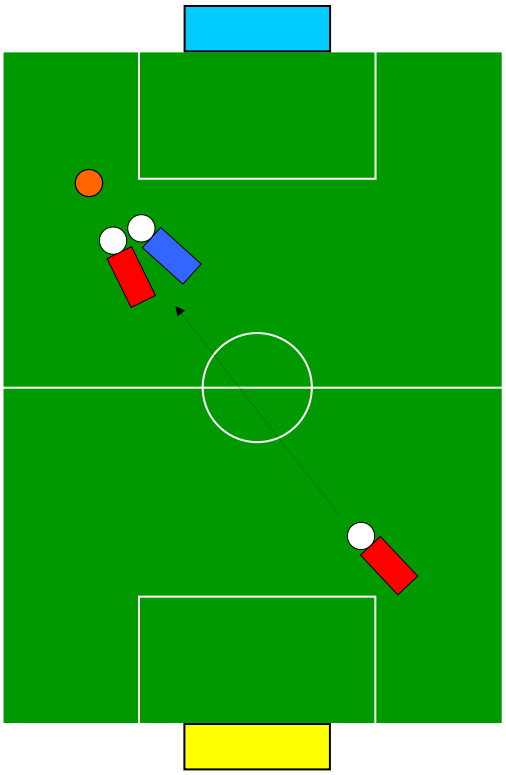


Figure 7.25: The player's moving when BhTrWk works.

7.5.12 BhLocSw3

This behavior is called when the player sets its own location. It is often called because it may set wrong location with odometer only. When it is called, the player moves its head from left or right side to the other every forty degrees. If the player could have found over two beacons at the end of this moving, it refreshes its own location. It moves its head forward, this behavior stops.

7.5.13 BhPosGB

This behavior is called when the player wants to go to its right location with facing the opposite goal. It is used as positioning for the goalie because it, in many cases, has an advantage in the point of which it can move with understanding the situation of the court. When this behavior is called, the player moves its head to three directions and tries localization. In the case it could find over two beacons, it refreshes its own location with being based on the beacons. Such its head moving is done at regular intervals. The player, in this way, often refreshes its own localization. By this localization, it turns toward the opposite goal and walks to the designated location with watching the ball. This behavior stops when the player is in the designated location with looking forward.

7.5.14 BhLocGd

This behavior was made for the goalie. It is good chances for the goalie to understand its own location because it is not moving when it is in guard position. When it is called, the player becomes the guard position. After this, it renews its own localization like BhPosLoc3.

7.5.15 BhSrGd

This behavior was made for the goalie. It is difficult for the opposite players to score when the goalie is in the guard position at fine location. So it has to find the ball and becomes in the guard position as near from the ball as it can even though it is in guard position in front of own goal.

This behavior is called when the player understands its own location. When it is called, the player becomes the guard position. After this, it moves its head from left or right side to the other.

So this behavior has an advantage that it is easy for the player to save shoots even if it has not found the ball yet.

7.5.16 BhGdOd

This behavior was made for the goalie. It is called when the player knows both its own location and the ball one. When it is called, the player keeps

moving its head toward the ball. And it walks to the side which the ball is inside of the goal area if the ball is far to the left or right from the player. It becomes the guard position at its right location.

Once this behavior starts to run, we need writing "behavior:setInitial()" expressibly to stop it.

7.6 Old Behaviors

7.6.1 BhAppIA2

This is the first behavior we made. It works when the player watches the ball, and the player approaches and turns to the ball. But in the behavior the player accelerates and stops suddenly. And when it turns, it turns more angle than the ball angle. Because the ball angle is not an angle the ball with rotation axis.

7.6.2 BhAppMCL

This behavior is based on BhApp. But it differs from BhApp in the ball distance and angle. In BhApp they are calculated only when the player watches the ball. But in this behavior they are calculated and memorized in few time. If the player does not watch the ball in a moment, it chases the memorized ball.

7.6.3 BhAppCB

This is used in AtHk7T when it is approaching. For checking self localization it swings its head right and left.

7.6.4 BhAppFG

This which is based on BhAppCB is used in At9ts. For checking self localization it swings its head right or left which is decided by the target goal's angle in Monte Carlo. A difference in swinging its head cuts the waste time and makes it stable.

7.6.5 BhAtPos

This works when the attacker does not watch the ball and receives the ball Information which is sent by the other player. It moves between the ball and the searching position(Fig. 7.26).

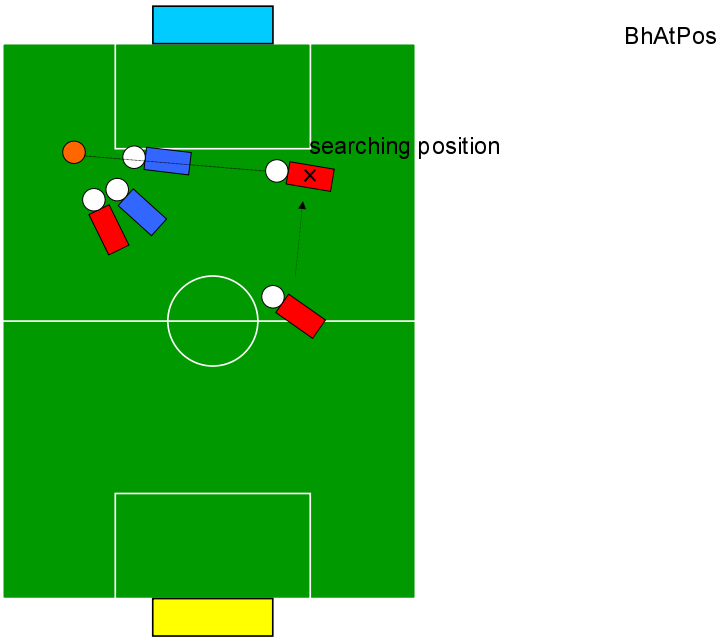


Figure 7.26: The player's moving when BhAtPos works.

Chapter 8

Shot Motions

8.1 Introduction

It requires that the all acts of kicking the ball that include 'pass' and the 'shot' will be called a shot. Excluding shots used in the special environment, the shots that has wide runs batted in a short motion, and an accurate, strong power is liked The sensor also that is the sense organ of Aibo has concentrates on all aspects of Aibo. It is assumed that the ball is in front for Aibo when the shot is developed and does the shot development. But it is difficult to develop the shot of the ball in front side. A center of gravity of Aibo is originally around little bit back about the forefoot. So, the influence power of the forefoot is stronger than that of the hind leg. The hind leg slips when both the forefoot and hind legs are extended at the same time aiming at outside of. To shoot a front ball, some devices are needed.

There are roughly separately three kinds of parts used when Aibo shoots.

- head
- legs
- body

After this, it explains the process of the shot development of each part.

8.2 The shot development tool

Special motions represented by the shot were developed with Motion Editor. Motion Editor is made by Jolly Pochie with python. This work is very simple. It decides each joint's angle and the time that is spent to change the state. A motion is made from many states. It is possible that we can order directly to Aibo by motion editor. This makes the shot development easier.

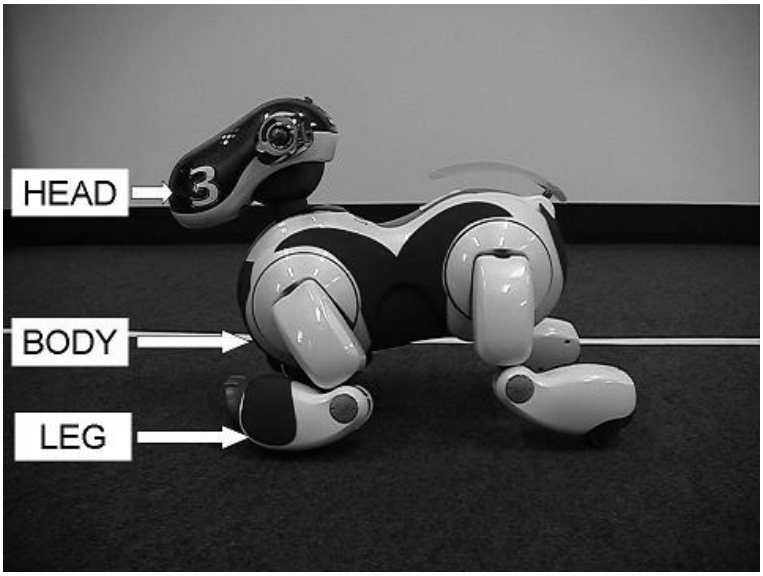


Figure 8.1: Aibo's shot parts

8.3 Development of shot with head

The head is a shot part most often used. The reason is that the ball can fly if there is a ball in the front only it bows without devise. But, that is not the shot that has wide runs batted in a short motion, and an accurate, strong power. The reason is described.

At first, it is a cause that the motion of the first joint (tilt1) of the neck is one of the slowest one in all joints. It is needless to say that the shot motion is slow because the speed at which the head is lowered is slow. The power of the shot weakens, too, when the speed is slow. When thinking about the shot that uses the head, it is necessary to devise the method of bowing besides tilt1 is moved.

Secondarily, the head of Aibo is in no so much largeness. The width is almost the same as the diameter of the ball. With this, the shot cannot be stricken if the ball is not in the presence.

Thirdly, it is in rounding of the head of Aibo. The ball doesn't run straight if the ball doesn't hit the center of the head because it draws the curved surface.

Therefore, some devices are necessary for the shot. The process of the device is written as follows.

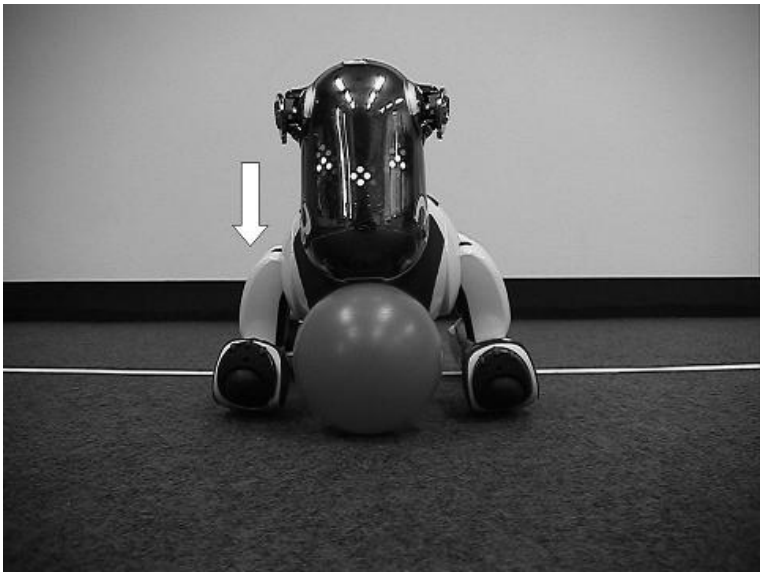


Figure 8.2: the shot with head

8.3.1 Direction of ball running

At first, it thought the ball run straight.

If the ball doesn't run in the intended direction, the ball goes out and does the own goal. When shooting the ball, the both arms are put out to previous in parallel. The direction that the ball runs to is definable by putting and the shot of the ball between arms. At this time, it is necessary to draw the jaw. Because the shot ball narrows between the mouth and the body when there is a space between the mouth and the body.

8.3.2 Power of shoot

Next, we made the power of the shot stronger. The ball shot weakly is slow, run little distance and is robbed by opponent easily.

The most general method of doing the shot of strong power is to use the weight of Aibo. The weight of Aibo is heavy with 1.6kg. We use the force when this object falls from high position. The body part of Aibo is very heavy compared with the leg and head parts. Aibo's weight concentrates on body part. Therefore, lifting the body is necessary to make the shot to use the weight of Aibo. But the weight of Aibo is heavy for Aibo too. Lifting the body by one leg is very difficult. Aibo cannot lift its body fast. It is necessary to use front both legs or hide both legs.

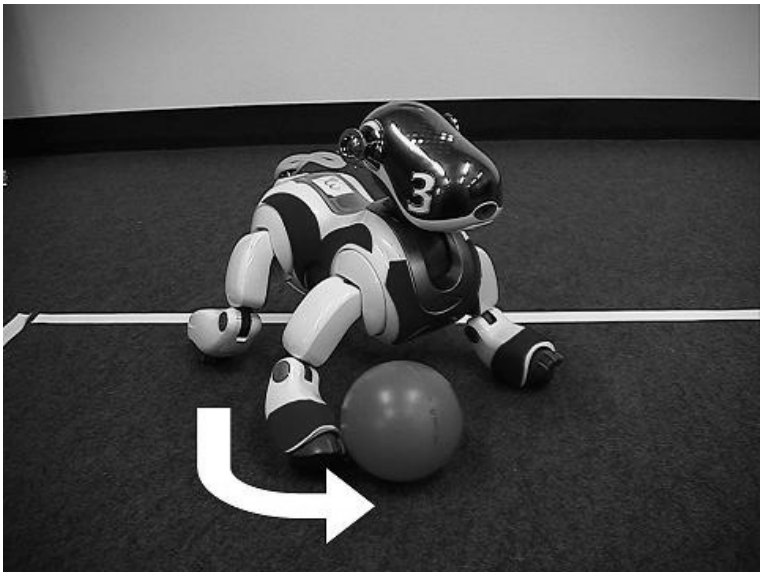


Figure 8.3: the shot with leg

8.3.3 Wide batting area

Batting area is area that Aibo can shoot and the ball run direction intended to some degree. To have wide batting area is important. When batting area is not wide, the shot of Aibo is easily failed and this shot cannot be used at the actual games. But, it is necessary to put the ball between arms so that the ball runs straight. Therefore, batting area is limited. We have the motion that catches ball before the shot. The motion that catches ball put the ball on sweet spot where Aibo can shoot the most powerful.

8.3.4 Shot test

We develop the shot that has correct direction of ball running, strong power of the shot and wide batting area. We use this shot in the actual game. The ball robbed before catch ball. When Aibo was shooting the ball, opposing Aibo enclosed it and disturbed the shot. The shot must be short motion. But, the motion that catches the ball and the motion that lift Aibo's body take much time. Another viewpoint is necessary for development shot.

8.4 Development shot with legs

The shot with legs is difficult. Because Aibo's weight concentrate on body part, it is difficult that the one leg is made free. The leg move in parallel

to ground when Aibo shoots with leg. So it is difficult that Aibo uses itself weight. Aibo's leg draws curve, the ball do not run straight. We tell how we solve these problems.

8.4.1 How to shot

We make Aibo to stand only three legs to make the one leg free. It doesn't use for the shot but the forearm is adjusted in. Width of hide legs expands a little and Aibo stabilized. We confirm Aibo does not fail down when another front leg expands as much as possible. Aibo shoots the ball to use this leg.

8.4.2 Power of shoot and wide batting area

Leg angle tilt1 is faster than head's tilt1. The ball easily runs when Aibo can shoot the ball. The leg is some length, so this batting area is wider than head one. These are advantage of the shot with leg. But to make very strong shot is difficult. When it swings the leg, it swings the head to strengthen centrifugal force and its hind legs extend to put its body out ahead.

8.4.3 Quick motion

The shot that had strong power and wide batting area would have little sphere of activity, if the shot was long motion. Quick motion is one of the most important factors. The shot with leg is separated two motions. The leg expands and the leg put out ahead. If these motions are made one motion, the shot with leg is short motion. But, this is very difficult. Aibo can not do weight shift well and can not make one leg free. When one leg expands the leg touches the ball. So when Aibo shoots the ball, Aibo shifts not only one leg but the entire body. When the one leg expands Aibo put up its body a little. It comes to be able to extend the leg sideways quickly. The leg is not so put out ahead. Batting area is narrow. But the motion of the shot is very short and after the shot Aibo can start walking easily.

8.5 Development shot with body

When Aibo shoots by body, it must put the ball under body before the shot. It is difficult. Because the body must be lifted higher than ball, the motion of the shot with body is very long. When Aibo shoots by the body, Aibo's front legs are mostly ahead of Aibo. So, the ball shot by body runs straight. The shot uses the weight of the Aibo enough. The shot is too strong rather than strong.

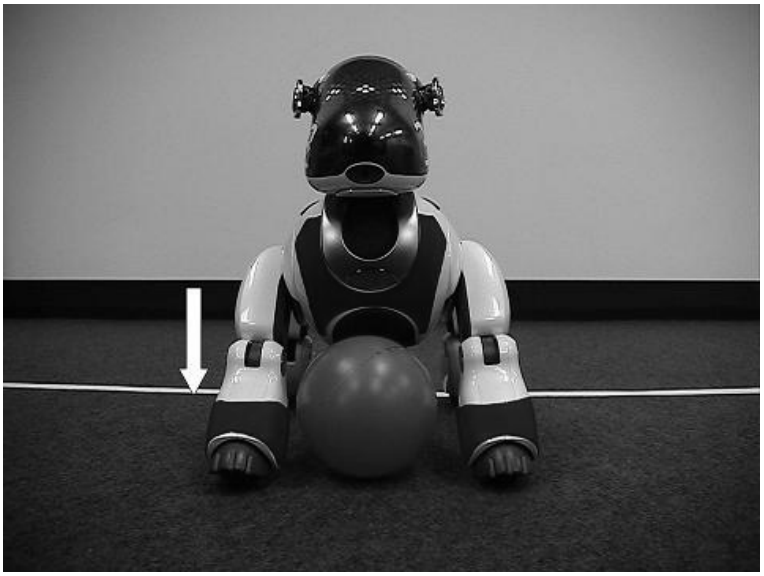


Figure 8.4: the shot with head

8.5.1 Head shot and body shot

The way of the shot with body is like to the way of the shot with head. The know-how when the shot with the head is developed can be made the best use of development of the shot with the body.

8.5.2 short motion

We examined how to lift up the body fast, but we can not find. We thought that it was a limit of the motor. When we examined, the load was put on the motor of Aibo too much and the motor was broken. The shot with the body is very strong. So the shot was not used in opposite area. We finish the development of the shot with the body.

8.6 Select the most available shot

We developed various shots here. The shot necessary in each scene is different. We have to select the shot in each scene. We use ShootSelector. ShootSelector evaluates the shots by the ball position after the shots. It is necessary to input the shot's information such as distance and direction of ball running. ShootSelector decide the position from this information. But, because the ball is not complete globe, the ball does not run straight and fast. Accurate distance and direction of ball running can not be inputted.

8.7 The result and the prospects

The influence of environment was smaller than walking and recognition of brightness but the shot was influenced. Adjustment was needed at the hall. This adjustment was done by men, so this adjustment was not extremely accurate. The shot was successful at the laboratory but it was often failed at the hall. Sometimes the shot that ShootSelector decide was not good. At the result, I know the shots had much incomplete part. Especially, that transition from walking state to the shot motion was not smooth was a big problem.

8.8 Nageppanashi German

ref:"BhGrmn"

The field was changed since RoboCup2005, so the ball can go out of the field. If using simple shoots as usual, it is difficult to keep up with the new field. In the new field, too strong shoot often put out the ball. So we need reliable shoot for the purpose carrying the ball to opposing goal.

The following sections explain structure of the shoot named "Nageppanashi German".

8.8.1 Step forward for catch

ref:"JPlib/cmotion.lua/cmotion:approachCatch"

At first, a player catches a ball.(Fig.8.5)

Then too large motion hinders smooth state transition into next motion. So we use step forward with catching.

In "Behaivior:BhAppN", a player approaches the ball until the distance to the ball becomes below a parameter that we decided. This parameter is so delicate. If a player has a bad parameter, it has some problem that it hits the ball or slowly approaches. But the best parameter to approach the ball is not so suitable to catch it. So after "Behaivior:BhAppN", a player adjusts its position to catch the ball.

8.8.2 Catch decision

After the motion to catch the ball, a player decides if the catch was success or not. If a player can see a ball, it decides false. Because if catch is success, a player cannot see the ball. When succeeding in catching the ball, a state transitions into "Nageppanashi German".

8.8.3 Nageppanashi German

ref:"JPlib/cmotion.lua/cmotion:GermanStart"

In “cmotion:GermanStart”, a player turns by an input angle, and throws out the ball.(Fig.8.6)

The motion changes according to the direction of the turn and the width of the turn. As a result, throwing the ball often failed. We build a flag by an input angle to solve it. As a result, a player chooses better motion. In “Behavior:BhGrmn”, a player calculates the angle to opposing goal position by MonteCarlo, and inputs it “cmotion:GermanStart”. But if a player notices opposing goal, it uses opposing goal position by Vision.

Figure 8.5: Step forward for catch.

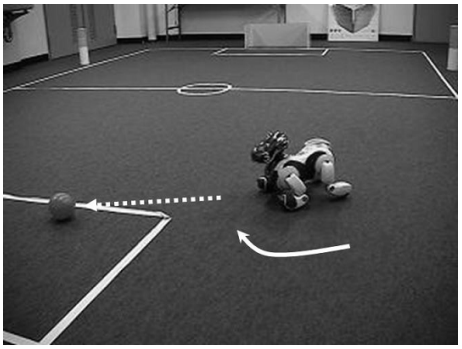


Figure 8.6: Nageppanashi German.

8.8.4 Origin of name

“Nageppanashi German” was named after the following sentences.

“Nageppanashi” means “throw out” in English. This dynamic shoot is associated with a Japanese professional wrestling technique “Nageppanashi German Suplex”. We are inspired by German Team to develop this shoot. So this shoot was named “Nageppanashi German” in respect for German Team.

8.9 Abe shoot

ref:”BhShAb”

This shoot is a strong and accurate shoot to the front side. It is conditioned on catching the ball.

The following sections explain structure of the shoot named “Abe shoot”.

8.9.1 Step forward for catch

At first, a player catches the ball the same as in Sec8.8.1. So a player adjusts its position to catch the ball the same as in Sec8.8.1.

8.9.2 Catch decision

A player decides about catch the same as in Sec8.8.2. When succeeding in catching the ball, a state transitions into "cmotion:catchTurn" (Fig. 8.7). This turn is a part in the first half of Abe shoot.

8.9.3 Abe shoot

In "BhShAb", a player turns by an input angle, and shoot to the front side.(Fig.8.8)

If it is not so accurate, a player often puts out the ball. In "Behaivior:BhShAb", a player calculates the angle to opposing goal position by MonteCarlo, and inputs it to "cmotion:catchTurn". But if a player notices opposing goal, it uses opposing goal position by Vision.

After the turn, a player calculates the angle again by way of precaution. If the error margin is too large, a player turns again to adjust position.

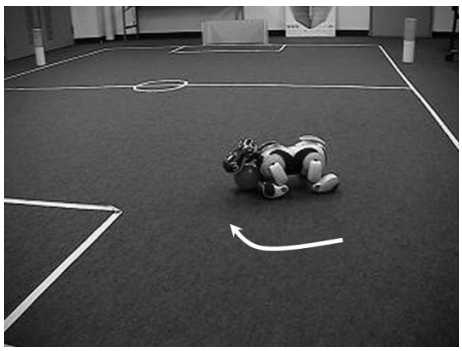


Figure 8.7: Turn with ball.

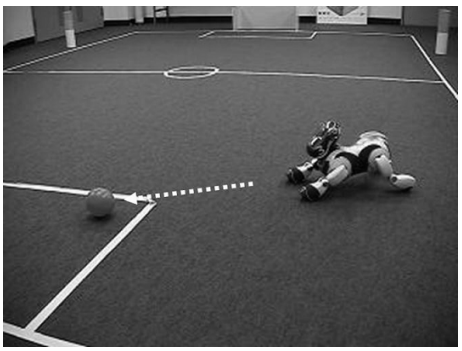


Figure 8.8: Abe shoot.

8.9.4 Origin of name

This shoot was produced in RoboCup 2005 league.

"Abe shoot" was named after shoot producer's name.

Chapter 9

Bots

9.1 Players

Since strategies for soccer are described in Lua scripts, we can use the same bot in attacker, defender, and goalie. As far as goalie concerned, we used a localization module using line information in penalty area, because we had not developed a localization module using all line information at first. We could develop many modules, and many bots during the competition, because an old module can be easily replaced with a new module to make a new bot in our framework. We have developed new modules according to problems occurred during the soccer games in the RoboCup competition, as follows.

BasicPlayer7
: The bot *BasicPlayer7* has developed before the competition. Fig. 9.1 shows the dependence of every module (except for debug modules) in the bot.

BasicPlayer7cc
: The bot *BasicPlayer7cc* is a version using the vision module *CDTBoxTable5JPM* that uses camera calibration method on trial, instead of the module *CDTBoxTable3JPM*.

BasicPlayer8
: The bot *BasicPlayer8* is a version using the vision module *CDTBoxTable6JPM* that speeds up camera calibration, instead of the module *CDTBoxTable5JPM*.

BasicPlayer8strict
: The bot *BasicPlayer8strict* is a version using the ball recognition module *DetectBall9JPM* that reduces recognition errors, instead of the module *DetectBall8JPM*.

Figure 9.1: The module dependence of the bot *BasicPlayer7*.

BasicPlayer8mcl3

The bot *BasicPlayer8mcl3* is a version using the self localization module *SelfMCL3JPM* with line information instead of the module *SelfMCL2JPM*.

9.2 Capturing Images

We used tha bot *ImageCapture7* to take pictures with AIBO's camera. Using this bot, we could take some pictures in various positions that we want to. We also used the same bot as the bot used for players, so that we could get the images that AIBO was seeing in the game. Using the images captured by this bot, we could more exactly make the color table, and debug for vision modules.

Chapter 10

Tools

10.1 Motion Editor

When we create static motion (shoot, pass, guard, ...etc), we use our original motion editor (Figure 10.1). The basic usage is so simple; input angles for each joints, and repeat this for a number of frames. Details of this tool are mentioned below.

10.1.1 Main Window

This tool has two window; main window (has grid area) and control window (has many buttons). Parameters for each joint angles are input in the grid area. It has 16 rows: 15 rows of them are for joint angles and 1 row for time interval. The number of columns can be configured by yourself as a command line argument (the default is 32). Columns are for frames of motion, like animation cells.

Many numbers in this grid, represents robot's joint angles or time interval, are may be mathematical scheme of python. For example, `30`, `30*2`, `(10+15)*sin(pi/6)`, or `last/2` are allowed. A special parameter `last` means the last value, so if a joint angle was `30`, then `last/2` in the next column will be `15`.

The cells for time interval means a number of steps. For time interval `n`, robot plays this motion spending about `n*8` [ms]. In fact, the value smaller than about `10` has no accuracy.

10.1.2 Control Window

The control window has 10 buttons, 1 spin box, and 1 text box. Their usage are below.

- Send Angles
  Send a selected column to the connected robot. This button will be used when you check joint angles.

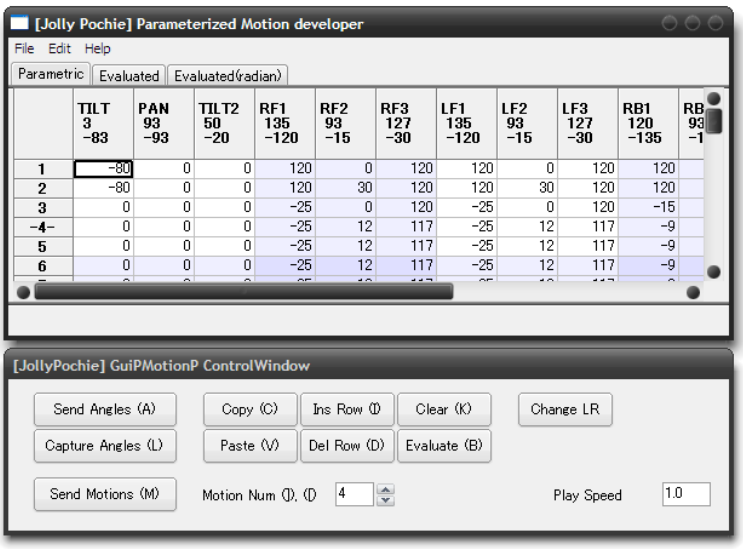


Figure 10.1: Motion Editor

- Capture Angles
  Capture joint angles of the connected robot. When angles were captured and received, parameters on the selected column will change to received values.

- Send Motions
  Send selected columns (motion) to the connected robot. You must select columns or set "Motion Num" when use this button.

- Copy
  Copy selected cells. This is useful when you create a motion which has repetitive motions.

- Paste
  Paste the values to selected cells. This will be used with "Copy" button.

- Ins Row
  Insert new column to the current position. (Attention to confusing name "Ins Row".)

- Del Row
  Delete current column.

- Clear
  Clear selected cells and set default values.

- Evaluate
  Evaluate selected cells if they were written by mathematical scheme. Note that you must evaluate all cells on the grid when save the motion and copy the file to the memory stick.
- Change LR
  Shift left and right on selected cells. Left shoot will be right shoot.
- Motion Num
  Motion number which you want to send to the connected robot. For "Motion Num" n, column 1 to column n will be send as a motion.
- Play Speed
  Play speed of motion. "Play Speed" 2 lets motion playing time half.

10.2 Camera Calibration

The ERS-7 camera has a serious vignetting effect (radiometric distortion), which makes the image appear dyed in blue at the corners, as is pointed out by GermanTeam [2] and others. Since our object recognition is based on color classification, the blue cast on the corners of the images disturbs correct recognition. By reading the technical report 2004 of GermanTeam [2], we decided not to correct the geometrical distortions of the images due to lens effects, because it does not seem to pay the computational cost. However, the color distortions cannot be overlooked for our object recognition, so that we decided to correct it.

A general goal of color corrector is to find a good function

$$(Y', U', V') = F(x, y, Y, U, V),$$

where (Y, U, V) is an original color of a pixel at (x, y), and (Y', U', V') is the corrected color. If there is no color distortion, $Y' = Y$, $U' = U$, and $V' = V$ for any x and y. Since the image captured by ERS-7 is apparently distorted, we have to find a good approximation of F from various observed data. Moreover, F has to be computed efficiently. We dealt with this problem as follows. First we simplify the problem by assuming that Y' (U' and V', resp.) can be determined by the position (x, y) and Y (U and V, resp.). That is, we will find three functions F_Y, F_U, and F_V

$$\begin{aligned} Y' &= F_Y(x, y, Y), \\ U' &= F_U(x, y, U), \\ V' &= F_V(x, y, V), \end{aligned}$$

instead of finding F. Moreover, since the color distortion seems to be radiometric, we also assume that Y' can be computed by

$$Y' = F_Y(r, Y),$$

where r is the distance between the position (x, y) and the center (x_c, y_c), that is $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$. Since the computations of U' and V' can be treated in the same way, we will only mention about Y' in the sequel. We tried the following two methods.

10.2.1 Color Correction based on White Color Only

First we tried a simple method, where we assumed that the distortion of Y' depends on the position (x, y) *but not* Y *value.* That is, assuming that $F_Y(r, Y)$ can be represented by

$$F_Y(r, Y) = f(r) * Y.$$

The function $f(r)$ is experimentally determined using the image of white wall captured by ERS-7 as follows. Let $Y_{x,y}$ be the intensity of Y channel at pixel (x, y). Let C be the average intensity of the pixels $\{Y_{x,y} \mid \sqrt{(x - x_c)^2 + (y - y_c)^2} \leq 10\}$, and C_r be the average intensity of the pixels $\{Y_{x,y} \mid \sqrt{(x - x_c)^2 + (y - y_c)^2} = r\}$. Then

$$f(r) = \begin{cases} 1.0 & (\text{if } 0 \leq r \leq 10) \\ C/C_r & (\text{if } 10 < r). \end{cases}$$

Unfortunately, it turned out that the method is not very effective to correct all colors. It means that a single color is not enough to determine the color corrector.

10.2.2 Color Correction based on Various Colors

In order to get more accurate color correcting function $F_Y(r, Y)$, we have to use various colors. However, it is not so easy to prepare various color sheets, and it is quite time consuming task to capture the images of all these sheets by hand. Thus, we utilized a display of PC (Fig. 10.3). We wrote a simple script, which shows various uniform colors on the display one after another, and sends ERS-7 a command to capture the image. Then ERS-7 sends back the images to PC, and these images are saved into the disk. By analyzing these images, we get the curves of the distortion for various intensities. Fig. 10.2 shows some of these curves, although we used a huge number of colors. By interpolating these curves continuously, we have the surface $G(Y, r)$. Then we get the color correcting function $F(Y, r)$, as a reverse of $G(Y, r)$, that is $F(Y, r) = Y'$ with $G(Y', r) = Y$. We represent the function $F(Y, r)$ as a lookup table.

Fig. 10.4 shows a result of the color corrector.

10.3 Simulator for Robot Scripts

Using a real robot for testing a script takes a lot of trouble. A real robot will break down easily. A real robot will run out of juice within a hour. A

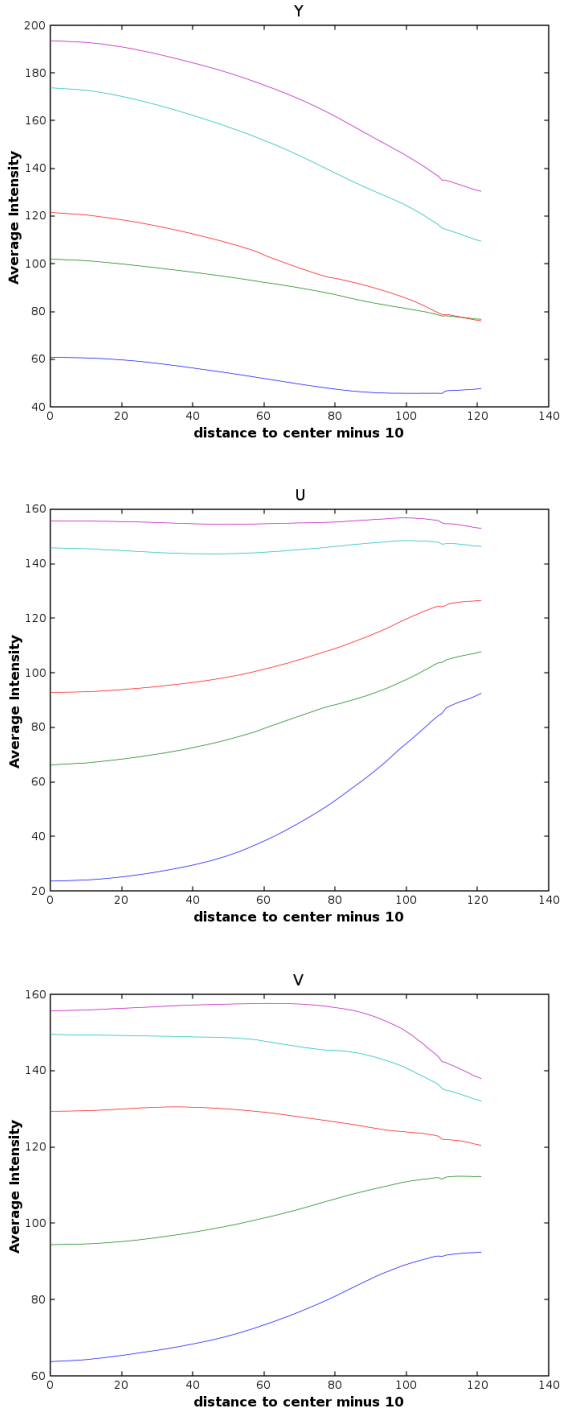


Figure 10.2: Distortions of Y, U, and V planes. The horizontal axis represents the distance to the center (minus 10), and the vertical axis represents the average intensity.

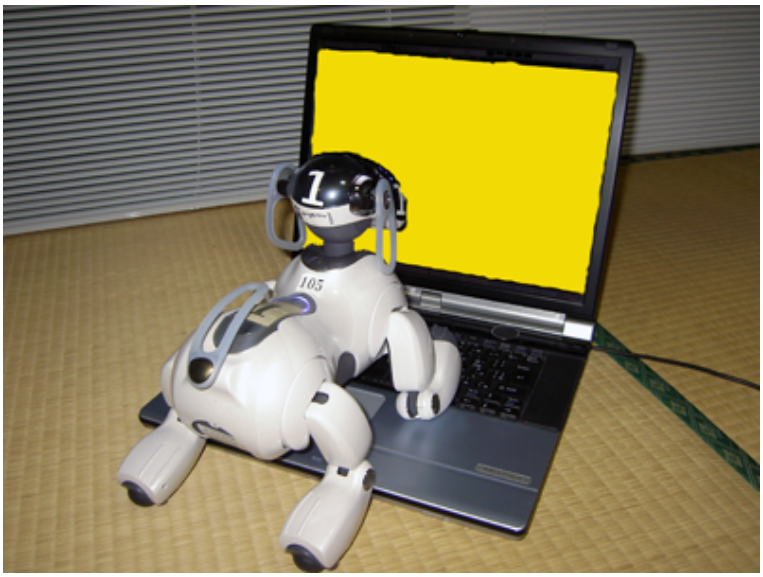


Figure 10.3: Automatic camera calibration process. Various uniform colors are shown in the display of PC one after another, and the captured images by ERS-7 are saved into the disk of PC via TCP connection.

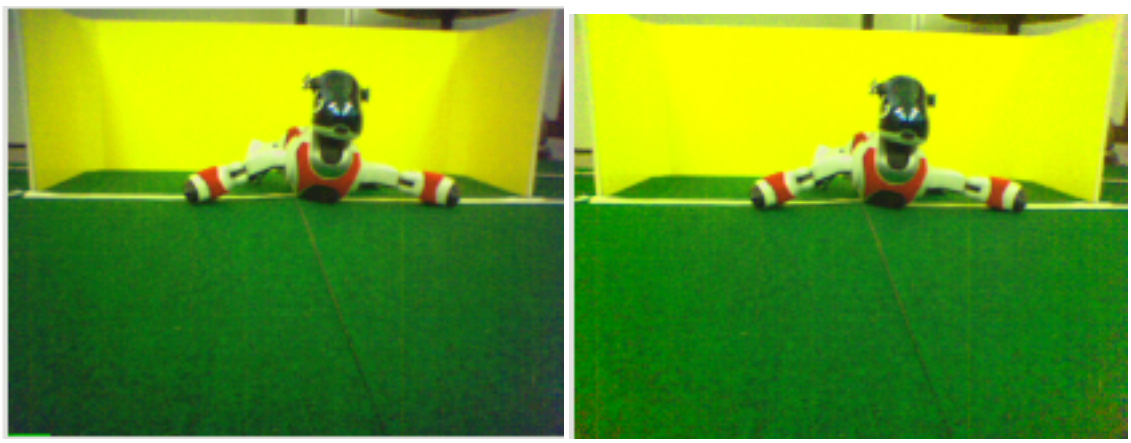


Figure 10.4: Original image (left) and corrected image (right).

Figure 10.5: Execution of our simulator

real robot never makes the same behavior twice in the real world. A real robot provides debugging information via only wireless LAN. Therefore, the need of a simulator (fig. 10.5) that executes a robot script presents itself to improve the development.

Fig. 10.6 shows the outline of the implementation of the simulator. the left side shows the real world including a robot program, and the right side shows the virtual world in the simulator. For easier comprehension, we divide the real world into three components: an environment, active modules, and passive modules. We can make the simulator by simulating or emulating these components. This will enable us to run the same script in both real world and virtual world.

A pseudo environment is the component that simulates the real environment, including a soccer field, a ball, and robots in a real world. Since these objects are represented as 3D models, we can visually check whether a virtual robot can see a ball or not in the simulator. We implement these objects in VPython [8], which is a 3D graphics module for python. Fig. 10.7 shows a model of AIBO in our simulator. It is a very simple model, which has only a body, a neck, and a head, but this way adequately realizes that robots ideally work in the simulator, as if the robots have omni-directional wheels, and an accurate vision system.

Pseudo active modules are the components that emulate active modules, which are modules that need to call lua functions. An active module has a function called at regular intervals or at the time an event happens, that is, a special function shown in the previous chapter. Specifically, pseudo active

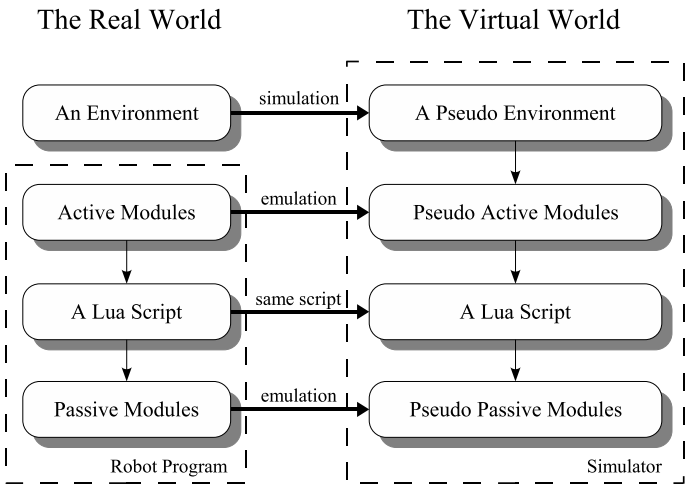


Figure 10.6: The outline of the implementation of the simulator

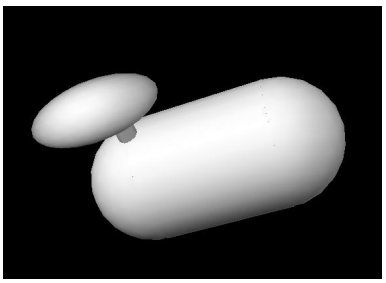


Figure 10.7: The pearly white very simple model of AIBO.

modules consist of a mind module, an action module, an UDP module.

Pseudo passive modules are the components that emulate passive modules, which are modules that will be called in lua functions. Specifically, passive modules consist of a locomotion module, a shoot module, a self localization module, a ball localization module, and so on. Since the simulator know the accurate global positions of a ball and robots, and the distance of one step that was measured by a ceiling camera, we do not need to calculate them in the simulator. In addition, the simulator do not have to change LED colors, play sounds, and so on. Therefore, we only have to define simple dummy functions on Lua side as follows.

```
soundPlayer = {}

function soundPlayer:playSoundOnce()
end

function soundPlayer:playSoundStop()
end
```

In conclusion, the simulator increased the development efficiency dramatically, because we could develop many robot scripts all over the place (e.g. in our house, in airplanes, and in any hotel) without real robots. As a matter of course, since robots in the virtual world are in some degree different from them in the real world, it is necessary to test these script in the real world at the end. Nevertheless, the simulator will certainly reduce the heavy burden to test a robot script with a real robot. Moreover, since a real robot can communicate with a virtual robot via UDP, we can establish a multi-agent environment in the real world by using only one real robot. This makes it easy to practice team play, which needs more than two robots.

10.4 Position Visualizer

A position visualizer (fig. 10.8) is a tool visualizing various positions of a robot, a ball, beacons, and so on. Using the position visualizer, we can check the accuracy of a self localization module, a ball localization module, a ball recognition module, and a beacon recognition module. If a script does not work well, we can find where the reason is, especially in the modules. The position visualizer acts, in a manner, as an intermediary between the simulator and the real world.

The position visualizer was developed based on the simulator. The following is an illustration about objects shown in the position visualizer. Each object in the position visualizer may be the same position as that in the real world, if we can develop a robot program ideally.

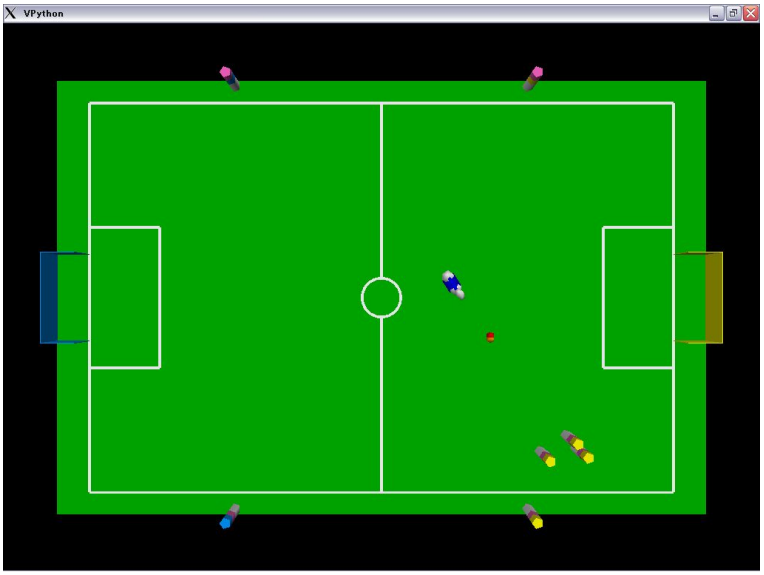


Figure 10.8: Position Visualizer

- An aibo model shows the position of a robot calculated by a self localization module.
- An orange ball model shows the position of a ball calculated by a ball recognition module.
- A red ball model shows the position of a ball calculated by a ball localization module.
- Redundant beacon models show the histories of the positions of beacons calculated by a beacon recognition module.

The positions of the objects are automatically updated at regular time intervals. The position visualizer can ask the modules about position information using a remote control module *RCLuaScript*, and update the positions using the answers. The module *RCLuaScript* allows us to execute any Lua script code via TCP. The following is Lua functions, which make some values a character string, used in the position visualizer.

`mcLocalization:toString()`
: This function returns a character string made by joining the positions x, y, and t of a robot.

`body:headToString()`
: This function returns a character string made by joining the joint angles *tilt1*, *pan*, and *tilt2* of the head of a robot.

`detectStatus:ballToString()`
: This function returns a character string made by joining the distance *dist*, and the angle *pan* of a ball.

`ballMCL:toString()`
: This function returns a character string made by joining the distance *dist*, and the angle *pan* of a ball calculated using Monte Carlo method.

`detectStatus:historyToString()`
: This function returns a character string made by joining the histories of the positions of beacons.

In order to receive the returned values, We have to send a script code as follows. The module *RCLuaScript* returns a value assigned in *"retRC"*.

```
retRC = mcLocalization:toString()
```

Chapter 11

Technical Challenges

11.1 The Open Challenge: AiboLingual

AiboLingual is the program which was made for Open Challenge. We developed it obtaining hint from "Bowlingual" which translates barks.
But game hall is so noisy that transfering data to AIBO by the sound is difficult, thus this program communicates with ear, face or back LED. Concretely, it transfers data between AIBO's LED and USB camera connected with PC. Then data is decoded on the PC and the character is displayed. Applied plan to the soccer is as follows.

- Means of communication between AIBO without wireless LAN
- Output of AIBO's state by means of LED under situation in which wireless LAN cannot use

We cannot achieve them, because its transfer rate is slow and it cannot correct error. So our next challenge is to solve these problems.

11.1.1 Bot

We use the bot which flashes LED for AiboLingual, but such bot is included in Atacker's one. So we used it as is.

11.1.2 Input Device

Input device is "Logicool QcamPro4000".

This camera's interface is USB1.1, so transfer speed is not fast. Thus, this part became problem when data transfers. It was necessary to use the device transfer rate is fast.

11.1.3 Transmission program

AIBO has ear, face and back LED. But face and back LED cannot recognize well, because a range they flash is narrow and the number of colors is few. So we decided to use ear LED.
Ear LED can blink easily by using Lua script. From this, we decide to transfer data by flashing AIBO's ear LED continuously. Colors we use are yellow, blue, green, red, and purple. These colors are the most recognizable. Among these colors, purple uses as control signal.
Next, we explain a concrete method. This method is so simple. First, each character of sentence that we want to transfer changes ASCII code which is shown by decimal number. Second, this code changes quinary number. Finally, this quinary number represents by four colors (yellow, blue, green, red, and purple), and it makes ear LED flash. Purple is the pause of each character.

11.1.4 Reception Program

Color adjustment

We make color settings by deciding each color's HSV range. It is easy to adjust color recognition, but it is weak to the change of environment. If it changes even a little, it cannot recognize ear LED's color. To solve this problem, we reduce camera's brightness extremely. By doing so, it copes well with a change of environment.

Now we are using another color adjustment system. We want to replace this method with present, and to recognize color automatically in future.

Reception method

Next, we explain a reception method of data. Fundamentally, it makes a same thing as transmission program in reverse order. It recognizes ear LED's color continuously. It changes quinary number into alphabet when purple is inputed and shows character on the display.
This method cannot get up speed of transfer well. In future, we want to make more efficient method for solving this problem.

11.2 The Variable Lighting Challenge

We can not use only one color table, because a lighting condition changes with times in the Variable Lighting Challenge. So, we considered that some color tables were changed with the time. But, times of changing the color tables is long. In this year, we changed only a camera calibration. we knew timings which the lighting condition change. So we changed the calibration

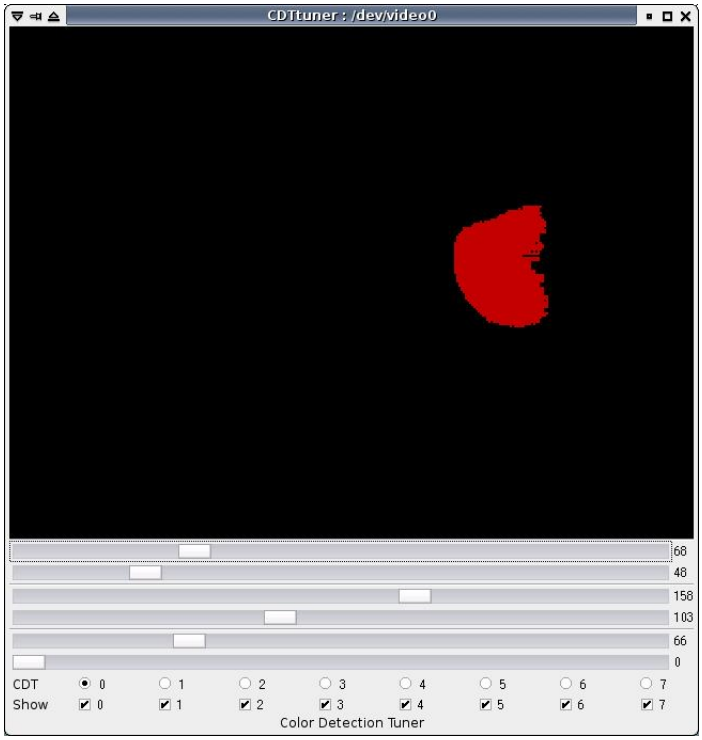


Figure 11.1: Adjustment Window

with the timing. When the calibration is changed, the Aibo looks own front arm. The form is fixed when the Aibo looks own front arm. So the Aibo can get the same condition pictures which have white of own arm and green of the field on anywhere. Hence, we can change the calibration against the changing lighting condition.

11.3 The almost SLAM Challenge

This year, we did not achieve a satisfactory result in SLAM Challenge because of the lack of development time. In this section, we describe only parts of our approach that have been implemented.

11.3.1 Additional Landmarks Remembrance

The same self localization module as an actual game could not be used in SLAM Challenge. The robot needed to remember additional landmarks described in the rule book. There were at least three landmarks containing a patch of pink at least 10 cm across, and they were outside the playing are but on the green field. Therefore, at first, we developed the module that

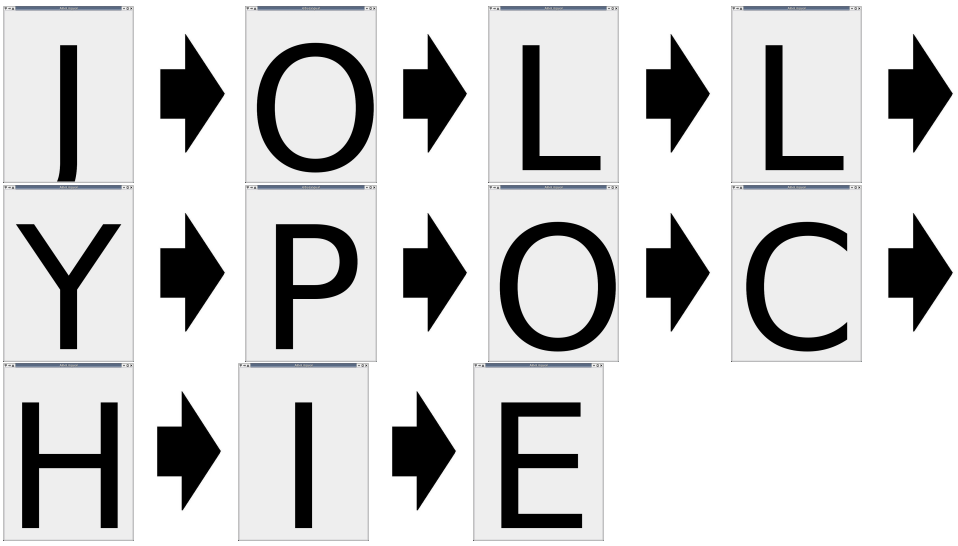


Figure 11.2: Output Window

remembered those landmarks for the first stage of the SLAM Challenge. We divided each edge on the field into twenty parts. We calculated where landmarks are observed, and incremented the number of counters corresponding to the parts while the robot went round with observation, because the result of observations by our vision system had various errors, and those landmarks were at least 15 cm apart. For each part, if the number of the counter of a part exceeded the average number of counters, it was decided that additional landmark exist that part. The parts in which beacons had already included were excluded.

In Fig 11.3, red cross show the parts in which beacons have already included, pink circles present that additional landmarks exist in the parts.

Secondly, we developed the self localization module that estimated the position according to those information. However, we did not have enough time for testing that module, and did not check a performance of it. The result was that we did not achieve a satisfactory result.

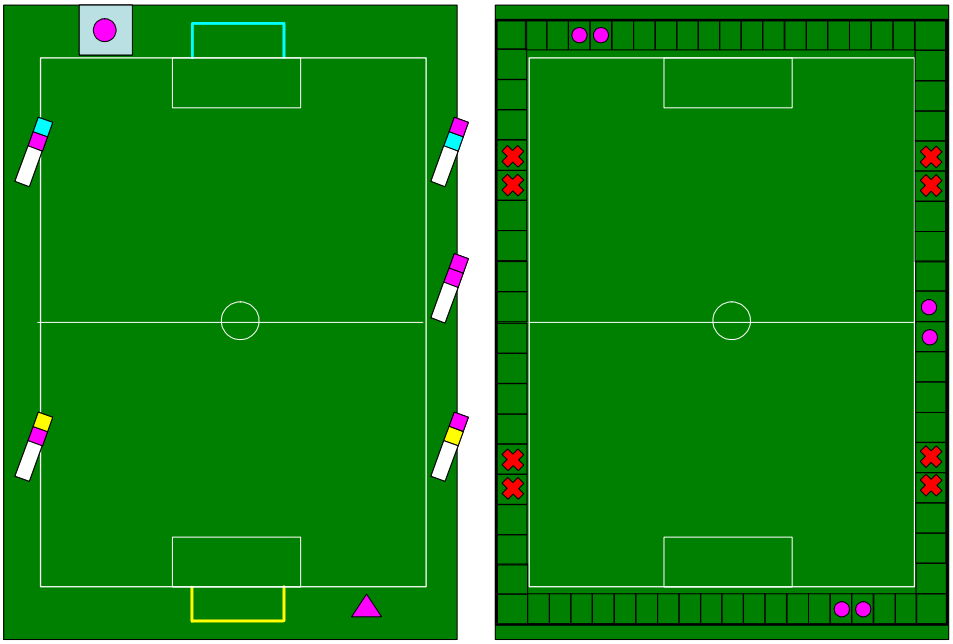


Figure 11.3: The result of line recognition

Chapter 12

Conclusion

The team Jolly Pochie has participated in the RoboCup Four-Legged League for three years. This year was a first participation as the united team of Kyushu University and Tohoku University. The collaborative innovation with two universities made us advance to the quarter finals.

We have improved many things in our system. Especially, embedding script language Lua into our system had a large improvement in speed of our development, and applying genetic algorithm for tuning locomotions found good parameters.

We will continue the development of the soccer simulator and combine it with machine learning techniques.

Bibliography

[1] JollyPochie —team for RoboCup soccer 4-legged robot league—. <http://www.i.kyushu-u.ac.jp/JollyPochie/>.

[2] Thomas Röfer et. al. GermanTeam roboCup 2004. Technical report, 2004.

[3] Ben Fry and Casey Reas. Processing. <http://processing.org/>.

[4] Stefan J. Johansson and Alessandro Saffiotti. Using the Electric Field Approach in the RoboCup Domain. In *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *LNAI*, pages 399–404. Springer-Verlag, 2002.

[5] Luabind. http://luabind.sourceforge.net/.

[6] TeamChaos. http://www.aass.oru.se/Agora/RoboCup/.

[7] The Programming Language Lua. http://www.lua.org/.

[8] VPython. http://vpython.org/.