

```

#define WATCH_DEBUG_MODE
#include "action.hh"
#include "trig.h"
#include "assert.h"

Command_obj::Command_obj(int type, double a, double b) {
    command_type = type;
    arg_a = a;
    arg_b = b;
    arg_text[0] = '\0';
    next = prev = NULL;
    action = NULL;
    sent = 0;
}

Command_obj::Command_obj (Action_obj *embedded_action) {
    command_type = ACTION_COMMAND;
    action = embedded_action;
    next = prev = NULL;
    sent = 0;
}

Command_obj::Command_obj(int type, char* msg) {
    command_type = type;
    arg_a = 0;
    arg_b = 0;
    strcpy(arg_text, msg);
} else {
    //! what should we do here? somebody really screwed up.
    assert(0);
    command_type = COMMAND_ERROR;
    arg_a = 0;
    arg_b = 0;
    arg_text[0] = '\0';
}
action = NULL;
next = prev = NULL;
sent = 0;
}

Command_obj::~Command_obj() {
    Action_obj *temp;
    temp = action;
    if (temp != NULL) {
        action = NULL;
        delete temp; /* could be trouble if some command in this action is being
        executed in the signal handler when we delete this */
    }
}

Command_obj::Command_obj () {
    next = prev = NULL;
    action = NULL;
    sent = 0;
}

Command_obj* Command_obj::send_command(World *world) {
    Command_obj *retval = this;
    switch( command_type ) {
        case MOVE_COMMAND :
            world->me->command_move(world, arg_a, arg_b);
            sent = 1;
            break;
        case TURN_COMMAND :
            world->me->command_turn(world, arg_a);
            sent = 1;
            break;

```

```

        case KICK_COMMAND :
            world->me->command_kick(world, arg_a, arg_b);
            sent = 1;
            break;
        case DASH_COMMAND :
            world->me->command_dash(world, arg_a);
            sent = 1;
            break;
        case SAY_COMMAND :
            world->me->command_say(world, arg_text);
            sent = 1;
            break;
        case SENSE_BODY_COMMAND :
            world->me->command_sense_body(world);
            sent = 1;
            break;
        case CHANGE_VIEW_COMMAND :
            world->me->command_change_view(world, arg_a, arg_b);
            sent = 1;
            if ((world->me->view_quality != arg_b) || (world->me->view_width != arg_a))
                assert(0);
            break;
        case ACTION_COMMAND :
            assert (action != NULL);
            retval = action->send_next_command();
            if (retval != NULL)
                retval = this; /*don't return a command which is marked as sent, so that the c
            ommand list
            //sent = 0; don't set sent, let update take care of it
            break;
        case HALT_COMMAND :
            //same as empty command, but we don't move past it
            retval = this;
            //sent = 0
            break;
        case EMPTY_COMMAND :
            #ifdef DEBUG_MODE
            world->me->command_dash (world, 0);
            #endif
            //nothing to do
            sent = 1;
            break;
        default:
            //bad code! bad comma bad comma bad bad bad
            assert (0);
            return NULL;
        }
        return retval;
    }

    void Command_obj::fprintf_command(FILE *f) {
        switch( command_type ) {
            case MOVE_COMMAND :
                fprintf(f, "Move to (%.3lf, %.3lf)", arg_a, arg_b);
                break;
            case TURN_COMMAND :
                fprintf(f, "Turn %d degrees", (int)arg_a);
                break;
            case KICK_COMMAND :
                fprintf(f, "Kick with power %d at %d degrees", (int)arg_a, (int)arg_b);
                break;
            case DASH_COMMAND :
                fprintf(f, "Dash with power %d", (int) arg_a);
                break;
            case SAY_COMMAND :
                fprintf(f, "Say %s", arg_text);
                break;
            case SENSE_BODY_COMMAND :
                fprintf(f, "Send sense_body request");
                break;
            case HALT_COMMAND:
                fprintf(f, "Halt / Null command");

```

```

break;
case EMPTY_COMMAND :
    fprintf(f, "Wait / Empty command");
    break;
case ACTION_COMMAND :
    fprintf(f, "Embedded action:\n");
    action->fprintf_action (f);
    break;
case CHANGE_VIEW_COMMAND :
    fprintf(f, "Change view to ");
    switch ((int)arg_a) {
    case VIEW_WIDTH_NARROW :
        fprintf(f, "narrow ");
        break;
    case VIEW_WIDTH_NORMAL :
        fprintf(f, "normal ");
        break;
    case VIEW_WIDTH_WIDE :
        fprintf(f, "wide ");
        break;
    } switch ((int)arg_b) {
    case VIEW_QUALITY_HIGH :
        fprintf(f, "high");
        break;
    case VIEW_QUALITY_LOW :
        fprintf(f, "low");
        break;
    }
    default :
        return;
    }
}

Command_list :: Command_list (World *w, Command_obj *cmd) {
    unsend_command_count = command_count = 1;
    top = last = nextup = cmd;
}

Command_list :: Command_list () {
    command_count = 0;
    unsend_command_count = 0;
}

Command_list :: ~Command_list () {
    empty();
}

void Command_list::empty () {
    Command_obj *temp;
    while (top != NULL) {
        temp = top;
        top = top->next;
        delete temp;
    }
    command_count = unsend_command_count = 0;
    top = nextup = last = NULL;
}

Command_obj *Command_list::append_command (World *w, Command_obj *cmd) {
    if (command_count > 0) {
        last -> next = cmd;
        cmd -> prev = last;
    }
    else {
        top = cmd;
        nextup = cmd;
        cmd->prev = NULL;
    }
}

```

```

last = cmd;
cmd->next = NULL;
command_count++;
unsend_command_count++;

/* now command is in the list */
if (nextup == NULL) /* if we dequeued in the meantime */
    nextup = cmd;
return cmd;
}

Command_obj *Command_list::insert_next_command (World *w, Command_obj *cmd) {
    nextup = insert_before (w, nextup, cmd);
    return cmd;
}

Command_obj *Command_list::insert_before (World *w, Command_obj *before, Command_obj
*cmd) {
    if ((nextup == NULL) || (before == NULL)) // no commands waiting, so append
        return append_command (w, cmd);
    else /* fix list */
        Command_obj *tmp = before;
        if (tmp->prev != NULL)
            tmp->prev->next = cmd;
        cmd->prev = tmp->prev;
        tmp->prev = cmd;
        cmd->next = tmp;

        /* set new nextup */
        command_count++;
        unsend_command_count++;
        return cmd;
    }
}

Command_obj *Command_list::send_next_command (World *w) {
    Command_obj *retval = NULL;

    /* repeat while we haven't successfully sent a command from this list */
    while ((nextup != NULL) && (retval == NULL)) {
        /* repeat until the next command is unsend */
        while ((nextup != NULL) && (nextup->sent))
            nextup = nextup -> next;
        if (nextup != NULL) { /* could have broken out of loop
            because the list have no unsend commands */
            retval = nextup->send_command (w);
            if (retval == NULL)
                nextup = nextup -> next;
        }
        if (retval != NULL)
            if (retval->sent) {
                unsend_command_count--;
                nextup = nextup->next;
            }
        return retval;
    }

    void Command_list::delete_command (Command_obj *obj) {
        if (obj != NULL) {
            /* deal with nextup, with the possibility that we get interrupted */
            if (nextup != NULL) {
                Command_obj *temp = nextup->next;
                if (obj == nextup) //what about if obj == nextup->next?, I think its okay

```

```

    }
    nextup = temp;
}
if (!obj->sent)
    unsent_command_count --;
obj -> sent = 1;

if (obj->prev != NULL)
    obj->prev->next = obj->next;
if (obj->next != NULL)
    obj->next->prev = obj->prev;
if (obj == top)
    top = obj->next;
if (obj == last)
    last = obj->prev;

delete obj;
command_count--;
}
}

Action_obj::Action_obj () {
    post_info = NULL;
    world = NULL;
    next = prev = NULL;
}

int Action_obj::update () {
    return (command_list.nextup != NULL);
}

Action_obj::~Action_obj () {
    world = NULL;
    next = prev = NULL;
    if (post_info != NULL) {
        delete post_info;
        post_info = NULL;
    }
}

int Action_obj::nextup_is_cvs () {
    if (command_list.nextup == NULL)
        return 0;
    else
        return ((command_list.nextup->command_type == CHANGE_VIEW_COMMAND) ||
                (command_list.nextup->command_type == SAY_COMMAND));
}

Move_action::Move_action(World* w, double x, double y) {
    action_type = MOVE_ACTION;
    command_list.append_command (w, new Command_obj(MOVE_COMMAND, x, y));
    x_coord = x;
    y_coord = y;
    world = w;
    next = prev = NULL;
    strcpy(name, "move");
}

Dash_action::Dash_action(World* w, double pow) {
    action_type = DASH_ACTION;
    command_list.append_command (w, new Command_obj(DASH_COMMAND, pow, ARG_ERR));
    power = pow;
    world = w;
    next = prev = NULL;
    strcpy(name, "dash");
}

Turn_action::Turn_action(World *w, double ang) {
    action_type = TURN_ACTION;
    command_list.append_command (w, new Command_obj(TURN_COMMAND, ang, ARG_ERR));
    angle = ang;
    // next_command = 0;
}

```

```

// num_commands = 1;
world = w;
next = prev = NULL;
strcpy(name, "turn");
}

Kick_action::Kick_action () {
    action_type = KICK_ACTION;
    world = NULL;
    next = prev = NULL;
    strcpy(name, "***UNINITIALIZED *** kick");
}

Kick_action::Kick_action(World* w, double pow, double ang) {
    fill_kick_action (w, pow, ang);
}

void Kick_action::fill_kick_action (World* w, double pow, double ang) {
    // command_list = new Command_obj* [1];
    // * A kick_action only takes 1 command*/
    action_type = KICK_ACTION;
    command_list.append_command (w, new Command_obj(KICK_COMMAND, pow, ang));
    angle = ang;
    power = pow;
    // next_command = 0;
    // num_commands = 1;
    world = w;
    next = prev = NULL;
    strcpy(name, "kick");
}

Say_action::Say_action(World* w, char* msg_in ) {
    action_type = SAY_ACTION;
    command_list.append_command (w, new Command_obj(SAY_COMMAND, msg_in));
    strcpy(msg,msg_in);
    // next_command = 0;
    // num_commands = 1;
    world = w;
    next = prev = NULL;
    strcpy(name, "say");
}

Sense_body_action::Sense_body_action(World* w) {
    action_type = SENSE_BODY_ACTION;
    // command_list[0] = new Command_obj();
    command_list.append_command (w, new Command_obj(SENSE_BODY_COMMAND,0,0));
    world = w;
    next = prev = NULL;
    strcpy(name, "sense body");
}

Change_view_action::Change_view_action(World* w, int width, int quality) {
    action_type = CHANGE_VIEW_ACTION;
    command_list.append_command (w, new Command_obj(CHANGE_VIEW_COMMAND, width, quality));
    world = w;
    next = prev = NULL;
    strcpy(name, "change_view");
}

/* *****
 * Action_obj
 * *****
Command_obj* Action_obj::send_next_command() {
    if (command_list.nextup == NULL)
        return NULL;
    else
        return command_list.send_next_command (world);
}

int Action_obj::is_finished() {
    return (command_list.nextup == NULL);
}

```

```

void Action_obj::fprintf_action(FILE *output_file) {
    Command_obj *temp = command_list.top;
    char reached_new_commands = 0;

    fprintf(output_file, "\tCommands: %d, (Unsent: %d)\n", command_list.command_count
    , command_list.unsent_command_count);
    while (temp != NULL) {
        fprintf(output_file, "\t");
        if (temp == command_list.nextup) // This command is the first not to have been seen
            nt
            reached_new_commands = 1;
        fprintf(output_file, (reached_new_commands) ? " " : "**** ");
        temp->fprintf_command(output_file);
        if (temp->sent)
            fprintf(output_file, " (sent) ");
        fprintf(output_file, "\n");
        temp = temp->next;
    }
}

int Watch_ball_action::update() {
    do_watch_ball(world, s);
    return 1;
}

Watch_ball_action::Watch_ball_action(World *w, Self_obj *s, int urgent, int patience
e_rounds) {
    world = w;
    this->s = s;
    next = prev = NULL;
    post_info = NULL;
    patience = patience_rounds;

    /* set missing_ball according to urgent;
    if missing_ball starts at zero, we wait at current facing
    if the ball was in our cone of vision last time we saw it, and it
    might just be too far away to see. We will wait at our current facing
    for a patience rounds number of rounds before getting into a tiff
    about finding the ball.

    if urgent is true, we go into a tiff right away
    */
    missing_ball = (urgent) ? patience_rounds + 1 : 1;
    do_watch_ball(world, s);
    strcpy(name, "Watch_ball_action");
    action_type = WATCH_BALL_ACTION;
}

void Watch_ball_action::do_watch_ball(World *w, Self_obj *s) {
    command_list.empty();
    Pair disp;

    disp = w->ball->position - s->position;
    double angle_now;

    angle_now = normalize_rad (disp.angle() - s->facing * DEG2RAD);
    double half_view_width_angle;

    switch (s->view_width) {
    case VIEW_WIDTH_NARROW:
        half_view_width_angle = P125PI;
        break;

```

```

    case VIEW_WIDTH_NORMAL:
        half_view_width_angle = P25PI;
        break;
    case VIEW_WIDTH_WIDE:
        half_view_width_angle = P5PI;
        break;
    }

    if (w->ball->see_timestamp == w->last_vis_info_time) { /* we see the ball */
        if (missing_ball > 0)
            w->debug("I see the ball now.\n");
        missing_ball = 0;
        find_angle_to_seen_ball (&(w->ball->position), &(w->ball->velocity), &(s->positi
on),
                                &(s->velocity), s->facing);
    }
    else {
        if (abs (angle_now) <= (half_view_width_angle)) { /* last time, ball was in
cone of view but we didn't see it */
            &(s->velocity), s->facing);
        }
        else {
            if ((disp.r() > 20) && (missing_ball <= patience)) { /* chance ball is sti
ll in cone */
                missing_ball++;
                find_angle_to_seen_ball (&(w->ball->position), &(w->ball->velocity), &(s
->position),
                                        &(s->velocity), s->facing);
                fprintf(w->output_file, "I don't see the ball yet. In %d rounds I'm gon
n
a have a fit\n", patience - missing_ball);
            }
            else {
                /* either ball used to be close enough so we should see it,
                or we're out of patience */
                missing_ball++;
                cant_see_ball_panic ();
                w->debug("That's it. I can't find the ball, and I'm tired of waiting.\n
I'm going to spin and try to find it.\n");
            }
        }
        /* end ball not in cone */
        else { /* last seen ball is outside angle */
            missing_ball++;
            /* I think this is structured so that if this clause is reached, then eith
er we just started looking
            for the ball, or we haven't been able to see it for a while, or the bal
l just changed direction */
            cant_see_ball_panic ();
            #ifdef WATCH_DEBUG_MODE
                w->debug("I have no idea where the ball is, so I'm going to spin and try t
o find it.\n");
            #endif
            fprintf (world->output_file, "\tI saw the ball last at round %d, got last
visinfo at %d, and it's now round %d\n",
                    world->ball->see_timestamp, world->last_vis_info_time, world->ti
me_now);
        }
    }
    #endif

    /* placeholder so that we don't lose our command */
    command_list.append_command (w, new Command_obj (HALT_COMMAND, ARG_ERR, ARG_ERR));

    #ifdef WATCH_DEBUG_MODE
        //fprintf_action (world->output_file);
    #endif
}

void Watch_ball_action::cant_see_ball_panic () {
    /* this probably could be more intelligent */
    command_list.append_command (world, new Command_obj (TURN_COMMAND, 180, ARG_ERR));
}

void Watch_ball_action::find_angle_to_seen_ball (Pair *ball_position, Pair *ball_vel
ocity, Pair *us_position,

```

```

Pair *us_velocity, double facing) {

Pair next_round_displacement;
double needed_turn_angle;
double my_speed = us_velocity->r();
double should_do_turn_angle;

const rounds_to_watch = 2;

for (int round = 1; rounds_to_watch >= round; round++) {
if (ball_velocity->error < ball_velocity->r()) /*if error is too large, the ball
probably isn't moving
o far away a turn
    *ball_position += *ball_velocity;
*us_position += *us_velocity;
next_round_displacement = *ball_position - *us_position;
needed_turn_angle = normalize_rad (next_round_displacement.angle() - facing*DEG2
RAD);
should_do_turn_angle = s->compute_needed_angle_at_speed (needed_turn_angle, my_s
peed);
if (abs (should_do_turn_angle) > PI) {
should_do_turn_angle = Sign(should_do_turn_angle) * PI;
/* figure out the amount we will turn for later */
needed_turn_angle = s->compute_turned_angle_at_speed (should_do_turn_angle * R
AD2DEG, my_speed);
if (abs(should_do_turn_angle) > (1 * DEG2RAD)) {
command_list.append_command (world, new Command_obj (TURN_COMMAND,
should_do_turn_angle * RA
D2DEG, ARG_ERR));
} else {
needed_turn_angle = 0;
}
if (rounds_to_watch != round) { /* don't bother on last iteration */
facing += RAD2DEG * needed_turn_angle;
*us_velocity *= PLAYER_DECAY;
*ball_velocity *= BALL_DECAY;
}
}
}

/* *****
Action_O
* *****
Action_O::Action_O(World* w) {
first_obj = last_obj = NULL;
num_actions = 0;
queue_locked = FALSE;
}

Action_O::~Action_O() {
wipe();
}

void Action_O::wipe () {
Action_obj* temp, *a;
a = first_obj;
num_actions = 0;
while (a != NULL ) {
temp = a->next;
delete a;
a = temp;
}
last_obj = first_obj = NULL;
}

```

```

}

/* A dangerous function, to be used with caution
int Action_O::insert_first(Action_obj* action) {
queue_locked = TRUE;
if (action == NULL)
return num_actions;
if (num_actions == 0) {
first_obj = last_obj = action;
} else {
Action_obj* temp;
temp = first_obj;
first_obj = action;
first_obj->next = temp;
temp->prev = first_obj;
first_obj->prev = NULL;
}
num_actions++;
queue_locked = FALSE;
return num_actions;
}

int Action_O::enqueue(Action_obj* action) {
queue_locked = TRUE;
if (action == NULL)
return num_actions;
if (num_actions == 0) {
first_obj = last_obj = action;
action->next = action->prev = NULL;
} else {
Action_obj* temp;
temp = last_obj;
action->prev = temp;
temp->next = action;
last_obj = action;
action->next = NULL;
}
num_actions++;
queue_locked = FALSE;
return num_actions;
}

void Action_O::remove_first_action () {
Action_obj* temp = first_obj;
if (first_obj == NULL)
return;
first_obj = first_obj->next;
if (first_obj != NULL)
first_obj->prev = NULL;
delete temp;
temp = NULL;
num_actions--;
}

Command_obj* Action_O::dequeue () {
// if (queue_locked) assert(0); //commented for debugging of enqueue_before_last
Action_obj *next2do = first_obj;
next2do = next_unfinished_action ();
if (next2do == NULL)
return NULL;
else
return next2do->send_next_command();
}

Action_obj* Action_O::peek(int index) {
if (( num_actions == 0 || (index < 0))
return NULL;
Action_obj* ret_val = first_obj;
}

```

```

for (int i = 0 ; i < index ; i++, ret_val = ret_val->next ) {
    if (ret_val == NULL)
        return NULL;
    return ret_val;
}

// Returns number of actions that were removed
int Action_Q::remove_actions(int start_index) {
    Action_obj* first_to_remove;
    Action_obj* last_to_leave;
    Action_obj* temp;
    if (num_actions == 0)
        return 0;

    int ret_val = num_actions - start_index;
    if (start_index < 0) {
        return K_NOT_USED;
    } else if (start_index > 0) {
        last_to_leave = peek(start_index - 1);
        first_to_remove = last_to_leave->next;
        last_to_leave->next = NULL;
    } else
        first_to_remove = first_obj;

    while (first_to_remove != NULL ) {
        temp = first_to_remove;
        first_to_remove = first_to_remove->next;
        delete temp;
        temp = NULL;
        num_actions--;
    }
    return ret_val;
}

int Action_Q::size() {
    return num_actions;
}

void Action_Q::update () {
    /* update the first action, if it exists. If the update returns null, then
    the action is [successfully] done and we delete it and repeat */
    int update_retval;
    while (num_actions != 0) {
        update_retval = first_obj->update();
        if ((update_retval == 0) || (update_retval == 2))
            remove_first_action();
        else
            break;
    }
}

Action_obj * Action_Q::next_unfinished_action (void) {
    Action_obj *next_unfinished = first_obj;
    int num_actions_left = num_actions;

    if (num_actions == 0)
        return NULL;

    /* don't want to send the next command in this action if there isn't one */
    while ( (num_actions_left > 0) && (next_unfinished->is_finished()) ) {
        num_actions_left--;
        next_unfinished = next_unfinished->next;
    }
    return (num_actions_left == 0) ? NULL: next_unfinished;
}

int Action_Q::enqueue_before_last(Action_obj* action) {

```

```

queue_locked = TRUE;
if (action == NULL)
    return num_actions;
if (num_actions == 0) {
    first_obj = last_obj = action;
    action->next = action->prev = NULL;
} else {
    action->prev = last_obj->prev;
    last_obj->prev = action;
    action->next = last_obj;
    if (action->prev != NULL)
        action->prev->next = action;
    if (first_obj == last_obj)
        first_obj = action;
}
num_actions++;
queue_locked = FALSE;
return num_actions;
}

void Action_Q::fprintf_queue(FILE *output_file) {
    Action_obj *a;
    int i;
    if (num_actions == 0) {
        fprintf(output_file, "The queue is empty\n");
        return;
    }
    fprintf(output_file, "The queue contains %d actions.\n", num_actions);
    for (a = first_obj, i = 1 ; a != NULL ; a = a->next, i++) {
        fprintf(output_file, "Action #%d is a %s action with the following commands:\n",
            i, a->name);
        a->fprintf_action(output_file);
    }
}

/*****
| POST INFO, YO!
| void Post_info::init() {
|     parent_action = NULL;
|     my_pos = my_vel = ball_pos = ball_vel = NULL;
|     my_facing = NULL;
|     expected_round = parent_action_type = K_NOT_USED;
| }
|
| Post_info::Post_info() {
|     init();
| }
|
| Post_info::~Post_info() {
|     clear();
| }
|
| Post_info::Post_info(Post_info *pi) {
|     init();
|     if (pi != NULL) {
|         set_my_vel(pi->my_vel);
|         set_my_pos(pi->my_pos);
|         set_my_facing(pi->my_facing);
|         set_ball_vel(pi->ball_vel);
|         set_ball_pos(pi->ball_pos);
|         parent_action = pi->parent_action;
|         expected_round = pi->expected_round;
|         parent_action_type = pi->parent_action_type;
|     }
| }
|
| Post_info::Post_info(Action_obj *parent) {
|     init();
|     parent_action = parent;
| }
|
| *****/

```

```

if (parent != NULL) {
    parent_action_type = parent->action_type;
}
}

void Post_info::clear() {
    if (my_pos != NULL) {
        delete my_pos;
        my_pos = NULL;
    }
    if (my_facing != NULL) {
        delete my_facing;
        my_facing = NULL;
    }
    if (my_vel != NULL) {
        delete my_vel;
        my_vel = NULL;
    }
    if (ball_vel != NULL) {
        delete ball_vel;
        ball_vel = NULL;
    }
}

if (ball_pos != NULL) {
    ball_pos = NULL;
    delete ball_pos;
}

parent_action = NULL;
expected_round = parent_action_type = K_NOT_USED;
}

void Post_info::fill(World *world) {
    clear();
    my_pos = new Pair(world->me->position.x, world->me->position.y);
    my_vel = new Pair(world->me->velocity.x, world->me->velocity.y);
    ball_pos = new Pair(world->ball->position.x, world->ball->position.y);
    my_facing = new double;
    *my_facing = world->me->facing;
    ball_vel = new Pair(world->ball->velocity.x, world->ball->velocity.y);
    ball_vel->error = world->ball->velocity.error;
    expected_round = world->last_vis_info_time;
}

void Post_info::complete_nulls(World *world) {
    if (my_pos == NULL) {
        my_pos = new Pair(world->me->position.x, world->me->position.y);
    }
    if (my_vel == NULL) {
        my_vel = new Pair(world->me->velocity.x, world->me->velocity.y);
    }
    if (ball_pos == NULL) {
        ball_pos = new Pair(world->ball->position.x, world->ball->position.y);
    }
    if (my_facing == NULL) {
        my_facing = new double;
        *my_facing = world->me->facing;
    }
    if (ball_vel == NULL) {
        ball_vel = new Pair(world->ball->velocity.x, world->ball->velocity.y);
        ball_vel->error = world->ball->velocity.error;
    }
}

void Post_info::set_my_vel(Pair *my_v) {
    if (my_v == NULL) return;
    if (my_vel != NULL) {
        delete my_vel;
        my_vel = NULL;
    }
}

```

```

}
my_vel = new Pair (my_v->x, my_v->y);
}

void Post_info::set_my_pos(Pair *my_p) {
    if (my_p == NULL) return;
    if (my_pos != NULL) {
        delete my_pos;
        my_pos = NULL;
    }
    my_pos = new Pair (my_p->x, my_p->y);
}

void Post_info::set_my_facing(double *my_f) {
    if (my_f == NULL) return;
    if (my_facing != NULL) {
        delete my_facing;
        my_facing = NULL;
    }
    my_facing = new double;
    *my_facing = *my_f;
}

void Post_info::set_ball_vel(Pair *ball_v) {
    if (ball_v == NULL) return;
    if (ball_vel != NULL) {
        delete ball_vel;
        ball_vel = NULL;
    }
    ball_vel = new Pair(ball_v->x, ball_v->y);
    ball_vel->error = ball_v->error;
}

void Post_info::set_ball_pos(Pair *ball_p) {
    if (ball_p == NULL) return;
    if (ball_pos != NULL) {
        delete ball_pos;
        ball_pos = NULL;
    }
    ball_pos = new Pair (ball_p->x, ball_p->y);
}
}

```

```

// Home includes
#include <new_action.hh>
#include <robocup_com.hh>
#include <world.hh>
#include <robocup_act.hh>
#include <sensor.hh>

#include <libscilient.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

#include <iostream.h>
#include <sstream.h>
#include <utils.hh>

void prepare_starting_queue(World *the_world, void *aq_v) {
    // Enqueue find ball
    Action_Queue *aq = (Action_Queue*) aq_v;
    Find_ball *fba = new Find_ball(the_world);
    aq->insert_first(fba);

    Move_to *mifa = new Move_to(the_world, the_world->me, the_world->ball->position,
        100.0, 2.0, FALSE, TRUE);
    aq->insert_last(mifa);
    Kick_ball_action *kba = new Kick_ball_action(the_world);
    aq->insert_last(kba);

    mifa->world = the_world; // Magic
}

int main(int argc, char **argv) {
    FILE *outfile;
    Socket sock;
    char *server;
    int port;
    InitInfo info;
    char tname[16];
    char* cmd;
    char output_fname[16];
    Action_Queue *aq = (Action_Queue*) NULL;

    if(argc == 6) {
        strcpy(tname, argv[1]);
        server = argv[2];
        port = atoi(argv[3]);
        //port = 6000;
        strcpy(output_fname, argv[5]);
    } else if (argc == 5) {
        strcpy(tname, argv[1]);
        server = argv[2];
        port = 6000;
        if (strcmp(argv[3], "-f") == 0) {
            strcpy(output_fname, argv[4]);
        } else {
            printf("usage: %s TEAMNAME [HOST [PORT]] [-f OUTPUT_FILENAME] \n", \
                argv[0]);
            exit(0);
        }
    } else if (argc == 4) {
        strcpy(tname, argv[1]);
        if (strcmp(argv[2], "-f") == 0) {
            server = "localhost";
            port = 6000;
        }
    }
}

```

```

        strcpy(output_fname, argv[3]);
    } else {
        strcpy(tname, argv[1]);
        server = argv[2];
        port = atoi(argv[3]);
        strcpy(output_fname, "stdout");
    }
} else if (argc == 3) {
    strcpy(tname, argv[1]);
    server = argv[2];
    port = 6000;
    strcpy(output_fname, "stdout");
} else if (argc == 2) {
    strcpy(tname, argv[1]);
    server = "localhost";
    port = 6000;
    strcpy(output_fname, "stdout");
} else {
    printf("usage: %s TEAMNAME [HOST [PORT]] [-f OUTPUT_FILENAME] \n", argv[0]);
    exit(0);
}
if (strcmp(output_fname, "stdout") == 0) {
    strcpy(output_fname, "/dev/tty");
}

outfile = fdopen(open(output_fname,
    O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
    S_IRWXU | S_IRWXO | S_IRWXG), "a");
setvbuf(outfile, (char*) NULL, _IONBF, 0);

sock = init_connection(server, port);
if(sock.socketfd == -1)
    exit(1);

info = send_com_init(&sock, tname);
fprintf(outfile, "\ninit!: %d, %d, %s, %d\n\n",
    info.side, info.unum, info.playmodestr, info.playmode);

if (info.side == S_UNKNOWN) {
    fprintf(outfile, "Can't init\n");
    exit(-1);
}

World* the_world = new World(&sock);
the_world->me->sock = &sock; // Don't know who, but this needs to be here
// a simple init which just makes a World and fills in its socket.
the_world->fill_world(info.side, tname, info.unum, info.playmodestr, outfile, FALSE);
E);

// A new empty queue
aq = new Action_Queue();

randomize(); // Seed the random number generator
// This will put us in a random position on the field
the_world->send_initial_move();

// Get some vis info
Vis_info* vi = NULL;
int got_hear_info;
while (vi == NULL)
    vi = the_world->get_info(&got_hear_info);
delete vi;
vi = NULL;

prepare_starting_queue(the_world, aq);

```

```
// This will poll the incoming socket and await the start of the game
the_world->wait_for_mode_change();
aq->start_timer(); // This should send the commands
Update_info *ui;
aq->update_first_action_only = TRUE;
while ((the_world->mode != BEFORE_KICK_OFF) &&
      (the_world->mode != TIME_OVER)) {
    sigpause(SIGUSR1);
    // Count counts alarms since last info
    if (the_world->alarms_since_last_info > 40) {
        close_connection(*the_world->me->sock);
        printf("Shutting down player %d (server seems DED. . .)\n", the_world->me->u_number);
        goto END;
    }
    if (aq->is_empty()) {
        cout << "Player " << the_world->me->u_number
              << " is at destination. Quitting." << endl;
        goto END;
    }
    spin_lock(aq->q_lock);
    aq->clean_finished(); // Removes the finished actions
    vi = the_world->get_info(&got_hear_info);
    if (vi != NULL) {
        the_world->alarms_since_last_info = 0;
        the_world->last_vis_info_time = vi->timestamp;
        delete vi;
        vi = NULL;
        ui = aq->update_queue();
        if (ui != NULL) {
            delete ui;
            ui = NULL;
        }
    }
    release_lock(aq->q_lock);
}

END:
if (the_world != NULL)
    delete the_world;
if (aq != NULL)
    delete aq;
exit(0); //return 0;
}
```

```

#include <command.hh>

Command_obj::Command_obj() {
    next = NULL;
    prev = NULL;
}

Command_obj::~Command_obj() {
    next = NULL;
    prev = NULL;
}

Command_list::Command_list() {
    null_all();
}

Command_list::~Command_list() {
    remove_all(); // Deletes all nodes. Will also null the pointers
}

void Command_list::remove_all() {
    Command_obj *temp;
    while (first != NULL) {
        temp = first;
        first = first->next;
        delete temp;
        temp = NULL;
    }
}

void null_all();

void Command_list::null_all() {
    first = last = next_to_send = NULL;
}

void Command_list::insert_into_blank_list(Command_obj *com) {
    last = first = next_to_send = com;
    com->next = com->prev = NULL;
}

void Command_list::check_list() {
    Command_obj *temp = first;
    Command_obj * temp_prev = NULL;
    bool saw_nextup = FALSE;
    while (temp != NULL) {
        if (temp == next_to_send)
            saw_nextup = TRUE;
        temp_prev = temp;
        temp = temp->next;
    }
    if (!is_empty()) {
        assert(saw_nextup);
        assert(temp_prev == last);
    } else {
        assert(first == last);
        assert(next_to_send == NULL);
        assert(last == NULL);
    }
}

// This function needs to be atomized
// But maybe not as long it is only called through the
// signal handler
Command_obj * Command_list::send_next_command() {
    check_list();
    Command_obj *ret_val;
    if (next_to_send != NULL) {
        ret_val = next_to_send;
    }
}

```

```

int num_advance = next_to_send->send_command();
if (num_advance == 0) // Do not advance next to send
    return ret_val;
if (num_advance > 0)
    for (; (num_advance > 0) && (next_to_send != NULL); num_advance--)
        next_to_send = next_to_send->next;
else
    for (; (num_advance < 0) && (next_to_send != NULL); num_advance++)
        next_to_send = next_to_send->prev;
}
check_list();
return ret_val;
}

void Command_list::insert_last(Command_obj * com) {
    check_list();
    char i = 0;
    i++;
    if (com != NULL) {
        if (is_empty()) {
            last->next = com;
            com->prev = last;
            com->next = NULL;
        } else {
            insert_into_blank_list(com);
        }
    }
    check_list();
}

void Command_list::insert_first(Command_obj *com) {
    if (com != NULL) {
        if (is_empty()) {
            first->prev = com;
            com->next = first;
            com->prev = NULL;
        } else {
            first = com;
            insert_into_blank_list(com);
        }
    }
    check_list();
}

// This is will insert before the next to send, making
// com be next instead
// If nextup is null, this is inserted at the end of the list
void Command_list::insert_as_next(Command_obj *com) {
    check_list();
    if (com != NULL) {
        if (is_empty()) {
            insert_last(com);
        } else {
            com->prev = next_to_send->prev;
            if (com->prev != NULL)
                com->prev->next = com;
            next_to_send = com;
            next_to_send->prev = com;
            if (next_to_send == first)
                first = com;
            next_to_send = com;
        }
    } else {
        insert_into_blank_list(com);
    }
}

// Have all the commands in the list been sent
int Command_list::is_finished() {
}

```

```

    return (next_to_send == NULL);
}

// This also ensures that if one of the three pointers to list elements is null
// then so are the other two, and if one is non-NULL, then so are the other two.
// Note: next_to_send could be null in a non-empty list, if all the commands have
// been sent
int Command_list::is_empty() {
    if (first == NULL) {
        // The list is empty
        return(1);
    }
    // Broken list assertion, first is null, but last is not
    assert(0);
} else if (last == NULL)
} // first is not null, but last is => broken list
assert(0);
else
// The list is not empty
return(0);
}

// This does a basic pointer compare
int Command_list::is_in_list(Command_obj *com) {
    Command_obj *temp = NULL;
    if (com == NULL) return(0);
    for (temp = first; temp != NULL; temp = temp->next) {
        if (temp == com)
            return(1);
    }
    return(0);
}

// Returns a bool for success or failure
// Fails if com is null or if before is not in the list
// Note: if before==next to send, com does NOT become next_to_send,
// but is inserted before it
int Command_list::insert_before(Command_obj *before, Command_obj *com) {
    check_list();
    if ((com == NULL) || (is_in_list(before)))
        return(0);
    com->prev = before->prev;
    com->next = before;
    if (com->prev != NULL)
        com->prev->next = com;
    before->prev = com;
    if (before == first)
        first = com;
    check_list();
    return(1);
}

// If we are deleting the next_to_send -- next_to_send will advance to the
// next command, or become NULL
int Command_list::delete_command(Command_obj *com) {
    if (com == NULL)
        return(0);
    if (com->prev != NULL)
        com->prev->next = com->next;
    if (com->next != NULL)
        com->next->prev = com->prev;
    if (com == first)
        first = com->next;
    if (com == last)
        last = com->prev;
    if (com == next_to_send)
        next_to_send = com->next;
    if (com != NULL) {

```

```

        delete com;
        com = NULL;
    }
    return(1);
}

Command_obj* Command_list::get_next() {
    return next_to_send;
}

int Command_list::advance_next(int num_advance) {
    int ret_val = 0;
    if (is_empty() || (num_advance == 0)) // Do not advance next up
        return ret_val;
    else if (num_advance > 0) {
        for (; (num_advance > 0) && (next_to_send != NULL); num_advance--) {
            next_to_send = next_to_send->next;
            ret_val++;
        }
    } else if (num_advance < 0) {
        for (; ((num_advance < 0) && (next_to_send != NULL)); num_advance++) {
            ret_val--;
        }
    }
    assert(0); // One of the three above has to be true
    return ret_val;
}

void Command_obj::print() {
    cout << "Command ADT" << endl;
}

void Command_list::print() {
    if (is_empty())
        cout << "List is empty" << endl;
    else {
        int i, next_ind = -1;
        Command_obj *temp;
        for (i = 0, temp = first; temp != NULL; temp = temp->next, i++) {
            if (temp == next_to_send) next_ind = i;
        };
        cout << "Command list has " << i << " commands. Command # "
            << next_ind << " is nextup." << endl;
        for (temp = first; temp != NULL; temp = temp->next) {
            cout << "\t", temp->print();
        }
    }
}

```

```

// --c++--
// The above line causes xemacs to open this file in c++-mode
#include <assert.h>
#include <sensor.h>

State_data::State_data() {
Facing::Facing() {}
Facing::Facing(int dir) {
    facing = dir;
}

/* Moving_obj - empty constructor, inits debug fields only at this time */
Moving_obj::Moving_obj () {
}

Ball_obj::Ball_obj () {
//odo data
odo_position.x = 0.0;
odo_position.y = 0.0;
odo_velocity.x = 0.0;
odo_velocity.y = 0.0;
odo_timestamp = -1;

//most recent data
// Sometimes odo goes here,
// but usually the visual info does (position inherited from Object.)
position.x = 0.0;
position.y = 0.0;
velocity.x = 0.0;
velocity.y = 0.0;
see_timestamp = timestamp = -1;
}

/* *****
* compute ball steps:
* computes discrete timesteps of ball -
* returns array, s.t. array[i] = distance ball will be at time i
* last entry k in array is the one s.t. array[k+1] > max_distance
*/

double *Ball_obj::compute_ball_steps (
    double speed,
    double max_distance,
    int max_rounds,
    double speed_cap,
    int *size) /* 0: size of returned array */
{
    double minspeed = speed;
    int round_cap;
    assert ((max_distance >= 0) && (speed >= 0) && (max_rounds >= 0));
    if (speed == 0)
        round_cap = (speed >= speed_cap) ? max_rounds : 0;
    else
        /* do {
            round_cap = (int) (max_distance / minspeed) + 1;
            minspeed = speed * x_to_y (.94, round_cap-1);
        } while ((round_cap < max_rounds) && ( (int) (max_distance / minspeed) != round
            cap) && (minspeed >= speed_cap));
        */
        round_cap = max_rounds;

    double *step_array;
    int i;
    double tdistance = 0;

```

```

step_array = new double [round_cap + 1];
i = 0;
do {
    assert (i <= round_cap);
    step_array[i] = tdistance;
    tdistance = step_array[i] + speed;
    speed *= BALL_DECAY;
    i++;
} while ((tdistance <= max_distance) && (i <= max_rounds) && (speed >= speed_cap));
*size = i;
return (step_array);
}

Line_obj::Line_obj(int type) {
//! It is not clear what the position of the line should store
//! A suggestion: the inward-facing normal unit vector?
line_type = type;
}

Goal_obj::Goal_obj(int goal_side) {
side = goal_side;
position.y = 0.0;
if (side == LEFT) {
    position.x = -PITCH_LENGTH/2.0;
} else {
    position.x = PITCH_LENGTH/2.0;
}
}

void Ball_obj::noda_step(int cur_timestamp) {
Pair old_velocity = odo_velocity;
odo_position = odo_position + odo_velocity;
odo_velocity = odo_velocity * BALL_DECAY;
odo_velocity_error = old_velocity_error;
timestamp = odo_timestamp = cur_timestamp;
}

void Ball_obj::get_ball_odo_up_to_speed(World* world) {
for (int i = 0 ; i < (world->last_vis_info_time - odo_timestamp) ; i++) {
    noda_step(world->last_vis_info_time);
}
velocity = odo_velocity;
timestamp = odo_timestamp = world->last_vis_info_time;
}

Opponent_obj::Opponent_obj () {
u_number = VIS_UNUM_ERR;
is_not_dead = FALSE;
}

Teammate_obj::Teammate_obj () {
u_number = VIS_UNUM_ERR;
is_not_dead = FALSE;
}

Player_obj::Player_obj () {
is_not_dead = FALSE;
}

/* *****
* Player::rounds to point
*
*/

int Player_obj::rounds_to_point (Pair *pt, double tolerance, double max_power) {
return (hypothetical_rounds_to_point (pt, tolerance, &(this->position)),

```

```

    &(this->velocity), this->facing, max_power))
}

int Player_obj::rounds_to_point (Pair *pt, double tolerance) {
    return (hypothetical_rounds_to_point (pt, tolerance, &(this->position),
    &(this->velocity), this->facing, 100));
}

/* note velocity is only used to compute tangential displacement and not max rounds
to a
point. we always assume the highest possible speed */
int Player_obj::hypothetical_rounds_to_point (Pair *pt, double tolerance, Pair *posi
tion,
Pair *velocity, double facing, double
max_power) {
    Pair abs_displacement = *pt - *position;
    int rounds = 0;
    double radial_speed;
    if (abs displacement.r() <= tolerance)
        return 0; //wow thats quick
    //if (displacement.r() / PLAYER_SPEED_MAX
    //be smart
    double tangent_angle = abs displacement.angle() + P5PI;
    Pair unit_tangential (cos (tangent_angle), sin (tangent_angle));
    Pair displacement = abs displacement - (/*PLAYER_DECAYED_DIST * */ unit_tangential);
    1 * velocity->dot (unit_tangential);
    double dist_remaining = displacement.r();

    /* determine if turn needs to happen */
    if (facing != VIS_DIR_ERR) {
        if (abs (sin (displacement.angle() - facing * DEG2RAD)) * displacement.r() > tol
erance) {
            dist_remaining -= displacement.dot (*velocity) / (dist_remaining); // * (1/dis
t_remaining) to get unit vector
            rounds++;
        }
    }
    else
        /*don't know what's right here; assume
        his facing is not correct */
        rounds ++;

    double max_speed_at_power = min (max_power * POWERRATE * PLAYER_DIST_FROM_VEL_FACT
OR, PLAYER_SPEED_MAX);
    /* now, since player's max speed is attainable in 1 dash assume max speed to poin
t */
    if (dist_remaining > tolerance)
        rounds += ceiling( (dist_remaining - tolerance) / max_speed_at_power);
    return rounds;
}

double Player_obj::compute_needed_angle_at_speed (double angle, double speed) {
    return ((1 + MOMENT_INERTIA * speed) * angle);
}

double Player_obj::compute_turned_angle_at_speed (double angle, double speed) {
    return (angle / (1 + MOMENT_INERTIA * speed));
}

double Player_obj::find_kick_into_self_angle (double distance) {
    return ((BALL_SIZE + PLAYER_SIZE) > distance) ? 0 : (1.10) * asin ((BALL_SIZE +
PLAYER_SIZE)/distance));
}

```

```

// **c++**
// The above line causes xemacs to open this file in c++-mode
#include <iostream.h>
#include <object.hh>
#include <param.hh>

Flag_obj::Flag_obj(int type) {
    flag_type = type;
    switch(flag_type) {
    case TOP_LEFT_FLAG:
        position.x = -PITCH_LENGTH/2.0;
        position.y = -PITCH_WIDTH/2.0;
        break;
    case TOP_RIGHT_FLAG:
        position.x = PITCH_LENGTH/2.0;
        position.y = -PITCH_WIDTH/2.0;
        break;
    case BOTTOM_LEFT_FLAG:
        position.x = -PITCH_LENGTH/2.0;
        position.y = PITCH_WIDTH/2.0;
        break;
    case BOTTOM_RIGHT_FLAG:
        position.x = PITCH_LENGTH/2.0;
        position.y = PITCH_WIDTH/2.0;
        break;
    case TOP_CENTER_FLAG:
        position.x = 0.0;
        position.y = -PITCH_WIDTH/2.0;
        break;
    case BOTTOM_CENTER_FLAG:
        position.x = 0.0;
        position.y = PITCH_WIDTH/2.0;
        break;
    case LEFT_PBOX_TOP_FLAG:
        position.x = -PITCH_LENGTH/2.0+PENALTY_AREA_LENGTH;
        position.y = -PENALTY_AREA_WIDTH/2.0;
        break;
    case LEFT_PBOX_BOTTOM_FLAG:
        position.x = -PITCH_LENGTH/2.0+PENALTY_AREA_LENGTH;
        position.y = PENALTY_AREA_WIDTH/2.0;
        break;
    case LEFT_PBOX_CENTER_FLAG:
        position.x = -PITCH_LENGTH/2.0+PENALTY_AREA_LENGTH;
        position.y = 0.0;
        break;
    case RIGHT_PBOX_TOP_FLAG:
        position.x = PITCH_LENGTH/2.0-PENALTY_AREA_LENGTH;
        position.y = -PENALTY_AREA_WIDTH/2.0;
        break;
    case RIGHT_PBOX_BOTTOM_FLAG:
        position.x = PITCH_LENGTH/2.0-PENALTY_AREA_LENGTH;
        position.y = PENALTY_AREA_WIDTH/2.0;
        break;
    case RIGHT_PBOX_CENTER_FLAG:
        position.x = PITCH_LENGTH/2.0-PENALTY_AREA_LENGTH;
        position.y = 0.0;
        break;
    // The gajillion and a half new flags
    case CENTER_FLAG:
        position.x = 0.0;
        position.y = 0.0;
        break;
    case LEFT_GOAL_TOP_FLAG:
        position.x = -PITCH_LENGTH/2.0;
        position.y = -GOAL_WIDTH/2.0;
        break;
    case LEFT_GOAL_BOTTOM_FLAG:
        position.x = -PITCH_LENGTH/2.0;
        position.y = GOAL_WIDTH/2.0;
        break;

```

```

case RIGHT_GOAL_TOP_FLAG:
    position.x = PITCH_LENGTH/2.0;
    position.y = -GOAL_WIDTH/2.0;
    break;
case RIGHT_GOAL_BOTTOM_FLAG:
    position.x = PITCH_LENGTH/2.0;
    position.y = GOAL_WIDTH/2.0;
    break;
// The new flags that are outside the field
case OUT_TOP_ZERO_FLAG:
    position.x = 0.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_LEFT_10_FLAG:
    position.x = -10.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_LEFT_20_FLAG:
    position.x = -20.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_LEFT_30_FLAG:
    position.x = -30.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_LEFT_40_FLAG:
    position.x = -40.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_LEFT_50_FLAG:
    position.x = -50.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_RIGHT_10_FLAG:
    position.x = 10.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_RIGHT_20_FLAG:
    position.x = 20.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_RIGHT_30_FLAG:
    position.x = 30.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_RIGHT_40_FLAG:
    position.x = 40.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_TOP_RIGHT_50_FLAG:
    position.x = 50.0;
    position.y = -PITCH_WIDTH/2 - 5.0;
    break;
case OUT_BOTTOM_ZERO_FLAG:
    position.x = 0.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_LEFT_10_FLAG:
    position.x = -10.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_LEFT_20_FLAG:
    position.x = -20.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_LEFT_30_FLAG:
    position.x = -30.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;

```

```

break;
case OUT_BOTTOM_LEFT_40_FLAG:
    position.x = -40.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_LEFT_50_FLAG:
    position.x = -50.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_RIGHT_10_FLAG:
    position.x = 10.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_RIGHT_20_FLAG:
    position.x = 20.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_RIGHT_30_FLAG:
    position.x = 30.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_RIGHT_40_FLAG:
    position.x = 40.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_BOTTOM_RIGHT_50_FLAG:
    position.x = 50.0;
    position.y = PITCH_WIDTH/2.0 + 5.0;
    break;
case OUT_RIGHT_ZERO_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = 0.0;
    break;
case OUT_RIGHT_TOP_10_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = -10.0;
    break;
case OUT_RIGHT_TOP_20_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = -20.0;
    break;
case OUT_RIGHT_TOP_30_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = -30.0;
    break;
case OUT_RIGHT_BOTTOM_10_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = 10.0;
    break;
case OUT_RIGHT_BOTTOM_20_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = 20.0;
    break;
case OUT_RIGHT_BOTTOM_30_FLAG:
    position.x = PITCH_LENGTH/2.0 + 5.0;
    position.y = 30.0;
    break;
case OUT_LEFT_ZERO_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;
    position.y = 0.0;
    break;
case OUT_LEFT_TOP_10_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;
    position.y = -10.0;
    break;
case OUT_LEFT_TOP_20_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;

```

```

    position.y = -20.0;
    break;
case OUT_LEFT_TOP_30_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;
    position.y = -30.0;
    break;
case OUT_LEFT_BOTTOM_10_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;
    position.y = 10.0;
    break;
case OUT_LEFT_BOTTOM_20_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;
    position.y = 20.0;
    break;
case OUT_LEFT_BOTTOM_30_FLAG:
    position.x = -PITCH_LENGTH/2.0 - 5.0;
    position.y = 30.0;
    break;
}
}
// For now outputs only the type
ostream& operator<< (ostream &o, const Flag_obj& f) {
    switch (f.flag type) {
        case TOP_LEFT_FLAG:
            o << "Top Left Corner Flag\t";
            break;
        case TOP_RIGHT_FLAG:
            o << "Top Right Corner Flag\t";
            break;
        case BOTTOM_LEFT_FLAG:
            o << "Bottom Left Corner Flag\t";
            break;
        case BOTTOM_RIGHT_FLAG:
            o << "Bottom Right Corner Flag\t";
            break;
        case TOP_CENTER_FLAG:
            o << "Top Center Flag\t\t";
            break;
        case BOTTOM_CENTER_FLAG:
            o << "Bottom Center Flag\t";
            break;
        case LEFT_PBOX_TOP_FLAG:
            o << "Left Penalty Box Top Flag";
            break;
        case LEFT_PBOX_BOTTOM_FLAG:
            o << "Left Penalty Box Bottom Flag";
            break;
        case LEFT_PBOX_CENTER_FLAG:
            o << "Left Penalty Box Center Flag";
            break;
        case RIGHT_PBOX_TOP_FLAG:
            o << "Right Penalty Box Top Flag";
            break;
        case RIGHT_PBOX_BOTTOM_FLAG:
            o << "Right Penalty Box Bottom Flag";
            break;
        case RIGHT_PBOX_CENTER_FLAG:
            o << "Right Penalty Box Center Flag";
            break;
        // The gajillion and a half new flags
        case CENTER_FLAG:
            o << "Super-Main Center Flag\t";
            break;
        case LEFT_GOAL_TOP_FLAG:
            o << "Left Goal Top Flag\t";
            break;
        case LEFT_GOAL_BOTTOM_FLAG:
            o << "Left Goal Bottom Flag\t";
            break;
        case RIGHT_GOAL_TOP_FLAG:
            o << "Right Goal Top Flag\t";
            break;
        case RIGHT_GOAL_BOTTOM_FLAG:
            o << "Right Goal Bottom Flag\t";
            break;
    }
}

```

```

o << "Right Goal Top Flag\t";
break;
case RIGHT_GOAL_BOTTOM_FLAG:
o << "Right Goal Bottom Flag\t";
break;

// The new flags that are outside the field
case OUT_TOP_ZERO_FLAG:
o << "Outside Top Flag at 0";
break;

case OUT_TOP_LEFT_10_FLAG:
o << "Outside Top Flag at 10 Left";
break;
case OUT_TOP_LEFT_20_FLAG:
o << "Outside Top Flag at 20 Left";
break;
case OUT_TOP_LEFT_30_FLAG:
o << "Outside Top Flag at 30 Left";
break;
case OUT_TOP_LEFT_40_FLAG:
o << "Outside Top Flag at 40 Left";
break;
case OUT_TOP_LEFT_50_FLAG:
o << "Outside Top Flag at 50 Left";
break;

case OUT_TOP_RIGHT_10_FLAG:
o << "Outside Top Flag at 10 Right";
break;
case OUT_TOP_RIGHT_20_FLAG:
o << "Outside Top Flag at 20 Right";
break;
case OUT_TOP_RIGHT_30_FLAG:
o << "Outside Top Flag at 30 Right";
break;
case OUT_TOP_RIGHT_40_FLAG:
o << "Outside Top Flag at 40 Right";
break;
case OUT_TOP_RIGHT_50_FLAG:
o << "Outside Top Flag at 50 Right";
break;

case OUT_BOTTOM_ZERO_FLAG:
o << "Outside Bottom Flag at 0";
break;

case OUT_BOTTOM_LEFT_10_FLAG:
o << "Outside Bottom Flag at 10 Left";
break;
case OUT_BOTTOM_LEFT_20_FLAG:
o << "Outside Bottom Flag at 20 Left";
break;
case OUT_BOTTOM_LEFT_30_FLAG:
o << "Outside Bottom Flag at 30 Left";
break;
case OUT_BOTTOM_LEFT_40_FLAG:
o << "Outside Bottom Flag at 40 Left";
break;
case OUT_BOTTOM_LEFT_50_FLAG:
o << "Outside Bottom Flag at 50 Left";
break;

case OUT_BOTTOM_RIGHT_10_FLAG:
o << "Outside Bottom Flag at 10 Right";
break;
case OUT_BOTTOM_RIGHT_20_FLAG:
o << "Outside Bottom Flag at 20 Right";
break;
case OUT_BOTTOM_RIGHT_30_FLAG:
o << "Outside Bottom Flag at 30 Right";
break;
case OUT_BOTTOM_RIGHT_40_FLAG:

```

```

o << "Outside Bottom Flag at 40 Right";
break;
case OUT_BOTTOM_RIGHT_50_FLAG:
o << "Outside Bottom Flag at 50 Right";
break;

case OUT_RIGHT_ZERO_FLAG:
o << "Outside Right Flag at 0";
break;

case OUT_RIGHT_TOP_10_FLAG:
o << "Outside Right Flag at 10 Top";
break;
case OUT_RIGHT_TOP_20_FLAG:
o << "Outside Right Flag at 20 Top";
break;
case OUT_RIGHT_TOP_30_FLAG:
o << "Outside Right Flag at 30 Top";
break;

case OUT_RIGHT_BOTTOM_10_FLAG:
o << "Outside Right Flag at 10 Bottom";
break;
case OUT_RIGHT_BOTTOM_20_FLAG:
o << "Outside Right Flag at 20 Bottom";
break;
case OUT_RIGHT_BOTTOM_30_FLAG:
o << "Outside Right Flag at 30 Bottom";
break;

case OUT_LEFT_ZERO_FLAG:
o << "Outside Left Flag at 0";
break;

case OUT_LEFT_TOP_10_FLAG:
o << "Outside Left Flag at 10 Top";
break;
case OUT_LEFT_TOP_20_FLAG:
o << "Outside Left Flag at 20 Top";
break;
case OUT_LEFT_TOP_30_FLAG:
o << "Outside Left Flag at 30 Top";
break;

case OUT_LEFT_BOTTOM_10_FLAG:
o << "Outside Left Flag at 10 Bottom";
break;
case OUT_LEFT_BOTTOM_20_FLAG:
o << "Outside Left Flag at 20 Bottom";
break;
case OUT_LEFT_BOTTOM_30_FLAG:
o << "Outside Left Flag at 30 Bottom";
break;
default:
o << "Unknown flag type";
break;
}
return o;
}

// For now outputs only the type
void Flag_obj::fprintf_flag_type(FILE* f) {
switch (flag_type) {
case TOP_LEFT_FLAG:
fprintf(f, "Top Left Corner Flag\t");
break;
case TOP_RIGHT_FLAG:
fprintf(f, "Top Right Corner Flag\t");
break;
case BOTTOM_LEFT_FLAG:
fprintf(f, "Bottom Left Corner Flag\t");
break;

```

```

break;
case BOTTOM_RIGHT_FLAG:
    printf(f, "Bottom Right Corner Flag\t");
    break;
case TOP_CENTER_FLAG:
    printf(f, "Top Center Flag\t\t");
    break;
case BOTTOM_CENTER_FLAG:
    printf(f, "Bottom Center Flag\t");
    break;
case LEFT_PBOX_TOP_FLAG:
    printf(f, "Left Penalty Box Top Flag");
    break;
case LEFT_PBOX_BOTTOM_FLAG:
    printf(f, "Left Penalty Box Bottom Flag");
    break;
case LEFT_PBOX_CENTER_FLAG:
    printf(f, "Left Penalty Box Center Flag");
    break;
case RIGHT_PBOX_TOP_FLAG:
    printf(f, "Right Penalty Box Top Flag");
    break;
case RIGHT_PBOX_BOTTOM_FLAG:
    printf(f, "Right Penalty Box Bottom Flag");
    break;
case RIGHT_PBOX_CENTER_FLAG:
    printf(f, "Right Penalty Box Center Flag");
    break;
// The gajillion and a half new flags
case CENTER_FLAG:
    printf(f, "Super-Main Center Flag\t");
    break;
case LEFT_GOAL_TOP_FLAG:
    printf(f, "Left Goal Top Flag\t");
    break;
case LEFT_GOAL_BOTTOM_FLAG:
    printf(f, "Left Goal Bottom Flag\t");
    break;
case RIGHT_GOAL_TOP_FLAG:
    printf(f, "Right Goal Top Flag\t");
    break;
case RIGHT_GOAL_BOTTOM_FLAG:
    printf(f, "Right Goal Bottom Flag\t");
    break;
// The new flags that are outside the field
case OUT_TOP_ZERO_FLAG:
    printf(f, "Outside Top Flag at 0");
    break;
case OUT_TOP_LEFT_10_FLAG:
    printf(f, "Outside Top Flag at 10 Left");
    break;
case OUT_TOP_LEFT_20_FLAG:
    printf(f, "Outside Top Flag at 20 Left");
    break;
case OUT_TOP_LEFT_30_FLAG:
    printf(f, "Outside Top Flag at 30 Left");
    break;
case OUT_TOP_LEFT_40_FLAG:
    printf(f, "Outside Top Flag at 40 Left");
    break;
case OUT_TOP_LEFT_50_FLAG:
    printf(f, "Outside Top Flag at 50 Left");
    break;
case OUT_TOP_RIGHT_10_FLAG:
    printf(f, "Outside Top Flag at 10 Right");
    break;
case OUT_TOP_RIGHT_20_FLAG:
    printf(f, "Outside Top Flag at 20 Right");
    break;

```

```

case OUT_TOP_RIGHT_30_FLAG:
    printf(f, "Outside Top Flag at 30 Right");
    break;
case OUT_TOP_RIGHT_40_FLAG:
    printf(f, "Outside Top Flag at 40 Right");
    break;
case OUT_TOP_RIGHT_50_FLAG:
    printf(f, "Outside Top Flag at 50 Right");
    break;
case OUT_BOTTOM_ZERO_FLAG:
    printf(f, "Outside Bottom Flag at 0");
    break;
case OUT_BOTTOM_LEFT_10_FLAG:
    printf(f, "Outside Bottom Flag at 10 Left");
    break;
case OUT_BOTTOM_LEFT_20_FLAG:
    printf(f, "Outside Bottom Flag at 20 Left");
    break;
case OUT_BOTTOM_LEFT_30_FLAG:
    printf(f, "Outside Bottom Flag at 30 Left");
    break;
case OUT_BOTTOM_LEFT_40_FLAG:
    printf(f, "Outside Bottom Flag at 40 Left");
    break;
case OUT_BOTTOM_LEFT_50_FLAG:
    printf(f, "Outside Bottom Flag at 50 Left");
    break;
case OUT_BOTTOM_RIGHT_10_FLAG:
    printf(f, "Outside Bottom Flag at 10 Right");
    break;
case OUT_BOTTOM_RIGHT_20_FLAG:
    printf(f, "Outside Bottom Flag at 20 Right");
    break;
case OUT_BOTTOM_RIGHT_30_FLAG:
    printf(f, "Outside Bottom Flag at 30 Right");
    break;
case OUT_BOTTOM_RIGHT_40_FLAG:
    printf(f, "Outside Bottom Flag at 40 Right");
    break;
case OUT_BOTTOM_RIGHT_50_FLAG:
    printf(f, "Outside Bottom Flag at 50 Right");
    break;
case OUT_RIGHT_ZERO_FLAG:
    printf(f, "Outside Right Flag at 0");
    break;
case OUT_RIGHT_TOP_10_FLAG:
    printf(f, "Outside Right Flag at 10 Top");
    break;
case OUT_RIGHT_TOP_20_FLAG:
    printf(f, "Outside Right Flag at 20 Top");
    break;
case OUT_RIGHT_TOP_30_FLAG:
    printf(f, "Outside Right Flag at 30 Top");
    break;
case OUT_RIGHT_BOTTOM_10_FLAG:
    printf(f, "Outside Right Flag at 10 Bottom");
    break;
case OUT_RIGHT_BOTTOM_20_FLAG:
    printf(f, "Outside Right Flag at 20 Bottom");
    break;
case OUT_RIGHT_BOTTOM_30_FLAG:
    printf(f, "Outside Right Flag at 30 Bottom");
    break;
case OUT_LEFT_ZERO_FLAG:
    printf(f, "Outside Left Flag at 0");
    break;

```

```

case OUT_LEFT_TOP_10_FLAG:
    fprintf(f, "Outside Left Flag at 10 Top");
    break;
case OUT_LEFT_TOP_20_FLAG:
    fprintf(f, "Outside Left Flag at 20 Top");
    break;
case OUT_LEFT_TOP_30_FLAG:
    fprintf(f, "Outside Left Flag at 30 Top");
    break;

case OUT_LEFT_BOTTOM_10_FLAG:
    fprintf(f, "Outside Left Flag at 10 Bottom");
    break;
case OUT_LEFT_BOTTOM_20_FLAG:
    fprintf(f, "Outside Left Flag at 20 Bottom");
    break;
case OUT_LEFT_BOTTOM_30_FLAG:
    fprintf(f, "Outside Left Flag at 30 Bottom");
    break;
default:
    fprintf(f, "Unknown flag type");
    break;
}
}
/*
int Flag_obj::is_outside_flag() {
    return
        (( flag_type == OUT_TOP_ZERO_FLAG ) ||
         ( flag_type == OUT_TOP_LEFT_10_FLAG ) ||
         ( flag_type == OUT_TOP_LEFT_20_FLAG ) ||
         ( flag_type == OUT_TOP_LEFT_30_FLAG ) ||
         ( flag_type == OUT_TOP_LEFT_40_FLAG ) ||
         ( flag_type == OUT_TOP_LEFT_50_FLAG ) ||

         ( flag_type == OUT_TOP_RIGHT_10_FLAG ) ||
         ( flag_type == OUT_TOP_RIGHT_20_FLAG ) ||
         ( flag_type == OUT_TOP_RIGHT_30_FLAG ) ||
         ( flag_type == OUT_TOP_RIGHT_40_FLAG ) ||
         ( flag_type == OUT_TOP_RIGHT_50_FLAG ) ||

         ( flag_type == OUT_BOTTOM_ZERO_FLAG ) ||

         ( flag_type == OUT_BOTTOM_LEFT_10_FLAG ) ||
         ( flag_type == OUT_BOTTOM_LEFT_20_FLAG ) ||
         ( flag_type == OUT_BOTTOM_LEFT_30_FLAG ) ||
         ( flag_type == OUT_BOTTOM_LEFT_40_FLAG ) ||
         ( flag_type == OUT_BOTTOM_LEFT_50_FLAG ) ||

         ( flag_type == OUT_BOTTOM_RIGHT_10_FLAG ) ||
         ( flag_type == OUT_BOTTOM_RIGHT_20_FLAG ) ||
         ( flag_type == OUT_BOTTOM_RIGHT_30_FLAG ) ||
         ( flag_type == OUT_BOTTOM_RIGHT_40_FLAG ) ||
         ( flag_type == OUT_BOTTOM_RIGHT_50_FLAG ) ||

         ( flag_type == OUT_RIGHT_ZERO_FLAG ) ||

         ( flag_type == OUT_RIGHT_TOP_10_FLAG ) ||
         ( flag_type == OUT_RIGHT_TOP_20_FLAG ) ||
         ( flag_type == OUT_RIGHT_TOP_30_FLAG ) ||

         ( flag_type == OUT_RIGHT_BOTTOM_10_FLAG ) ||
         ( flag_type == OUT_RIGHT_BOTTOM_20_FLAG ) ||
         ( flag_type == OUT_RIGHT_BOTTOM_30_FLAG ) ||

         ( flag_type == OUT_LEFT_ZERO_FLAG ) ||

         ( flag_type == OUT_LEFT_TOP_10_FLAG ) ||
         ( flag_type == OUT_LEFT_TOP_20_FLAG ) ||
         ( flag_type == OUT_LEFT_TOP_30_FLAG ) ||

         ( flag_type == OUT_LEFT_BOTTOM_10_FLAG ) ||
         ( flag_type == OUT_LEFT_BOTTOM_20_FLAG ) ||
         ( flag_type == OUT_LEFT_BOTTOM_30_FLAG ) ||
    );
}

```

```

    ( flag_type == OUT_LEFT_BOTTOM_30_FLAG ) )
}
*/

```

```
// **C++**
// The above line causes xemacs to open this file in c++-mode
#include <sensor.hh>
/* Note these types
#define SENDER_UNKNOWN -1
#define SENDER_REFEREE 0
#define SENDER_SELF 1
*/

Hear_info::Hear_info(char* msg)
{
    char tmp[MAX_MSG_LENGTH];
    int i;

    dir = 0;

    sscanf(msg, "hear %d %s", &timestamp, tmp);
    msg = next_token(msg + 1);
    msg = next_token(msg);
    msg = next_token(msg);

    for(i = strlen(msg); i >= 0; i--)
        if (msg[i] == ',') {
            msg[i] = '\0';
            break;
        }

    strcpy(message, msg);

    if (!strcmp(tmp, "referee")) {
        sender = SENDER_REFEREE;
    }
    else if (!strcmp(tmp, "self")) {
        sender = SENDER_SELF;
        dir = 0;
    }
    else {
        sender = SENDER_UNKNOWN;
        dir = (int)atof(tmp);
    }
}

void Hear_info::fprintf_hear_info(FILE* f)
{
    fprintf(f, "Hey! I heard down the grapevine at %d that %s\n", timestamp, message);
    fprintf(f, "It came from ");
    switch(sender) {
        case SENDER_REFEREE:
            fprintf(f, "the referee!!!\n");
            break;
        case SENDER_SELF:
            fprintf(f, "myself, though. Hee-hee.\n");
            break;
        case SENDER_UNKNOWN:
            fprintf(f, "Someone at %d degrees\n", dir);
            break;
        default:
            fprintf(f, "Hell knows who!\n");
            break;
    }
}
```

```

#include <libclient.h>
#include <self.hh> // For VIEW_ defines

#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <netdb.h>
#include <errno.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/filio.h>

#define bzero(a, b) memset(a, 0, b)

Bool receive_message(Socket *sock, char *buf, int size)
{
    int n, servlen ;
    struct sockaddr_in serv_addr ;

    servlen = sizeof(serv_addr) ;
    n = recvfrom(sock->sockfd, buf, size, 0,
                 (struct sockaddr *)&serv_addr, &servlen) ;

    if (n < 0) {
        if (errno != ENOTSOCK && errno != EWOULDBLOCK) {
            fprintf(stderr, "Error: recvfrom.\n") ;
            exit(1) ;
        }
        return FALSE ;
    }
    else {
        sock->serv_addr.sin_port = serv_addr.sin_port ;
        buf[n] = '\0' ;
        if (n == 0)
            return FALSE ;
        else
            return TRUE ;
    }
}

PlayMode get_playmode(char *pmodestr)
{
    char *pm_strings[] = PLAYMODE_STRINGS ;
    int pm ; // PlayMode
    for (pm=0; pm_strings[pm] != NULL; pm += 1) {
        if (!strcmp(pmodestr, pm_strings[pm]))
            return (PlayMode)pm ;
    }
    return PM_Error ;
}

InitInfo send_com_init(Socket *sock, char *teamname)
{
    PlayMode get_playmode(char *pmodestr) ;
    InitInfo info ;
    char msg_buf[BUFSIZE], com[16], side ;
    int i ;

    sprintf(msg_buf, "(init %s (version 4.18))", teamname) ;
    if (!send_message(*sock, msg_buf)) {
        info.side = S_UNKNOWN ;
        return info ;
    }

    while(!receive_message(sock, msg_buf, BUFSIZE))
        /* wait for reply */ ;
}

```

```

    sscanf(msg_buf, "%s ", com) ;
    if (!strcmp(com, "init")) {
        sscanf(msg_buf, "(init %c %d %s)",
               &side, &info.unum, info.playmodestr) ;
        if (side == 'l')
            info.side = S_LEFT ;
        else if (side == 'r')
            info.side = S_RIGHT ;
        for(i=0; info.playmodestr[i] != '\0'; i++)
            info.playmodestr[i] = '\0' ;
        info.playmode = get_playmode(info.playmodestr) ;
    }
    else
        info.side = S_UNKNOWN ;

    return info ;
}

InitInfo send_com_reconnect(Socket *sock, char *teamname, int unum)
{
    InitInfo info ;
    char msg_buf[BUFSIZE], com[16], side ;
    int i ;

    sprintf(msg_buf, "(reconnect %s %d)", teamname, unum) ;
    if (!send_message(*sock, msg_buf)) {
        info.side = S_UNKNOWN ;
        return info ;
    }

    while(!receive_message(sock, msg_buf, BUFSIZE))
        /* wait for reply */ ;

    sscanf(msg_buf, "%s ", com) ;
    if (!strcmp(com, "reconnect")) {
        sscanf(msg_buf, "(reconnect %c %s)", &side, info.playmodestr) ;
        if (side == 'l')
            info.side = S_LEFT ;
        else if (side == 'r')
            info.side = S_RIGHT ;
        for(i=0; info.playmodestr[i] != '\0'; i++)
            info.playmodestr[i] = '\0' ;
        info.playmode = get_playmode(info.playmodestr) ;
    }
    else
        info.side = S_UNKNOWN ;

    return info ;
}

Bool send_com_move(Socket *sock, double x, double y)
{
    char msg_buf[BUFSIZE] ;

    sprintf(msg_buf, "(move %lf %lf)\n", x, y) ;
    if (!send_message(*sock, msg_buf))
        return FALSE ;

    return TRUE ;
}

Bool send_com_turn(Socket *sock, double moment)
{
    char msg_buf[BUFSIZE] ;

    sprintf(msg_buf, "(turn %f)\n", moment) ;
    if (!send_message(*sock, msg_buf))
        return FALSE ;

    return TRUE ;
}

```

```

}

Bool send_com_dash(Socket *sock, double power)
{
    char msg_buf[BUFSIZE];
    sprintf(msg_buf, "(dash %f)\n", power);
    if (!send_message(*sock, msg_buf))
        return FALSE;
    return TRUE;
}

Bool send_com_kick(Socket *sock, double power, double dir)
{
    char msg_buf[BUFSIZE];
    sprintf(msg_buf, "(kick %f %f)\n", power, dir);
    if (!send_message(*sock, msg_buf))
        return FALSE;
    return TRUE;
}

Bool send_com_say(Socket *sock, char *message)
{
    char msg_buf[BUFSIZE];
    sprintf(msg_buf, "(say %s)\n", message);
    if (!send_message(*sock, msg_buf))
        return FALSE;
    return TRUE;
}

Bool send_com_catch(Socket *sock, int ang)
{
    char msg_buf[BUFSIZE];
    sprintf(msg_buf, "(catch %f)\n", ang);
    if (!send_message(*sock, msg_buf))
        return FALSE;
    return TRUE;
}

Bool send_com_change_view(Socket *sock, int width, int quality)
{
    char msg_buf[BUFSIZE], w_str[8], q_str[8];
    switch (width) {
    case VIEW_WIDTH_NARROW:
        strcpy(w_str, "narrow");
        break;
    case VIEW_WIDTH_NORMAL:
        strcpy(w_str, "normal");
        break;
    case VIEW_WIDTH_WIDE:
        strcpy(w_str, "wide");
        break;
    default:
        return FALSE;
    }
    switch (quality) {
    case VIEW_QUALITY_HIGH:
        strcpy(q_str, "high");
        break;
    case VIEW_QUALITY_LOW:
        strcpy(q_str, "low");
}

```

```

        break;
    default:
        return FALSE;
    }
    sprintf(msg_buf, "(change_view %s %s)\n", w_str, q_str);
    if (!send_message(*sock, msg_buf))
        return FALSE;
    return TRUE;
}

Socket init_connection(char *host, int port)
{
    struct hostent *host_ent;
    struct in_addr *addr_ptr;
    struct sockaddr_in cli_addr;
    int sockfd, one;
    Socket sock;
    sock.socketfd = -1;
    if ((host_ent = (struct hostent *)gethostbyname(host)) == NULL) {
        /* Check if a numeric address */
        if (inet_addr(host) == 0)
            return sock;
    }
    else {
        addr_ptr = (struct in_addr *)host_ent->h_addr_list;
        host = inet_ntoa(*addr_ptr);
    }
    /* Open UDP socket. */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        fprintf(stderr, "Can't open socket.\n");
        return sock;
    }
    one = 1;
    if (ioctl(sockfd, FIONBIO, &one) < 0) {
        fprintf(stderr, "Can't ioctl on socket.\n");
        return sock;
    }
    /* Bind any local address. */
    bzero((char *) &cli_addr, sizeof(cli_addr));
    cli_addr.sin_family = AF_INET;
    cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    cli_addr.sin_port = htons(0);
    if (bind(sockfd, (struct sockaddr *) &cli_addr,
        sizeof(cli_addr)) < 0) {
        fprintf(stderr, "Can't bind local address.\n");
        return sock;
    }
    /* Fill in the structure with the address of the server. */
    sock.socketfd = sockfd;
    bzero((char *) &sock.serv_addr, sizeof(sock.serv_addr));
    sock.serv_addr.sin_family = AF_INET;
    sock.serv_addr.sin_addr.s_addr = inet_addr(host);
    sock.serv_addr.sin_port = htons(port);
    return sock;
}

Bool send_com_sense(Socket *sock) {
    // Just execute some Noda-ish code. There's absolutely
    // no reason to take the world parameter, but I take it anyway.
    char msg_buf[BUFSIZE];
}

```

```
    sprintf(msg_buf, "(sense_body)\n");
    if (!send_message(*sock, msg_buf)) { // Master Noda's bracketing sucked
    }
    return FALSE ;
}
return TRUE ;
}

Bool send_message(Socket sock, char *buf)
{
    int n ;
    n = strlen(buf) ;
    if (sendto(sock.socketfd, buf, n, 0,
               (struct sockaddr *)&sock.serv_addr,
               sizeof(sock.serv_addr)) != n)
        return FALSE ;
    else
        return TRUE ;
}

void close_connection(Socket sock)
{
    close (sock.socketfd) ;
}
```

```

// #define MOVETO_DEBUG MODE
// #define MOVETO_DEBUG MODE
// #define CAREFUL_MOVE_DEBUG MODE

#include <roboocup_act.hh>
#include <trig.hh>
#include <assert.h>
#include <roboocup_com.hh>

#define BACKWARDS_TURN_GRACE_AMOUNT 4
#define CAREFUL_MOVE_WAITS 3

#define COLLISION_TEST_RADIUS 5
#define COLLISION_LOOKAHEAD 2

/* ***** */
/* ***** */
Move_to::Move_to () {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    s = NULL;
    world = NULL;
    backwards = 0;
}

void Move_to::compute_tangential_displacement () {
    double langle; // local variable
    langle = displacement.angle() + P5PI;

    tangent_disp.x = cos (langle); // unit vector in direction orthogonal to displace
    tangent_disp.y = sin (langle); // magnitude of velocity in this or
    thogonal direction
    double vel_90 = tangent_disp.dot (s->velocity); // compute tangential displa
    cement due to current velocity */
    tangent_disp *= vel_90 * PLAYER_DIST_FROM_VEL_FACTOR;

#ifdef MOVETO_DEBUG MODE
    fprintf (world->output_file, "\tLast pos: (%lf, %lf), Pos now (%lf, %lf)\n", D_last
    .pos.x, D_last.pos.y, s->position.x, s->position.y);
    fprintf (world->output_file, "Velocity, Data: (r:%lf, theta:%lf)\n", s->velocity.r
    (), s->velocity.angle() * RAD2DEG);
#endif

#ifdef MOVETO_DEBUG MODE
    fprintf (world->output_file, "MTA: tangential displacement is: (%lf, %lf)\n", tang
    ent_disp.x, tangent_disp.y);
#endif
}

Move_to::Move_to (World *w, Self_obj *me,
    Pair dest, double max_power,
    double tol_dist, int less_wasteful_dashes, int dest_might_change) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    s = NULL;
    world = w;
    backwards = 0;
}

void Move_to::Move_to (World *w, Self_obj *s, Pair dest, double max_power, double tol_dist,
    int less_wasteful_dashes,
    int dest_might_change, int watch_ball) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = s;
    this->world = w;

    this->less_wasteful_dashes = less_wasteful_dashes;
    this->dest_might_change = dest_might_change;
    this->watch_ball_too = watch_ball;

    backwards_turn_round = 1; // BACKWARDS_TURN_GRACE_AMOUNT;
    if (w->ball->timestamp == w->last_vis_info.time)
        backwards = decide_backwards_when_seeing_ball (w->ball->position, s->facing * DE
        G2RAD);
    else
        backwards = 0;

    fill (w, s, dest, max_power, tol_dist);
}

Move_to::Move_to (World *w, Self_obj *s, Pair dest, double max_power, double tol_dis
    t) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = s;
    this->world = w;

    backwards = 0;

    fill (w, s, dest, max_power, tol_dist);

    Move_to::Move_to (Move_to *mt) {
        next = mt->next;
        prev = mt->prev;
        for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
            formation_positions[i] = mt->formation_positions[i];
            position_taken[i] = mt->position_taken[i];
        }
        formation_positions.set = mt->formation_positions_set;
        destination_idx = mt->destination_idx;
        Pair dest = mt->destination;
        displacement = mt->displacement;
        tangent_disp = mt->tangent_disp;
        disp_minus_tangent = mt->disp_minus_tangent;
    }
}

```

```

this->less_wasteful_dashes = less_wasteful_dashes;
this->dest_might_change = dest_might_change;

backwards = 0;
backwards_turn_round = 2;
watch_ball_too = 0;
c_lists[0]->insert_first(new Command_wait());
destination = dest;
displacement = dest - s->position;
tolerance = tol_dist;
this->max_power = (int)max_power;

for (int i = 0; i < PLAYERS_IN_FORMATION ; i++) {
    position_taken[i] = FALSE;
}
formation_positions_set = FALSE;
return;
fill (w, s, dest, max_power, tol_dist);
}

Move_to::Move_to (World *w, Self_obj *s, Pair dest, double max_power, double tol_dist,
    int less_wasteful_dashes,
    int dest_might_change, int watch_ball) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = s;
    this->world = w;

    this->less_wasteful_dashes = less_wasteful_dashes;
    this->dest_might_change = dest_might_change;
    this->watch_ball_too = watch_ball;

    backwards_turn_round = 1; // BACKWARDS_TURN_GRACE_AMOUNT;
    if (w->ball->timestamp == w->last_vis_info.time)
        backwards = decide_backwards_when_seeing_ball (w->ball->position, s->facing * DE
        G2RAD);
    else
        backwards = 0;

    fill (w, s, dest, max_power, tol_dist);
}

Move_to::Move_to (World *w, Self_obj *s, Pair dest, double max_power, double tol_dis
    t) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = s;
    this->world = w;

    backwards = 0;

    fill (w, s, dest, max_power, tol_dist);

    Move_to::Move_to (Move_to *mt) {
        next = mt->next;
        prev = mt->prev;
        for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
            formation_positions[i] = mt->formation_positions[i];
            position_taken[i] = mt->position_taken[i];
        }
        formation_positions.set = mt->formation_positions_set;
        destination_idx = mt->destination_idx;
        Pair dest = mt->destination;
        displacement = mt->displacement;
        tangent_disp = mt->tangent_disp;
        disp_minus_tangent = mt->disp_minus_tangent;
    }
}

```

```

backwards = mt->backwards;
backwards_turn_round = mt->backwards_turn_round;
watch_ball_too = mt->watch_ball_too;
max_power = mt->max_power;
}
Move_to(mt->world, mt->s, dest, mt->max_power, mt->tolerance);
}
int Move_to::detect_collision_pt (Pair &pos1, Pair &pos2) {
    Pair d = pos2 - pos1;
    return (d.r() <= (2* PLAYER_SIZE) );
}
/* detect_collision_one_obj - sees if two objects, both moving at a *sustained* velo
city, will collide within COLLISION_LOOKAHEAD rounds */
int Move_to::detect_collision_one_obj (Pair colls_pos, Pair colls_vel, Pair self_pos
, Pair self_vel,
                                double self_speed, double self_faci
ngr) {
    int collision = 0;
    int i;
    for (i=0; i < COLLISION_LOOKAHEAD; i++) {
        if (colls_vel.r() > colls_vel.error)
            colls_pos += colls_vel;
        self_vel += Polar2Pair (self_speed, self_facingr);
        if (self_vel.r() > PLAYER_SPEED_MAX)
            self_vel *= (PLAYER_SPEED_MAX / self_vel.r());
        self_pos += self_vel;
        collision &= detect_collision_pt (self_pos, colls_pos);
        self_vel *= PLAYER_DECAV;
    }
    return collision;
}
int Move_to::detect_collision (World *world, Pair self_pos, Pair self_vel, double se
lf_speed, double facingr) {
    assert(0);
    Player_obj *test_coll;
    int collision = 0;
    world = Global_world;
    for (int i = 0; i < 44; i++) {
        if (i < 11)
            test_coll = world->teammates[i];
        else if (i < 22)
            test_coll = world->opponents[i-11];
        else
            test_coll = world->unknowns[i-22];
        if ((test_coll != NULL) && (test_coll->is_not_dead)) {
            if ((test_coll->timestamp == world->last_vis_info_time) && (test_coll->positio
n - self_pos).r() < COLLISION_TEST_RADIUS) {
                collision &= detect_collision_one_obj (test_coll->position, test_coll->veloc
ity, self_pos, self_vel, self_speed, facingr);
            }
        }
    }
    return collision;
}
/* run backwards to see ball - only has effect is watch_ball is true
if see ball: set saw ball to 1
if don't see ball and saw ball before: change direction and set saw ball to 0
if don't see ball and didn't see ball before: don't change direction yet, give
player time to turn, and set saw ball
to 0

```

```

*/
int Move_to::run_backwards_to_see_ball (int watch_ball) {
    // assert(0);
    int retval = backwards;
    if (watch_ball) {
        if (!world->see_ball_now()) {
            #ifdef CAREFUL_USE_WAITS
                if (TRUE) {
                    #else
                        if (backwards_turn_round == 0) {
                            #endif
                            retval = (backwards) ? 0 : 1;
                        }
                    #ifdef CAREFUL_USE_WAITS
                        for (int i = 0; i < CAREFUL_MOVE_WAITS; i++)
                            c_lists[0]->insert_as_next(new Command_null());
                    #else
                        backwards_turn_round = BACKWARDS_TURN_GRACE_AMOUNT;
                    #endif
                }
            #ifdef CAREFUL_MOVE_DEBUG_MODE
                fprintf (world->output_file, "(Round %d) Floating move: don't see ball, chan
ging direction, old backwards=%d, new %d, facing=%d\n",
                    world->time_now, backwards, retval, world->me->facing);
            #endif
        } else {
            backwards_turn_round -= 1;
            #ifdef CAREFUL_MOVE_DEBUG_MODE
                fprintf (world->output_file, "(Round %d) Floating move: don't see ball yet,
give turn %d visinfos to find it, backwards = %d, facing=%d\n",
                    world->time_now, backwards_turn_round, backwards, world->me->facing
                );
            #endif
        }
    } else {
        retval = decide_backwards_when_seeing_ball (world->ball->position, s->facing
        * DEG2RAD);
    }
    #ifdef CAREFUL_MOVE_DEBUG_MODE
        fprintf (world->output_file, "(Round %d) Floating move: see ball (old backwa
rds = %d, new = %d), facing = %d\n", world->time_now,
            backwards, retval, world->me->facing);
    #endif
    backwards_turn_round = 0;
}
return retval;
}
int Move_to::decide_backwards_when_seeing_ball (Pair &ball_pos, double facingr) {
    double player2ball_angle = (ball_pos - world->me->position).angle();
    double disp_angle = disp_minus_tangent.angle();
    return (abs(normalize_rad (disp_angle - player2ball_angle)) > P5PI);
}
int Move_to::okay_run_without_turning (double facingr, Pair &self_pos, Pair &self_ve
l) {
    int rounds_now;
    int rounds_in_future;

```

```

rounds_now = s->rounds_to_point(&destination, tolerance);
Pair unit_facing(cos (facingr), sin (facingr));
Pair this_round_vel;
double this_round_speed;
Pair next_round_pos;

const int ROUNDS_TO_POINT_LOOKAHEAD = 4;

this_round_vel = self_vel;
for (int i = 0; i < ROUNDS_TO_POINT_LOOKAHEAD; i++) {
    this_round_vel += unit_facing * (compute_best_power (unit_facing.dot (self_vel),
max_power) * POWERRATE);
    this_round_speed = this_round_vel.r();
    if (this_round_speed > PLAYER_SPEED_MAX)
        this_round_vel *= PLAYER_SPEED_MAX / this_round_speed;
    next_round_pos = self_pos + this_round_vel;
    this_round_vel *= PLAYER_DECAY;
}

rounds_in_future = s->hypothetical_rounds_to_point (&destination, tolerance, &next_
_round_pos, &this_round_vel,
    facingr * RAD2DEG, max_power);

#ifdef MOVE_TO_DEBUG_MODE
    fprintf (world->output_file, "Making turn - rounds to dest now: %d, next turn
: %d, velocity: (%f, %f)\n",
        rounds_now, rounds_next_turn, this_round_vel.x, this_round_vel.y);
#endif

return (rounds_in_future <= (rounds_now - ROUNDS_TO_POINT_LOOKAHEAD));
}

if 1
void Move_to::make_imm_turn (double dist_remaining) {
double sweep_tolerance; //amount we can be off and still hit the circle
double turn_angle; // angle we need to turn
double needed_turn_angle; // angle we need to turn with due to our vel
int rounds_now = 0;
int rounds_next_turn = 0;
double facingr = s->facing * DEG2RAD;

int need_to_check_another_turn;
int rounds_says_to_turn;
Pair self_vel = s->velocity;
Pair self_pos = s->position;
int first_turn = 1;
Command_obj *insertion_pt = c_lists[0]->get_next();
/*
if (detect_collision (world, s->position, s->velocity,
    compute_runatspeed from maxpower (max_power), facingr))
    //fprintf (world->output_file, "(Round %d) Collision imminent", world->time_now
);
printf("(Round %d) Collision imminent", world->time_now);
*/
need_to_check_another_turn = missing_turn = !compute_needed_angle (&sweep_toleranc
e, &turn_angle);

while (need_to_check_another_turn) {
    turn_angle = normalize_rad (turn_angle);
    // if ((abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= sweep_tolerance) ||
    // (abs (turn_angle) >= PI*25PI) || dist_remaining < 5) {
        rounds_says_to_turn = 0;
        if (((dest_might_change) || (abs (turn_angle * MAX_TURN_ERROR_PERCENT) > sweep_t
olerance))
            rounds_says_to_turn = (tokay_run_without_turning(facingr, self_pos, self_vel)
);
    if ((rounds_says_to_turn) ||
        ((!dest_might_change) && (abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= swee

```

```

p_tolerance) ||
    (abs (turn_angle) >= P5PI) )
{
    /* we want to turn now - we're outside of tolerance and either if we turn no
we won't miss, or
if we don't turn now our tangential facing makes us lose a round later */
    needed_turn_angle = s->compute_needed_angle_at_speed (turn_angle, self_vel
.r());
    if (abs(needed_turn_angle) > PI) {
        if (first_turn) {
            c_lists[0]->insert_as_next (new Command_turn(world,
                Sign(needed_turn_angle) *
180.0));
            first_turn = 0;
        } else {
            c_lists[0]->insert_before(insertion_pt,
                new Command_turn (world,
                    Sign(needed_turn_angle) *
180.0));
        }
        /* update new self data for next run */
        turn_angle -= s->compute_turned_angle_at_speed(Sign(needed_turn_angle) *
PI, self_vel.r());
        self_pos += self_vel;
        self_vel *= PLAYER_DECAY;
        need_to_check_another_turn = missing_turn = 1;
    } else { /* needed turn angle is possible at this speed */
        if (first_turn) {
            c_lists[0]->insert_as_next (new Command_turn(world,
                needed_turn_angle*RAD2DEG
            ) );
            first_turn = 0;
        } else {
            c_lists[0]->insert_before(insertion_pt,
                new Command_turn (world, needed_turn_angle*
RAD2DEG) );
        }
        missing_turn = FALSE;
        need_to_check_another_turn = 0;
    }
#ifdef MOVE_TO_DEBUG_MODE
        fprintf (Global_world->output_file,
            "Moveto - Enqueueing turn, Round %d, Angle, %f, facing %d, \n\
tdesired angle %f, destination (%f, %f)\n",
            global_world->time_now, needed_turn_angle * RAD2DEG, facingr * RA
D2DEG, turn_angle * RAD2DEG, destination.x, destination.y);
#endif
    } else { /* we've decided not to turn right now */
        missing_turn = 1;
        need_to_check_another_turn = 0;
    } /* end while */
}

void Move_to::make_imm_turn (double dist_remaining) {
double sweep_tolerance; //amount we can be off and still hit the circle
double turn_angle; // angle we need to turn

```

```

double needed_turn_angle; // angle we need to turn with due to our vel
int rounds_now = 0;
int rounds_next_turn = 0;
double facingr = s->facing * DEG2RAD;

if (!compute_needed_angle (&sweep_tolerance, &turn_angle)) {
    turn_angle = normalize_rad (turn_angle);
    // if ((abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= sweep_tolerance) ||
    // (abs (turn_angle) >= PI*25PI) || dist_remaining < 5) {
    if ((dest_might_change) || (abs (turn_angle * MAX_TURN_ERROR_PERCENT) > sweep_t
olerance)) {
        rounds_now = s->rounds_to_point (&destination, tolerance);
        Pair unit_facing(cos (facingr), sin (facingr));
        Pair this_round_vel = s->velocity + unit_facing * (compute_best_power (unit_f
acing.dot (s->velocity), max_power) * POWERRATE);
        double this_round_speed = this_round_vel.r();
        if (this_round_speed > PLAYER_SPEED_MAX)
            this_round_vel *= PLAYER_SPEED_MAX / this_round_speed;
        Pair next_round_pos = s->position + this_round_vel;
        this_round_vel *= PLAYER_DECAV;
        rounds_next_turn = s->hypothetical_rounds_to_point (&destination, tolerance,
&next_round_pos, &this_round_vel,
        facingr * RAD2DEG, max_po
wer);
    }
}
#ifdef MOVE_TO_DEBUG_MODE
fprintf (world->output_file, "Making turn - rounds to dest now: %d, next turn
: %d, velocity: (%lf, %lf)\n",
        rounds_now, rounds_next_turn, this_round_vel.x, this_round_vel.y);
#endif
// rounds_now = rounds_next_turn - 1 + (turn_angle >= PI*25PI);
}
if ((rounds_next_turn >= rounds_now) ||
    (((dest_might_change) && (abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= swee
p_tolerance)) ||
    {
        (abs (turn_angle) >= P5PI) )
    }
    /* we want to turn now - we're outside of tolerance and either if we turn no
w won't miss, or
    if we don't turn now our tangential facing makes us lose a round later */
    needed_turn_angle = s->compute_needed_angle_at_speed (turn_angle, s->velocit
y.r());
if (abs (needed_turn_angle) > PI) {
#ifdef MOVE_TO_DEBUG_MODE
    fprintf (Global_world->output_file,
            "Moveto - Enqueueing turn (will need 2 - speed: %lf), Round: %d, facing %
d, angle to input %lf\n", desired_angle %lf, destination (%lf, %lf)\n",
            d_turn_angle * RAD2DEG, turn_angle * RAD2DEG, destination.x, destination.y);
#endif
    c_lists[0]->insert_as_next (new Command_turn (world,
        Sign(needed_turn_angle) * PI
        *RAD2DEG));
        missing_turn = 1;
    }
    else { /* needed turn angle is possible at this speed */
        c_lists[0]->insert_as_next (new Command_turn (world, needed_turn_angle * RAD2
DEG ));
        missing_turn = FALSE;
    }
#ifdef MOVE_TO_DEBUG_MODE
    fprintf (Global_world->output_file,
            "Moveto - Enqueueing turn, Round %d, Angle, %lf, facing %d, \n",
            esired_angle %lf, destination (%lf, %lf)\n",

```

```

        global_world->time_now, needed_turn_angle * RAD2DEG, facingr * RA
D2DEG, turn_angle * RAD2DEG, destination.x, destination.y);
#endif
    } /* if we need to do the turn now */
    else
        missing_turn = 1; // we will need to turn, but for now dashing is the more
worthwhile action
    } else {
        missing_turn = 0;
    }
}
#endif
double Move_to::current_speed_for_dashes (Pair &velocity, double facing) {
    double retval;
    Pair unit_facing(cos (facing), sin (facing));
    double radial_speed = (missing_turn) ? unit_facing.dot (s->velocity)
        : disp_minus_tangent.dot (velocity) / (displacement.r());
    retval = (radial_speed < 0) ? radial_speed : velocity.r();
    return retval;
}
void Move_to::fill (World *w, Self_obj *s, Pair dest, double max_power, double tol_d
ist) {
    destination = dest;
    displacement = dest - s->position;
    // cerr << "Player " << world->me->su_number << ": Fill called. Dest is: " << dest
ination << endl;
    tolerance = tol_dist;
    this->max_power = (int)max_power;
    double sweep_tolerance, turn_angle;
    double dist_remaining;
    double dash_power;
    double radial_speed;
    missing_turn = 0;
    double needed_turn_angle;
    compute_tangential_displacement(); //need to set regardless of whether action is a
lready finished
    backwards = run_backwards_to_see_ball (watch_ball_too);
    dist_remaining = disp_minus_tangent.r();
    if (dist_remaining > tolerance)
        make_imm_turn(dist_remaining);
    /* make dash commands */
    radial_speed = current_speed_for_dashes (s->velocity, s->facing * DEG2RAD);
    compute_needed_dashes (&dist_remaining, radial_speed, max_power);
    /* take disp_minus_tangent.r() - dist_remaining = dist_travelled
    then est_destination = dist_travelled * unit vector in direction of disp_minus_
tangent = */
    est_destination = (1.0 - dist_remaining / disp_minus_tangent.r()) * disp_minus_tan
gent + s->position;
    // sprintf (name, "Moveto (%G, %G) w/ tol %G", destination.x, destination.y, toler
ance);
}
double Move_to::compute_runatspeed_from_maxpower (double power) {
    double run_at_speed;
    if (backwards) {

```

```

power = min (30, power);
run_at_speed = power * POWERRATE * PLAYER_DIST_FROM_VEL_FACTOR;
} else {
power = min (60, power);
run_at_speed = power * POWERRATE * PLAYER_DIST_FROM_VEL_FACTOR;
if (less wasteful dashes)
run_at_speed = min (run_at_speed, 0.95 * PLAYER_SPEED_MAX);
return run_at_speed;
}

double Move_to::compute_best_power (double current_speed, double max_power) {
double dash_power;

/* run_at_speed is the speed that is the speed we'll be running if we run at max_pow
er for an infinite amount of time. we will dash with a higher power to jump up to
this value if we are running slowly now */

double lmax_power;
double run_at_speed;

run_at_speed = compute_runatspeed_from_maxpower (max_power);
if (backwards) {
dash_power = min ((run_at_speed - current_speed) / POWERRATE, 30); //can't get u
p to speed in one dash if we run backwards
dash_power = max (0, dash_power); //if we're going faster than our run at speed
dash_power *= -1; //make it negative if we're going backwards
}
else {
dash_power = (run_at_speed - current_speed) / POWERRATE; //min( (run_at_speed -
current_speed) / POWERRATE, 100.0);
dash_power = max (dash_power, 0); //make dash power positive
}
return (dash_power);
}

Command_obj * Move_to::recompute_existing_dashes (double *dist_remaining, double *ra
dial_speed, double *max_power) {
Command_obj *temp = c_lists[0]->get_next();
double dash_power;

/* figure out how far our existing dashes go */
while ((temp != NULL) && (*dist_remaining > tolerance)) {
if (temp->command_type() == COMMAND_DASH) {
dash_power = compute_best_power (*radial_speed, max_power);
if (dash_power != ((Command_dash*)temp)->get_power()) {
#ifdef MOVE_TO_DEBUG_MODE
fprintf (world->output file, "Move_to: needed to increase dash power from
%lf to %lf\n", temp->arg_a, dash_power);
#endif
((Command_dash*)temp)->set_power (dash_power);
}
*radial_speed += (((Command_dash*)temp)->get_power() * POWERRATE) * ((backwa
rds) ? -1 : 1);
}
*dist_remaining -= *radial_speed;
#ifdef DEBUG_MODE
assert(temp != temp->next);
#endif
*radial_speed *= PLAYER_DECAY;
temp = ((Command_dash*)temp)->next;
}
return temp;
}

```

```

void Move_to::delete_remaining_commands (Command_obj *first_to_go) {
Command_obj *temp = first_to_go;
Command_obj *mark;

/* if there are still commands in the list here, then that means we're dashing t
oo far
remove those commands now */
while (temp != NULL) {
mark = temp->next;
#ifdef MOVE_TO_DEBUG_MODE
world->debug ("Move_to: deleting a dash in update\n");
#endif
c_lists[0]->delete_command (temp);
temp = mark;
}

void Move_to::compute_needed_dashes (double *dist_remaining, double radial_speed, do
uble max_power) {
double dash_power;

while (*dist_remaining > tolerance) {
dash_power = compute_best_power(radial_speed, this->max_power);
c_lists[0]-insert_last (new Command_dash (world, dash_power));
radial_speed += (dash_power * POWERRATE) * ((backwards) ? -1 : 1);
*dist_remaining -= radial_speed;
radial_speed *= PLAYER_DECAY;
}
}

char Move_to::compute_needed_angle (double *sweep_tolerance, double *turn_angle) {
double facingr = s->facing * DEG2RAD;
double ldistance; //local variable
double want_angle = disp_minus_tangent.angle();
// double player2ball_angle;

if (backwards) { // reversing direction - so turn towards the ball
//player2ball_angle = normalize_rad( (world->ball->position - s->position).angle
());
}
want_angle = want_angle + PI; // * Sign(player2ball_angle);

*turn_angle = want_angle - facingr;
*turn_angle = normalize_rad(*turn_angle);

#ifdef CAREFUL_MOVE_DEBUG_MODE
fprintf (world->output file, "Round %d) Float move, have to make an about face
of %lf (backwards = %d)\n", world->time_now, *turn_angle * RAD2DEG, backwards);
#endif

/* find length of altitude from displacement minus tangent point to current facing
trajectory */
ldistance = disp_minus_tangent.r();
double miss_distance = (ldistance * sin (*turn_angle));
miss_distance = abs (miss_distance);

/* given maximum allowable altitude, find angle at which is occurs
sweep is the same for positive or negative, since error is the same percentage
regardless of which direction we want to face */
*sweep_tolerance = asin (tolerance / ldistance);
return ((abs (*turn_angle) >= P5PI) ? 0 : (miss_distance <= tolerance)); //alway
s miss if we've going in the wrong direction
}

int Move_to::update (void) {

```



```

// #define MOVETO_DEBUG MODE
// #define MOVETO_DEBUG MODE
// #define CAREFUL_MOVE_DEBUG MODE

#include <roboocup_act.hh>
#include <trig.hh>
#include <assert.h>
#include <roboocup_com.hh>

#define BACKWARDS_TURN_GRACE_AMOUNT 4
#define CAREFUL_MOVE_WAITS 3

#define COLLISION_TEST_RADIUS 5
#define COLLISION_LOOKAHEAD 2

/* ***** */
/* ***** */
Move_to::Move_to () {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    s=NULL;
    world = NULL;
    backwards = 0;
}

void Move_to::compute_tangential_displacement () {
    double langle; // local variable
    langle = displacement.angle() + P5PI;
    tangent_disp.x = cos (langle); // unit vector in direction orthogonal to displace
    tangent_disp.y = sin (langle); // magnitude of velocity in this or
    ment
    double vel_90 = tangent_disp.dot (s->velocity); // magnitude of velocity in this or
    thogonal direction
    tangent_disp *= vel_90 * PLAYER_DIST_FROM_VEL_FACTOR; // compute tangential displa
    cement due to current velocity */

#ifdef MOVETO_DEBUG MODE
    fprintf (world->output_file, "\tLast pos: (%lf, %lf), Pos now (%lf, %lf)\n", D_las
    t_pos.x, D_last_pos.y, s->position.x, s->position.y);
    fprintf (world->output_file, "Velocity, Data: (r:%lf, theta:%lf)\n", s->velocity.r
    (), s->velocity.angle() * RAD2DEG);
#endif

#ifdef MOVETO_DEBUG MODE
    fprintf (world->output_file, "MTA: tangential displacement is: (%lf, %lf)\n", tang
    ent_disp.x, tangent_disp.y);
#endif
}

#ifdef MOVETO_DEBUG MODE
    fprintf (world->output_file, "MTA: computing angle to turn, disp_minus_tangent: (r
    :%lf, theta:%lf, our facing: %d)\n",
    disp_minus_tangent.r(), disp_minus_tangent.angle() * RAD2DEG, s->facing);
#endif

/* aim to offset the vec_90 distance that our current velocity brings us off course
*/
disp_minus_tangent = displacement - tangent_disp;
}

Move_to::Move_to(World *w, Self_obj *me,
                Pair dest, double max_power,
                double tol_dist, int less_wasteful_dashes, int dest_might_change) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = me;
    this->world = w;
}

```

```

this->less_wasteful_dashes = less_wasteful_dashes;
this->dest_might_change = dest_might_change;

backwards = 0;
backwards_turn_round = 2;
watch_ball_too = 0;
// c_lists[0]->insert_first(new Command_wait());
destination = dest;
displacement = dest - s->position;
tolerance = tol_dist;
this->max_power = (int)max_power;

for (int i = 0; i < PLAYERS_IN_FORMATION ; i++) {
    position_taken[i] = FALSE;
}
formation_positions_set = FALSE;
return;
fill (w, s, dest, max_power, tol_dist);
}

Move_to::Move_to(World *w, Self_obj *s, Pair dest, double max_power, double tol_dist
, int less_wasteful_dashes, int dest_might_change, int watch_ball) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = s;
    this->world = w;

    this->less_wasteful_dashes = less_wasteful_dashes;
    this->dest_might_change = dest_might_change;
    this->watch_ball_too = watch_ball;

    backwards_turn_round = 1; // BACKWARDS_TURN_GRACE_AMOUNT;
    if (w->ball->timestamp == w->last_vis_info.time)
        backwards = decide_backwards_when_seeing_ball (w->ball->position, s->facing * DE
        G2RAD);
    else
        backwards = 0;

    fill (w, s, dest, max_power, tol_dist);
}

Move_to::Move_to (World *w, Self_obj *s, Pair dest, double max_power, double tol_dis
t) {
    // action_type = MOVETO_ACTION;
    next = prev = NULL;
    this->s = s;
    this->world = w;

    backwards = 0;
    backwards_turn_round = BACKWARDS_TURN_GRACE_AMOUNT;
    this->less_wasteful_dashes = 0;
    this->dest_might_change = 0;

    fill (w, s, dest, max_power, tol_dist);
}

Move_to::Move_to (Move_to *mt) {
    next = mt->next;
    prev = mt->prev;
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        formation_positions[i] = mt->formation_positions[i];
        position_taken[i] = mt->position_taken[i];
    }
    formation_positions_set = mt->formation_positions_set;
    destination_idx = mt->destination_idx;
    Pair dest = mt->destination;
    displacement = mt->displacement;
    tangent_disp = mt->tangent_disp;
    disp_minus_tangent = mt->disp_minus_tangent;
}

```

```

backwards = mt->backwards;
backwards_turn_round = mt->backwards_turn_round;
watch_ball_too = mt->watch_ball_too;
max_power = mt->max_power;
}
Move_to(mt->world, mt->s, dest, mt->max_power, mt->tolerance);
}
int Move_to::detect_collision_pt (Pair &pos1, Pair &pos2) {
    Pair d = pos2 - pos1;
    return (d.r() <= (2* PLAYER_SIZE) );
}
/* detect_collision_one_obj - sees if two objects, both moving at a *sustained* velo
city, will collide within COLLISION_LOOKAHEAD rounds */
int Move_to::detect_collision_one_obj (Pair colls_pos, Pair colls_vel, Pair self_pos
, Pair self_vel,
                                double self_speed, double self_fac
ngr) {
    int collision = 0;
    int i;
    for (i=0; i < COLLISION_LOOKAHEAD; i++) {
        if (colls_vel.r() > colls_vel.error)
            colls_pos += colls_vel;
        self_vel += Polar2Pair (self_speed, self_facingr);
        if (self_vel.r() > PLAYER_SPEED_MAX)
            self_vel *= (PLAYER_SPEED_MAX / self_vel.r());
        self_pos += self_vel;
        collision &= detect_collision_pt (self_pos, colls_pos);
        self_vel *= PLAYER_DECAV;
    }
    return collision;
}
int Move_to::detect_collision (World *world, Pair self_pos, Pair self_vel, double se
lf_speed, double facingr) {
    assert(0);
    Player_obj *test_coll;
    int collision = 0;
    world = Global_world;
    for (int i = 0; i < 44; i++) {
        if (i < 11)
            test_coll = world->teammates[i];
        else if (i < 22)
            test_coll = world->opponents[i-11];
        else
            test_coll = world->unknowns[i-22];
        if ((test_coll != NULL) && (test_coll->is_not_dead)) {
            if ((test_coll->timestamp == world->last_vis_info.time) && (test_coll->positio
n - self_pos).r() < COLLISION_TEST_RADIUS) {
                collision &= detect_collision_one_obj (test_coll->position, test_coll->veloc
ity, self_pos, self_vel, self_speed, facingr);
            }
        }
    }
    return collision;
}
/* run backwards to see ball - only has effect is watch_ball is true
if see ball: set saw ball to 1
if don't see ball and saw ball before: change direction and set saw ball to 0
if don't see ball and didn't see ball before: don't change direction yet, give
player time to turn, and set saw ball
to 0

```

```

*/
int Move_to::run_backwards_to_see_ball (int watch_ball) {
    // assert(0);
    int retval = backwards;
    if (watch_ball) {
        if (!world->see_ball_now()) {
            #ifdef CAREFUL_USE_WAITS
                if (TRUE) {
                    #else
                        if (backwards_turn_round == 0) {
                            #endif
                                retval = (backwards) ? 0 : 1;
                            #ifdef CAREFUL_USE_WAITS
                                for (int i = 0; i < CAREFUL_MOVE_WAITS; i++)
                                    c_lists[0]->insert_as_next(new Command_null());
                                #else
                                    backwards_turn_round = BACKWARDS_TURN_GRACE_AMOUNT;
                                #endif
                            #ifdef CAREFUL_MOVE_DEBUG_MODE
                                fprintf (world->output_file, "(Round %d) Floating move: don't see ball, chan
ging direction, old backwards=%d, new %d, facing=%d\n",
                                        world->time_now, backwards, retval, world->me->facing);
                            #endif
                        } else {
                            backwards_turn_round -= 1;
                            #ifdef CAREFUL_MOVE_DEBUG_MODE
                                fprintf (world->output_file, "(Round %d) Floating move: don't see ball yet,
give turn %d visinfos to find it, backwards = %d, facing=%d\n",
                                        world->time_now, backwards_turn_round, backwards, world->me->facing
);
                            #endif
                        }
                    } else {
                        retval = decide_backwards_when_seeing_ball (world->ball->position, s->facing
* DEG2RAD);
                    #ifdef CAREFUL_MOVE_DEBUG_MODE
                        fprintf (world->output_file, "(Round %d) Floating move: see ball (old backwa
rds = %d, new = %d), facing = %d\n", world->time_now,
                                backwards, retval, world->me->facing);
                    #endif
                }
                backwards_turn_round = 0;
            }
        }
        return retval;
    }
    int Move_to::decide_backwards_when_seeing_ball (Pair &ball_pos, double facingr) {
        double player2ball_angle = (ball_pos - world->me->position).angle();
        double disp_angle = disp_minus_tangent.angle();
        return (abs(normalize_rad (disp_angle - player2ball_angle)) > P5PI);
    }
    int Move_to::okay_run_without_turning (double facingr, Pair &self_pos, Pair &self_ve
l) {
        int rounds_now;
        int rounds_in_future;

```

```

rounds_now = s->rounds_to_point(&destination, tolerance);
Pair unit_facing(cos (facingr), sin (facingr));
Pair this_round_vel;
double this_round_speed;
Pair next_round_pos;

const int ROUNDS_TO_POINT_LOOKAHEAD = 4;

this_round_vel = self_vel;
for (int i = 0; i < ROUNDS_TO_POINT_LOOKAHEAD; i++) {
    this_round_vel += unit_facing * (compute_best_power (unit_facing.dot (self_vel),
max_power) * POWERRATE);
    this_round_speed = this_round_vel.r();
    if (this_round_speed > PLAYER_SPEED MAX)
        this_round_vel *= PLAYER_SPEED MAX / this_round_speed;
    next_round_pos = self_pos + this_round_vel;
    this_round_vel *= PLAYER_DECAY;
}

rounds_in_future = s->hypothetical_rounds_to_point (&destination, tolerance, &next_
_round_pos, &this_round_vel,
    facingr * RAD2DEG, max_power);

#ifdef MOVE2O DEBUG MODE
    fprintf (world->output_file, "Making turn - rounds to dest now: %d, next turn
: %d, velocity: (%lf, %lf)\n",
        rounds_now, rounds_next_turn, this_round_vel.x, this_round_vel.y);
#endif

return (rounds_in_future <= (rounds_now - ROUNDS_TO_POINT_LOOKAHEAD));
}

if 1
void Move_to::make_imm_turn (double dist_remaining) {
double sweep_tolerance; //amount we can be off and still hit the circle
double turn_angle; // angle we need to turn
double needed_turn_angle; // angle we need to turn with due to our vel
int rounds_now = 0;
int rounds_next_turn = 0;
double facingr = s->facing * DEG2RAD;

int need_to_check_another_turn;
int rounds_says_to_turn;
Pair self_vel = s->velocity;
Pair self_pos = s->position;
int first_turn = 1;
Command_obj *insertion_pt = c_lists[0]->get_next();
/*
if (detect_collision (world, s->position, s->velocity,
    compute_runatspeed from maxpower (max_power), facingr))
    //fprintf (world->output_file, "(Round %d) Collision imminent", world->time_now)
;
printf("(Round %d) Collision imminent", world->time_now);
*/
need_to_check_another_turn = missing_turn = !compute_needed_angle (&sweep_toleranc
e, &turn_angle);

while (need_to_check_another_turn) {
    turn_angle = normalize_rad (turn_angle);
    // if ((abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= sweep_tolerance) ||
    // (abs (turn_angle) >= PI*25PI) || dist_remaining < 5) {
        rounds_says_to_turn = 0;
        if (((dest_might_change) || (abs (turn_angle * MAX_TURN_ERROR_PERCENT) > sweep_t
olerance))
            rounds_says_to_turn = (tokay_run_without_turning(facingr, self_pos, self_vel)
);
        if ((rounds_says_to_turn) ||
            ((!dest_might_change) && (abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= swee

```

```

p_tolerance) ||
    (abs (turn_angle) >= P5PI) )
{
/* we want to turn now - we're outside of tolerance and either if we turn no
w we won't miss, or
if we don't turn now our tangential facing makes us lose a round later */
needed_turn_angle = s->compute_needed_angle_at_speed (turn_angle, self_vel
.r());
if (abs(needed_turn_angle) > PI) {
    if (first_turn) {
        c_lists[0]->insert_as_next (new Command_turn(world,
            Sign(needed_turn_angle) *
180.0));
        first_turn = 0;
    } else {
        c_lists[0]->insert_before(insertion_pt,
            new Command_turn (world,
                Sign(needed_turn_angle) *
180.0));
    }
/* update new self data for next run */
turn_angle -= s->compute_turned_angle_at_speed(Sign(needed_turn_angle) *
PI, self_vel.r());
self_pos += self_vel;
self_vel *= PLAYER_DECAY;

    need_to_check_another_turn = missing_turn = 1;
} else { /* needed turn angle is possible at this speed */
    if (first_turn) {
        c_lists[0]->insert_as_next (new Command_turn(world,
            needed_turn_angle*RAD2DEG
        ) );
        first_turn = 0;
    } else {
        c_lists[0]->insert_before(insertion_pt,
            new Command_turn (world, needed_turn_angle*
RAD2DEG) );
        missing_turn = FALSE;
        need_to_check_another_turn = 0;
    }
#ifdef MOVE2O DEBUG MODE
        fprintf (Global_world->output_file,
            "Moveto - Enqueueing turn, Round %d, Angle, %lf, facing %d, \n\
tdesired angle %lf, destination (%lf, %lf)\n",
            global_world->time_now, needed_turn_angle * RAD2DEG, facingr * RA
D2DEG, turn_angle * RAD2DEG, destination.x, destination.y);
#endif
} else { /* we've decided not to turn right now */
    missing_turn = 1;
    need_to_check_another_turn = 0;
} /* end while */
}

void Move_to::make_imm_turn (double dist_remaining) {
double sweep_tolerance; //amount we can be off and still hit the circle
double turn_angle; // angle we need to turn

```

```

double needed_turn_angle; // angle we need to turn with due to our vel
int rounds_now = 0;
int rounds_next_turn = 0;
double facingr = s->facing * DEG2RAD;

if (!compute_needed_angle (&sweep_tolerance, &turn_angle)) {
    turn_angle = normalize_rad (turn_angle);
    // if ((abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= sweep_tolerance) ||
    // (abs (turn_angle) >= PI*25PI) || dist_remaining < 5) {
    if ((dest_might_change) || (abs (turn_angle * MAX_TURN_ERROR_PERCENT) > sweep_t
olerance)) {
        rounds_now = s->rounds_to_point (&destination, tolerance);
        Pair unit_facing (cos (facingr), sin (facingr));
        Pair this_round_vel = s->velocity + unit_facing * (compute_best_power (unit_f
acing.dot (s->velocity), max_power) * POWERRATE);
        double this_round_speed = this_round_vel.r();
        if (this_round_speed > PLAYER_SPEED_MAX)
            this_round_vel *= PLAYER_SPEED_MAX / this_round_speed;
        Pair next_round_pos = s->position + this_round_vel;
        this_round_vel *= PLAYER_DECAV;
        rounds_next_turn = s->hypothetical_rounds_to_point (&destination, tolerance,
&next_round_pos, &this_round_vel,
        facingr * RAD2DEG, max_po
wer);
    }
}
#ifdef MOVE_TO_DEBUG_MODE
printf (world->output_file, "Making turn - rounds to dest now: %d, next turn
: %d, velocity: (%lf, %lf)\n",
        rounds_now, rounds_next_turn, this_round_vel.x, this_round_vel.y);
#endif
// rounds_now = rounds_next_turn - 1 + (turn_angle >= PI*25PI);
}
if ((rounds_next_turn >= rounds_now) ||
    (((dest_might_change) && (abs (turn_angle * MAX_TURN_ERROR_PERCENT) <= swee
p_tolerance)) ||
    {
        (abs (turn_angle) >= PI) )
    }
    /* we want to turn now - we're outside of tolerance and either if we turn no
w we won't miss, or
if we don't turn now our tangential facing makes us lose a round later */
    needed_turn_angle = s->compute_needed_angle_at_speed (turn_angle, s->velocit
y.r());
if (abs (needed_turn_angle) > PI) {
#ifdef MOVE_TO_DEBUG_MODE
printf (Global_world->output_file,
        "Moveto - Enqueueing turn (will need 2 - speed: %lf), Round: %d, facing %
d, angle to input %lf\n", desired_angle %lf, destination (%lf, %lf)\n",
        turn_angle * RAD2DEG, turn_angle * RAD2DEG, destination.x, destination.y);
#endif
    c_lists[0]->insert_as_next (new Command_turn (world,
        Sign(needed_turn_angle) * PI
        *RAD2DEG));
    missing_turn = 1;
}
else { /* needed turn angle is possible at this speed */
    c_lists[0]->insert_as_next (new Command_turn (world, needed_turn_angle * RAD2
DEG ));
    missing_turn = FALSE;
}
#ifdef MOVE_TO_DEBUG_MODE
printf (Global_world->output_file,
        "Moveto - Enqueueing turn, Round %d, Angle, %lf, facing %d, \n",
        esired_angle %lf, destination (%lf, %lf)\n",

```

```

        global_world->time_now, needed_turn_angle * RAD2DEG, facingr * RA
D2DEG, turn_angle * RAD2DEG, destination.x, destination.y);
#endif
    } /* if we need to do the turn now */
    else
        missing_turn = 1; // we will need to turn, but for now dashing is the more
worthwhile action
    } else {
        missing_turn = 0;
    }
}
#endif
double Move_to::current_speed_for_dashes (Pair &velocity, double facing) {
    double retval;
    Pair unit_facing (cos (facing), sin (facing));
    double radial_speed = (missing_turn) ? unit_facing.dot (s->velocity)
        : disp_minus_tangent.dot (velocity) / (displacement.r());
    retval = (radial_speed < 0) ? radial_speed : velocity.r();
    return retval;
}
void Move_to::fill (World *w, Self_obj *s, Pair dest, double max_power, double tol_d
ist) {
    destination = dest;
    displacement = dest - s->position;
    // cerr << "Player " << world->me->su_number << ": Fill called. Dest is: " << dest
ination << endl;
    tolerance = tol_dist;
    this->max_power = (int)max_power;
    double sweep_tolerance, turn_angle;
    double dist_remaining;
    double dash_power;
    double radial_speed;
    missing_turn = 0;
    double needed_turn_angle;
    compute_tangential_displacement(); //need to set regardless of whether action is a
lready finished
    backwards = run_backwards_to_see_ball (watch_ball_too);
    dist_remaining = disp_minus_tangent.r();
    if (dist_remaining > tolerance)
        make_imm_turn (dist_remaining);
    /* make dash commands */
    radial_speed = current_speed_for_dashes (s->velocity, s->facing * DEG2RAD);
    compute_needed_dashes (&dist_remaining, radial_speed, max_power);
    /* take disp_minus_tangent.r() - dist_remaining = dist_travelled
then est_destination = dist_travelled * unit vector in direction of disp_minus_
tangent = */
    est_destination = (1.0 - dist_remaining / disp_minus_tangent.r()) * disp_minus_tan
gent + s->position;
    // printf (name, "Moveto (%G, %G) w/ tol %G", destination.x, destination.y, toler
ance);
}
double Move_to::compute_runatspeed_from_maxpower (double power) {
    double run_at_speed;
    if (backwards) {

```

```

power = min (30, power);
run_at_speed = power * POWERRATE * PLAYER_DIST_FROM_VEL_FACTOR;
} else {
power = min (60, power);
run_at_speed = power * POWERRATE * PLAYER_DIST_FROM_VEL_FACTOR;
if (less wasteful dashes)
run_at_speed = min (run_at_speed, 0.95 * PLAYER_SPEED_MAX);
return run_at_speed;
}

double Move_to::compute_best_power (double current_speed, double max_power) {
double dash_power;

/* run_at_speed is the speed that is the speed we'll be running if we run at max_pow
er for an infinite amount of time. we will dash with a higher power to jump up to
this value if we are running slowly now */

double lmax_power;
double run_at_speed;

run_at_speed = compute_runatspeed_from_maxpower (max_power);
if (backwards) {
dash_power = min ((run_at_speed - current_speed) / POWERRATE, 30); //can't get u
p to speed in one dash if we run backwards
dash_power = max (0, dash_power); //if we're going faster than our run at speed
dash_power *= -1; //make it negative if we're going backwards
}
else {
dash_power = (run_at_speed - current_speed) / POWERRATE; //min( (run_at_speed -
current_speed) / POWERRATE, 100.0);
dash_power = max (dash_power, 0); //make dash power positive
}
return (dash_power);
}

Command_obj * Move_to::recompute_existing_dashes (double *dist_remaining, double *ra
dial_speed, double *max_power) {
Command_obj *temp = c_lists[0]->get_next();
double dash_power;

/* figure out how far our existing dashes go */
while ((temp != NULL) && (*dist_remaining > tolerance)) {
if (temp->command_type() == COMMAND_DASH) {
dash_power = compute_best_power (*radial_speed, max_power);
if (dash_power != ((Command_dash*)temp)->get_power()) {
#ifdef MOVE_TO_DEBUG_MODE
fprintf (world->output file, "Move_to: needed to increase dash power from
%lf to %lf\n", temp->arg_a, dash_power);
#endif
((Command_dash*)temp)->set_power (dash_power);
}
*radial_speed += (((Command_dash*)temp)->get_power() * POWERRATE) * ((backwa
rds) ? -1 : 1);
}
*dist_remaining -= *radial_speed;
#ifdef DEBUG_MODE
assert(temp != temp->next);
#endif
*radial_speed *= PLAYER_DECAY;
temp = ((Command_dash*)temp)->next;
}
return temp;
}

```

```

void Move_to::delete_remaining_commands (Command_obj *first_to_go) {
Command_obj *temp = first_to_go;
Command_obj *mark;

/* if there are still commands in the list here, then that means we're dashing t
oo far
remove those commands now */
while (temp != NULL) {
mark = temp->next;
#ifdef MOVE_TO_DEBUG_MODE
world->debug ("Move_to: deleting a dash in update\n");
#endif
c_lists[0]->delete_command (temp);
temp = mark;
}

void Move_to::compute_needed_dashes (double *dist_remaining, double radial_speed, do
uble max_power) {
double dash_power;

while (*dist_remaining > tolerance) {
dash_power = compute_best_power(radial_speed, this->max_power);
c_lists[0]-insert_last (new Command_dash (world, dash_power));
radial_speed += (dash_power * POWERRATE) * ((backwards) ? -1 : 1);
*dist_remaining -= radial_speed;
radial_speed *= PLAYER_DECAY;
}
}

char Move_to::compute_needed_angle (double *sweep_tolerance, double *turn_angle) {
double facingr = s->facing * DEG2RAD;
double ldistance; //local variable
double want_angle = disp_minus_tangent.angle();
// double player2ball_angle;

if (backwards) { // reversing direction - so turn towards the ball
//player2ball_angle = normalize_rad( (world->ball->position - s->position).angle
());
}
want_angle = want_angle + PI; // * Sign(player2ball_angle);

*turn_angle = want_angle - facingr;
*turn_angle = normalize_rad(*turn_angle);

#ifdef CAREFUL_MOVE_DEBUG_MODE
fprintf (world->output file, "Round %d) Float move, have to make an about face
of %lf (backwards = %d)\n", world->time_now, *turn_angle * RAD2DEG, backwards);
#endif

/* find length of altitude from displacement minus tangent point to current facing
trajectory */
ldistance = disp_minus_tangent.r();
double miss_distance = (ldistance * sin (*turn_angle));
miss_distance = abs (miss_distance);

/* given maximum allowable altitude, find angle at which is occurs
sweep is the same for positive or negative, since error is the same percentage
regardless of which direction we want to face */
*sweep_tolerance = asin (tolerance / ldistance);
return ((abs (*turn_angle) >= P5PI) ? 0 : (miss_distance <= tolerance)); //alway
s miss if we've going in the wrong direction
}

int Move_to::update (void) {

```

```

world = global_world;
if ((world->ball->position - ZERO_PAIR).r() > 2.0) {
    if ((world->ball->position - destination).r() > 2.0) { // Ball has moved
        // Check to see if it has changed. If so -- redo action from scratch
        destination = world->ball->position;
        s = global_world->me; // magic
    }
    return update_calculation ();
} else {
    return 0; // Action has not started yet
}
}

int Move_to::update_calculation (void) {
    double sweep_tolerance, turn_angle;
    displacement = destination - s->position;
    double dist_remaining;
    // double dash_power;
    int next_action_is_turn = 0;
    double radial_speed;
    // double consider_speed_for_dash;
    // double needed_turn_angle;
    int retval;
    Command_obj *final_dash;

    if (displacement.r() < tolerance) {
        missing_turn = 0;
        retval = 0;
        action_done = TRUE;
    }
    else {
        #ifdef MOVETO_DEBUG_MCODE
        fprintf (world->output_file, "\nMTA: update called, round %d\n", world->time_now
        );
        #endif

        compute_tangential_displacement ();
        backwards = run_backwards_to_see_ball (watch_ball_too);

        dist_remaining = disp_minus_tangent.r();

        if (c_lists[0] ->get_next() != NULL)
            next_action_is_turn = (c_lists[0] ->get_next() ->command_type () == COMMAND_TURN
        );
        if ((next_action_is_turn)
            make_imm_turn(dist_remaining);

        /* compute distance we will be moving */
        // Command_obj *temp = c_lists[0] ->nextup;

        /* analyze dash commands */
        radial_speed = current_speed_for_dashes(s->velocity, s->facing * DEG2RAD);
        final_dash = recompute_existing_dashes (&dist_remaining, &radial_speed, max_power)
        ;
        if (final_dash != NULL)
            delete_remaining_commands (final_dash);
        else
            compute_needed_dashes (&dist_remaining, radial_speed, max_power);
        retval = 0;
    }

    est_destination = (1.0 - dist_remaining / disp_minus_tangent.r()) * disp_minus_tan
gent + s->position;
#ifdef MOVETO_DEBUG_MODE_ALWAYS_ROUNDS_TO_POINT
    int rounds_now, rounds_next_turn;
    Pair unit_facing, next_round_pos, next_round_vel;
    int no_inf_loops = 0;
foob:
    rounds_now = s->rounds_to_point(&destination, tolerance);

```

```

    unit_facing = Pair (cos (s->facing * DEG2RAD), sin (s->facing * DEG2RAD));
    next_round_vel = s->velocity + unit_facing;
    if (next_round_vel.r() > PLAYER_SPEED_MAX)
        next_round_vel *= PLAYER_SPEED_MAX / next_round_vel.r();
    next_round_pos = s->position + next_round_vel; //unit_facing; //speed is 1
    next_round_vel *= PLAYER_DECAY;

    rounds_next_turn = s->hypothetical_rounds_to_point (&destination, toleranc
e, &next_round_pos, &next_round_vel,
        s->facing, max_power);

    fprintf (world->output_file, "Moveto_abs_update: My rounds to point is %d
; next turn (running straight): %d\n",
        rounds_now, rounds_next_turn);

    if (!(!(rounds_now < rounds_next_turn + 2)) && (no_inf_loops)) {
        no_inf_loops = 1;
        goto foob;
    }

    // fprintf (world->output_file, "Leaving Moveto_update after visinfo in r
ound %d\n", world->last_vis_info.time);
    //fprintf_action (world->output_file);
    //fprintf (world->output_file, "++++++++++++++++++++++++++++++++++++++++
++++\n");
    #endif
    return retval;
}

```

```
#include <new action.hh>
#include <assert.h>
#include <iostream.h>

Action::Action(int num_clists = 1) {
    pre_state = post_state = NULL;
    prev = next = NULL;
    action_done = FALSE;
    num_command_lists = num_clists;
    c_lists = new Command_list*[num_command_lists];
    action_type = 0;
    for (int i = 0; i < num_command_lists; i++) {
        c_lists[i] = new Command_list();
    }
}

Command_obj* Action::send_next_command_from_list(int list_idx) {
    // Fummy call to ensure_virtual is_finished gets called (maybe unnecessary)
    if (this->is_finished(list_idx)) return NULL;
    return c_lists[list_idx]->send_next_command();
}

int Action::is_finished(int list_id = 0) {
    return action_done;
}
/* return ((c_lists == NULL) ||
(c_lists[list_id] == NULL) ||
c_lists[list_id]->is_finished());*/
}

Action::~Action() {
    pre_state = post_state = NULL;
    prev = next = NULL;
    for (int i = 0; i < num_command_lists; i++) {
        delete c_lists[i];
    }
    delete [] c_lists;
}

void Action::print() {
    cout << "Action ABC" << endl;
}
}
```

```

#include <roboocup_act.hh>
#include <trig.hh>
#include <assert.h>
#include <roboocup_com.hh>
#include <new_move.hh>

Move to::Move_to(World *w) {
    world = w;
    next = prev = NULL;
    s = world->me;
    c_lists[0]->insert_first(new Command_wait());
    formation_positions_set = FALSE;
}

int Move_to::update() {
    if ((world->ball->position - ZERO_PAIR).x() < 2.0) ||
        (world->ball->timestamp != world->last_vis_info_time)
        return 0; // Action has not yet started

    if (move_to_complete()) { // Check for done
        Pair displacement = destination - world->me->position;
        fprintf(world->output_file, "PLAYER %d stopped within %f of position %d\n",
            world->me->u_number, displacement.x(), destination_idx);
        return 1; // Completed -- remove action from queue
    } else {
        // Compute destination
        compute_destination();
        destination = get_dest();

        // Clear the lists
        c_lists[0]->remove_all();
        // Decide if turn is needed
        Pair displacement_needed = destination - world->me->position;
        double needed_turn_angle = normalize_deg
            (RAD2DEG * displacement_needed.angle() - world->me->facing);
        fprintf(world->output_file, "Player %d has dest (%f,%f), pos (%f,%f), facing %d.
\n",
            world->me->u_number, destination.x, destination.y,
            world->me->position.x, world->me->position.y, world->me->facing);

        if (abs(needed_turn_angle) > 5.0) // Enqueue turn (if needed)
            c_lists[0]->insert_first(new Command_turn(world, needed_turn_angle));
        // Enqueue dash
        c_lists[0]->insert_last(new Command_dash(world, 80.0));
        // Enqueue wait
        c_lists[0]->insert_last ( new Command_wait() );
        return 0;
    }
}

void Move_to::compute_destination() {
    // One of the members of the formation will stand
    // to the right of the ball, RADIUS away

    double angle = 0;
    double angle_interval = TWOPI/PLAYERS_IN_FORMATION;
    world = Global_world;
    if (formation_positions_set == FALSE) {
        for (int i = 0;
            i < PLAYERS_IN_FORMATION;
            i++) {
            position_taken[i] = FALSE;
            position_dist[i] = 99999.0;
        }
        formation_positions_set = TRUE;
    }
    for (int i = 0;
        i < PLAYERS_IN_FORMATION;
        i++, angle += angle_interval) {

```

```

        formation_positions[i] =
            world->ball->position + Polar2Pair(FORMATION_RADIUS, angle);
        if ((position_taken[i] < -1) ||
            (position_taken[i] > PLAYERS_IN_FORMATION)) {
            // assert(0); Can't do this -- everyone dies
            position_taken[i] = 0;
        }
        if (position_taken[i] == world->me->u_number)
            position_taken[i] = 0;
    }

    // Set myself at the closest pos to me
    // grab_closest();

    for (int j = 0; j < 11; j++) { // Check every teammate
        if (world->teammates[j] == NULL) continue;
        if (world->teammates[j]->timestamp !=
            world->last_vis_info_time) continue;
        // If we are here -- j is a teammate we have recently seen
        set_player_closest_position(j);
        // position_taken[get_closest_free_pos_index(world->teammates[j]->position)]
        // = world->teammates[j]->u_number;
    }

    // Set the taken ones anew
    int old_dest = destination_idx;
    grab_closest();
    if (old_dest != destination_idx) {
        world->num_dest_changes += 1;
        world->last_dest_time = world->last_vis_info_time;
    }

    world = global_world; // Magic
    // DEBUGGING OUTPUT START
    fprintf(world->output_file, "Player %d. Vis info time: %d : Num changes: %d",
        world->me->u_number, world->last_vis_info_time, world->num_dest_changes);
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        fprintf(world->output_file, "%d", position_taken[i]);
        if (position_dist[i] == 0.0)
            fprintf(world->output_file, "***");
        fprintf(world->output_file, " | ");
    }
    fprintf(world->output_file, "\n\n");
    // DEBUGGING OUTPUT END
}

void Move_to::set_player_closest_position(int teammate_idx) {
    double min_distance = 9999.0;
    int nearest_position = -1;

    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        // Clear the position this player had before this
        if ((position_taken[i] == world->teammates[teammate_idx]->u_number) &&
            (position_dist[i] == 0.0))
            return; // This player has CLAIMED a position
    }

    for (int i = 0;
        i < PLAYERS_IN_FORMATION;
        i++) {
        // if (position_taken[i] != 0)
        // continue;
        distance = (formation_positions[i] - world->teammates[teammate_idx]->position).r
    );
    if (distance <= FORMATION_TOLERANCE) {

```

```

distance = 0.0;
if (position_taken[i] == -1)
    position_dist[i] = FORMATION_TOLERANCE;
}
if ((position_taken[i] == FALSE) || (distance < position_dist[i])) &&
    (distance < min_distance){
    min_distance = distance;
    nearest_position = i;
}
}
// assert(nearest_position != -1);
for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
    // Clear the position this player had before this
    if (position_taken[i] == world->teammates[teammate_idx]->u_number)
        position_taken[i] = FALSE;
}
if ((position_taken[nearest_position] == 0)
    || (position_dist[nearest_position] >= min_distance))
{
    position_taken[nearest_position] =
        world->teammates[teammate_idx]->u_number;
    position_dist[nearest_position] =
        (min_distance <= FORMATION_TOLERANCE) ? 0.0 : min_distance;
}
}

void Move to::grab closest() {
    double min_distance = 9999.0;
    double distance;
    int nearest_position = -1;
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        // Clear the position this player had before this
        if (position_taken[i] == world->me->u_number) {
            position_taken[i] = FALSE;
        }
    }
    // This is preventing strange intermittent errors...
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        if ((position_taken[i] < -1) ||
            (position_taken[i] > PLAYERS_IN_FORMATION))
            position_taken[i] = 0;
    }
    for (int i = 0;
        i < PLAYERS_IN_FORMATION ;
        i++) {
        distance = (formation_positions[i] - world->me->position).r();
        if ((position_taken[i] == FALSE) || (distance < position_dist[i]))
            && (distance < min_distance)) {
                min_distance = distance;
                nearest_position = i;
            }
    }
    if (nearest_position != -1) {
        position_taken[nearest_position] = world->me->u_number;
        position_dist[nearest_position] = min_distance;
        destination_idx = nearest_position;
    } else {
        for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
            if (position_taken[i] == -1) nearest_position = i;
        }
        if (nearest_position != -1) {
            position_taken[nearest_position] = world->me->u_number;
            position_dist[nearest_position] = min_distance;
            destination_idx = nearest_position;
        }
    }
}

```

```

if (destination_idx == -1) {
    cerr << "player " << world->me->u_number << " got -1 at " << world->last_vis_inf
        o_time << " : ";
}
for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
    cerr << position_taken[i] ;
    if (position_taken[i] > PLAYERS_IN_FORMATION)
        cerr << "??" << position_dist[i];
    cerr << " | ";
}
cerr << "\n";
}
}
Pair Move to::get_dest() {
    if (destination_idx == -1) {
        assert(0);
        // cout << "It is -1!" << endl;
        return (world->me->position);
    } else {
        return formation_positions[destination_idx];
    }
}
int Move to::move to complete() {
    return ((destination - world->me->position).r() < 2.0);
}
}

```

```

#include <pair.hh>
Pair::Pair() {
    x = 0;
    y = 0;
    error = 0;
}

Pair::Pair(double in_x, double in_y) {
    x= in_x;
    y= in_y;
    error = 0;
}

int Pair::equals_pair(Pair b){
    return ((x == b.x) && (y == b.y));
}

Pair Pair::add_pair(Pair b) {
    Pair c(x + b.x, y + b.y);
    return c;
}

Pair Pair::subtract_pair(Pair b) {
    Pair c(x - b.x, y - b.y);
    return c;
}

//delta_pair interprets it arguments as a range and angle to translate this pair by
.
Pair Pair::delta_pair(double dist, double theta){
    Pair c;
    c.x = x + (dist * cos(theta));
    c.y = y + (dist * sin(theta));
    return c;
}

Pair Pair::delta_pair(Pair b) {
    return delta_pair(b.x,b.y);
}

void Pair::operator =(const Pair& v)
{
    x = v.x ;
    y = v.y ;
    error = v.error;
}

void Pair::operator +=(const Pair& v)
{
    x += v.x ;
    y += v.y ;
}

void Pair::operator =(const Pair& v)
{
    x -= v.x ;
    y -= v.y ;
}

void Pair::operator *=(const double& a)
{
    x *= a ;
    y *= a ;
}

void Pair::operator /=(const double& a)
{
    x /= a ;
    y /= a ;
}

```

```

ostream& operator<< (ostream& o, const Pair& v)
{
    return o << "#V[" << v.x << ", " << v.y << "]" ;
}

Pair operator +(const Pair& a, const Pair& b)
{
    return Pair((a.x + b.x), (a.y + b.y)) ;
}

Pair operator -(const Pair& a, const Pair& b)
{
    return Pair((a.x - b.x), (a.y - b.y)) ;
}

Pair operator *(const Pair& a, double b)
{
    return Pair (a.x * b, a.y * b);
}

Pair operator *(double b, const Pair& a)
{
    return Pair (a.x * b, a.y * b);
}

void Pair::normalize(const double& l)
{
    *this *= (1/max(r(),EPS)) ;
}

double Pair::distance(const Pair& orig)
{
    return (*this - orig).r() ;
}

double Pair::angle()
{
    return Atan(y,x) ;
}

double Pair::angle(const Pair& dir)
{
    double ang = Atan(dir.y,dir.x) - Atan(y,x) ;
    return normalize_rad(ang) ;
}

void Pair::rotate(const double& ang)
{
    double r1 = r() ;
    double th1 = th() ;
    x = r1 * cos(th1 + ang) ;
    y = r1 * sin(th1 + ang) ;
}

double Pair::vangle(const Pair& target, const Pair& origin)
{
    Pair axis = origin - *this ;
    Pair dir = target - *this ;
    return axis.angle(dir) ;
}

double Pair::vangle(const Pair& target, const double& origin)
{
    Pair dir = target - *this ;
    double ang = dir.angle() - origin ;
    return normalize_rad(ang) ;
}

```

```

#include <new action.hh>
#include <world.hh>
#include <robocup_act.hh>
#ifdef CHASER
#include <new_move.hh>
#endif

static Action_Queue *main_queue;

// This is automatically atomic
// Because it is only called from the interrupt handler
void handle_alarm() {
    // spin_lock(main_queue->q_lock);
    global_world->alarms_since_last_info++;
    switch (main_queue->timer_cycle) {
    case 0:
        send_com_sense(global_world->me->sock);
        break;
    case 1:
#ifdef CHASER
        if ((global_world->ball->see_timestamp != global_world->last_vis_info.time)
            && (main_queue->is_empty() || (main_queue->get_first()->action_type != 1)))
        {
            while (!main_queue->is_empty())
                main_queue->remove_action(main_queue->get_first());
            prepare_starting_queue(global_world, main_queue);
        }
#endif
        if ((main_queue->is_empty()) {
            // main_queue->first->c_lists[0]->check_list();
            int ret_val = main_queue->send_next_command();
            assert (ret_val != 2); // Queue is not empty
            if ((ret_val != 1) && // Success
                (ret_val != 5) // Action is done
                cerr << "bad send_command: " << ret_val << endl << flush;
            )
                break;
            default:
                assert(0);
            break;
        }
        main_queue->timer_cycle = (main_queue->timer_cycle == 0) ? 1 : 0;
        // release_lock(main_queue->q_lock);
    }

    Action_Queue::Action_Queue() {
        first = last = NULL;
        world_state = NULL;
        main_queue = this;
        update_first_action_only = FALSE;
        // Prepare the timer stuff
        itv.it_interval.tv_sec = 0;
        itv.it_interval.tv_usec = 50000;
        itv.it_value.tv_sec = 0;
        itv.it_value.tv_usec = 50000;
        alarm_action.sa_handler = (void (*)(int))handle_alarm;
        alarm_action.sa_flags &= (-SA_RESETHAND);
        timer_cycle = 0;
        q_lock = new ablock;
        init_lock(q_lock);
    }

    Action_Queue::~Action_Queue() {
        itv.it_interval.tv_sec = 0;
        itv.it_value.tv_sec = 0;
        setitimer(ITIMER_REAL, &itv, NULL);
        Action *temp;
        while (first != NULL) {
            temp = first;
            first = first->next;
            delete temp;
        }
        first = last = NULL;
}

```

```

        world_state = NULL;
    }

    void Action_Queue::insert_last (Action *act) {
        if (act != NULL) {
            if (!is_empty()) {
                last->next = act;
                act->prev = last;
                act->next = NULL;
                last = act;
                act->aq = this;
            } else {
                insert_into_blank_list(act);
            }
        }
    }

    void Action_Queue::insert_into_blank_list(Action *act) {
        last = first = act;
        act->next = act->prev = NULL;
        act->aq = this;
    }

    void Action_Queue::insert_first (Action *act) {
        if (act != NULL) {
            if (!is_empty()) {
                first->prev = act;
                act->next = first;
                act->prev = NULL;
                first = act;
                act->aq = this;
            } else {
                insert_into_blank_list(act);
            }
        }
    }

    int Action_Queue::is_empty () {
        if (first == NULL) {
            assert (last == NULL);
            return TRUE;
        } else {
            assert (last != NULL);
            return FALSE;
        }
    }

    int Action_Queue::insert_before (Action *before, Action *act) {
        if (is_empty() || (!is_in_list (before)))
            return FALSE;
        if (before == first) {
            insert_first (act);
            return TRUE;
        } else {
            before->prev->next = act;
            act->prev = before->prev;
            before->prev = act;
            act->next = before;
            act->aq = this;
            return TRUE;
        }
        // control should not reach here
        return FALSE;
    }

    // Just a simple pointer comparison
    int Action_Queue::is_in_list(Action *act) {
        Action* temp = NULL;
        if (act == NULL) return(0);
        for (temp = first ; temp != NULL; temp = temp->next) {

```

```

    if (temp == act)
    }
    return(1);
}
return(0);
}

// Again do the find by simple pointer comparison
// Note that this does not actually delete the action
int Action_Queue::remove_action(Action *act) {
    if (!is_in_list(act)) return FALSE;
    if (act->prev != NULL)
        act->prev->next = act->next;
    if (act->next != NULL)
        act->next->prev = act->prev;
    if (act == first) {
        first = act->next;
    }
    if (act == last) {
        last = act->prev;
    }
    act->next = act->prev = NULL;
    act->aq = NULL;
    return TRUE;
}

void Action_Queue::start_timer() {
    int i = sigaction(SIGALRM, &alarm_action, NULL);
    setitimer(ITIMER_REAL, &tv, NULL);
}

Update_info* Action_Queue::update_queue() {
    Action *current_action = first;
    Update_info *ret_val = new Update_info();
    int fail = FALSE;
    if (current_action == NULL) {
        ret_val->code = 0; // success
        return ret_val;
    }
    while (current_action != NULL) {
        if (current_action->action_done) {
            current_action = current_action->next;
            continue;
        }
        ret_val->code = current_action->update();
        ret_val->last_action = current_action;
        if (ret_val->code == 0) { // success
            if ((current_action == first) && update_first_action_only) {
                break;
            }
        }
        current_action = current_action->next;
    }
    // For now do this no matter what the fail code is
    int is_first, is_last = FALSE;
    if (current_action == first) is_first = TRUE;
    if (current_action == last) is_last = TRUE;
    int ra = remove_action(current_action);
    assert(ra); // That action damn better have been in the queue
    if (ret_val->code == SECRET_CODE) {
        // We need to reconstruct a move to action
        *Action *phoenix = new Move_to((Move_to*)current_action);
        ((Move_to*)phoenix)->tolerance = ((Move_to*)current_action)->tolerance; // M
        agic
        ((Move_to*)phoenix)->watch_ball_too = FALSE; // More magic
    }
    if (is_first)
        first = phoenix;
    if (is_last)
        last = phoenix;
}

```

```

    }
    return ret_val;
}

int Action_Queue::send_next_command() {
    if (is_empty())
        return 2; // Queue is empty
    Action *current_action = NULL;
    for (current_action = first;
         ((current_action != NULL) && current_action->is_finished());
         if (current_action == current_action->next) {} // Get to next unfinished action
         current_action->c_lists[0]->check_list();
         Command_obj *co = current_action->send_next_command_from_list(0);
         if (co == NULL) {
             if (current_action->is_finished())
                 return 5;
             else
                 return 4; // Command not sent unsuccessfully
         }
         else {
             return 1; // Success
         }
    }

    // This will remove all finished actions
    // from the queue and delete them
    void Action_Queue::clean_finished() {
        Action *current_action, *action_to_delete;
        for (current_action = first;
             current_action != NULL; ) {
            if (current_action->is_finished()) {
                action_to_delete = current_action;
                current_action = current_action->next;
                remove_action(action_to_delete);
                delete action_to_delete;
            }
            else {
                current_action = current_action->next;
            }
        }
    }
    // Update info stuff
    //*****
    Update_info *update_info = new Update_info();
    code = -99;
    last_action = NULL;
}

Update_info::~Update_info() {}

int Update_info::is_filled() {
    return (code != -99);
}

```

```

#include <robocup_act.hh>
#include <world.hh>
#include <robocup_com.hh>

// Assumes ball is stationary
// Assumes we have not turned since last vis info
// Assumes that an update is called with each new vis_info
Find_ball::Find_ball(World *w) {
    world = w;
    action_type = 1;
    if (world->ball->see_timestamp == world->last_vis_info_time) {
        Pair next_round_displacement = world->ball->position - world->me->position;
        double needed_turn_angle = normalize_rad(next_round_displacement.angle() - world->me->facing*DEG2RAD);
        c_lists[0] ->insert_first(new Command_turn(world, RAD2DEG * needed_turn_angle));
    }
    // A turn
    // c_lists[0] ->insert_last(new Command_wait()); // Wait to confirm that we s
    aw it
    } else {
        enqueue_turn_270();
    }
}

Find_ball::~Find_ball() {
    if (c_lists != NULL) {
        if (c_lists[0] != NULL) {
            delete c_lists[0];
        }
        delete [] c_lists;
    }
}

int Find_ball::update() {
    if (world->ball->see_timestamp == world->last_vis_info_time) { // We saw it!
        if (c_lists[0] ->get_next() != NULL) { // The action has not already finished
            c_lists[0] ->remove_all(); // Clear the list
            Pair next_round_displacement = world->ball->position - world->me->position;
            double needed_turn_angle = normalize_deg(RAD2DEG * next_round_displacement.angle() - world->me->facing);
            // Enqueue a turn to ball
            c_lists[0] ->insert_first(new Command_turn(world, needed_turn_angle));
        } else {
            action_done = TRUE;
            return 0; // The action is done
        }
    } else {
        // cerr << "LV: " << world->last_vis_info_time <<
        // "stamp: " << world->ball->see_timestamp << "\n";
        c_lists[0] ->remove_all(); // Clear the list
        enqueue_turn_270();
    }
    return 0; // SUCCESS!!
}

void Find_ball::enqueue_turn_270() {
    c_lists[0] ->insert_first(new Command_turn(world, 90.0));
    c_lists[0] ->insert_last(new Command_wait());
    c_lists[0] ->insert_first(new Command_turn(world, 90.0));
    c_lists[0] ->insert_last(new Command_wait());
    c_lists[0] ->insert_first(new Command_turn(world, 90.0));
    c_lists[0] ->insert_last(new Command_wait());
}

#if 0 // For new move
/*****
 * MOVE INTO FORMATION
 *****/
Move_into_formation::Move_into_formation(World *w) {
    world = w;
    destination_idx = -1;
    if (world->ball->see_timestamp == world->last_vis_info_time) {

```

```

        construct_from_scratch();
    } else {
        // Do nothing. We cannot start the action yet
    }
}

Move_into_formation::~Move_into_formation() {
    if (c_lists != NULL) {
        if (c_lists[0] != NULL) {
            delete c_lists[0];
        }
        c_lists[0] = NULL;
    }
    delete [] c_lists;
}

Pair Move_into_formation::get_dest() {
    if (destination_idx == -1) {
        return (world->me->position);
    } else {
        return formation_positions[destination_idx];
    }
}

//if 0
Pair Move_to::get_dest() {
    if (destination_idx == -1) {
        // cout << "It is -1!" << endl;
        return (world->me->position);
    } else {
        return formation_positions[destination_idx];
    }
}

// Assumes we have seen the ball
void Move_into_formation::construct_from_scratch() {
    assert(c_lists[0] ->is_empty());
    double dist_to_dest = compute_destination();
    double turn_to_dest = normalize_deg(RAD2DEG * get_dest().angle() - world->me->facing);
    // The facing tolerance should be greater if we are farther from ball
    //double facing_tol_fact = ceiling(dist_to_dest/FORMATION_TOLERANCE);
    if (turn_to_dest > FORMATION_TURN_TOL_DEG/**facing_tol_fact*/) {
        // We need to turn
        c_lists[0] ->insert_first(new Command_turn(world, turn_to_dest));
    }
    if (dist_to_dest < FORMATION_TOLERANCE) { // We are all set
        action_done = TRUE;
        return;
    }
    // For now just enqueue a couple of dashes here
    c_lists[0] ->insert_last(new Command_dash(world, 50));
    c_lists[0] ->insert_last(new Command_dash(world, 30));
    c_lists[0] ->insert_last(new Command_dash(world, 30));
}

// This assumes we already know where the ball is
// It returns the distance to the destination
// It computes all destinations for the formation
// And the desired dest. of this player
double Move_into_formation::compute_destination() {
    // One of the members of the formation will stand
    // to the left of the ball, RADIUS away
    cerr << "Choosing destination. Ball is at: " << world->ball->position;
    cerr << " I am at: " << world->me->position;
    double angle = 0;
    double angle_interval = TWOPI/PLAYERS_IN_FORMATION;
    double min_distance = 9999.0;
    double distance;
    for (int i = 0;
         i < PLAYERS_IN_FORMATION;
         i++, angle += angle_interval) {

```

```

formation_positions[i] =
    world->ball->position + Polar2Pair(FORMATION_RADIUS, angle);
distance = (formation_positions[i] - world->me->position).r();
if (distance < min_distance) {
    min_distance = distance;
    destination_idx = i;
}
}
return min_distance;
}

void MoveTo::compute_destination() {
    // One of the members of the formation will stand
    // to the right of the ball, RADIUS away

    double angle = 0;
    double angle_interval = TWOPI/PLAYERS_IN_FORMATION;
    world = global_world;
    if (!formation_positions_set) {
        for (int i = 0;
            i < PLAYERS_IN_FORMATION;
            i++, angle += angle_interval) {
            formation_positions[i] =
                world->ball->position + Polar2Pair(FORMATION_RADIUS, angle);
            position_taken[i] = FALSE;
            position_dist[i] = 9999.0;
        }
        formation_positions_set = TRUE;
    }
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        // This is an attempt to fix any random weirdness
        // That seems to often crop into this code
        if ((position_taken[i] < -1) ||
            (position_taken[i] > PLAYERS_IN_FORMATION)) {
            // assert(0); Can't do this -- everyone dies
            position_taken[i] = 0;
        }
    }

    // Set myself at the closest pos to me
    // grab_closest();

    for (int j = 0; j < 11; j++) { // Check every teammate
        if (world->teammates[j] == NULL) continue;
        if (world->teammates[j]->timestamp !=
            world->last_vis_info_time) continue;
        // If we are here -- j is a teammate we have recently seen
        set_player_closest_position(j);
        // position_taken[get_closest_free_pos_index(world->teammates[j]->position)
    ]
    // = world->teammates[j]->u_number;
    // Set the taken ones anew

    grab_closest();

    world = global_world; // Magic
    // DEBUGGING OUTPUT START
    fprintf(world->output_file, "Player %d. Vis info time: %d :",
            world->me->u_number, world->last_vis_info_time);
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++)
        fprintf(world->output_file, "%d ", position_taken[i]);
    fprintf(world->output_file, "\n");
    // DEBUGGING OUTPUT END

```

```

if (destination_idx == -1) {
    cerr << "player " << world->me->u_number << " got -1: ";
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++)
        cerr << "position_taken[" << i << "] = ";
}
}

int MoveTo::all_taken() {
    for (int i = 0;
        i < PLAYERS_IN_FORMATION;
        i++) {
        if (position_taken[i] == FALSE)
            return FALSE;
    }
    return TRUE;
}

// With no argument will get the closest pos to you
int MoveTo::get_closest_free_pos_index(Pair pos) {
    double min_distance = 9999.0;
    double distance;
    int ret_val = -1;
    for (int i = 0;
        i < PLAYERS_IN_FORMATION;
        i++) {
        if ((position_taken[i] != 0) &&
            (position_taken[i] != world->me->u_number))
            continue;
        distance = (formation_positions[i] - pos).r();

        if (distance < min_distance) {
            min_distance = distance;
            ret_val = i;
        }
    }
    position_taken[ret_val] = world->me->u_number;
    position_dist[ret_val] = min_distance;
    // assert(ret_val != -1);
    return ret_val;
}

void MoveTo::set_player_closest_position(int teammate_idx) {
    double min_distance = 9999.0;
    double distance;
    int nearest_position = -1;
    for (int i = 0;
        i < PLAYERS_IN_FORMATION;
        i++) {
        // if (position_taken[i] != 0)
            // continue;
        distance = (formation_positions[i] - world->teammates[teammate_idx]->position).r
    ();
    if (distance <= FORMATION_TOLERANCE) distance = 0.0;
    if ((position_taken[i] == FALSE) || (distance < min_distance)) {
        min_distance = distance;
        nearest_position = i;
    }
}

assert(nearest_position != -1);
for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
    // Clear the position this player had before this
    if (position_taken[i] == world->teammates[teammate_idx]->u_number)
        position_taken[i] = FALSE;
}

if ((position_taken[nearest_position] == 0)
    || (position_dist[nearest_position] >= min_distance))

```

```
{
    position_taken[nearest_position] =
        world->teammates[teammate_idx]->u_number;
    position_dist[nearest_position] =
        (min_distance <= FORMATION_TOLERANCE) ? 0.0 : min_distance;
}

}

void Move to::grab_closest() {
    double min_distance = 9999.0;
    double distance;
    int nearest_position = -1;
    for (int i = 0; i < PLAYERS_IN_FORMATION; i++) {
        // Clear the position this player had before this
        if (position_taken[i] == world->me->u_number) {
            position_taken[i] = FALSE;
        }
    }
    for (int i = 0 ;
         i < PLAYERS_IN_FORMATION ;
         i++) {
        distance = (formation_positions[i] - world->me->position).r();
        if ((position_taken[i] == FALSE) || (distance < min_distance)) {
            min_distance = distance;
            nearest_position = i;
        }
    }
    position_taken[nearest_position] = world->me->u_number;
    position_dist[nearest_position] = min_distance;
    destination_idx = nearest_position;
    // (min distance <= FORMATION_TOLERANCE) ? 0.0 : min_distance;
    // if (min_distance <= FORMATION_TOLERANCE)
    // cerr << "I, Player " << world->me->u_number << " am set at Pos "
    // << nearest_position << endl;
}

int Move into formation::update() {
    c_lists[0]->remove_all();
    construct_from_scratch();
    return 0;
}

#endif //For new_move

Kick_ball_action::Kick_ball_action(World *w) {
    for (int i = 0 ; i < I0 ; i++)
        c_lists[0]->insert_last(new Command_kick(world, 100.0, 0));
}

Kick_ball_action::update() {
    return 0;
}
```

```

#include <robocup_com.hh>
Command_dash::Command_dash(World* w, double p) {
    prev = next = NULL;
    world = w;
    power = p;
}

int Command_dash::send_command() {
#ifdef DEBUG_COM
    cout << "Sending dash at " << power << endl;
#endif
    return send_com_dash(world->me->sock, power);
}

void Command_dash::print () {
    cout << "Dash at " << power << endl;
}

Command_turn::Command_turn(World* w, int a) {
    prev = next = NULL;
    world = w;
    angle = a;
}

int Command_turn::send_command() {
#ifdef DEBUG_COM
    cout << "Sending turn at " << angle << " deg" << endl;
#endif
    return send_com_turn(world->me->sock, angle);
}

void Command_turn::print () {
    cout << "Turn at " << angle << " deg " << endl;
}

Command_move::Command_move(World* w, double xc, double yc) {
    prev = next = NULL;
    world = w;
    x = xc;
    y = yc;
}

int Command_move::send_command() {
#ifdef DEBUG_COM
    cout << "Sending move at x: " << x << " , y: " << y << endl;
#endif
    world->me->odo_position.x = x;
    world->me->odo_position.y = y;
    return send_com_move(world->me->sock, x, y);
}

void Command_move::print () {
    cout << "Move at x: " << x << " , y: " << y << endl;
}

Command_kick::Command_kick(World* w, double p, int a) {
    prev = next = NULL;
    world = w;
    power = p;
    angle = a;
}

int Command_kick::send_command() {
#ifdef DEBUG_COM
    cout << "Sending kick at power: " << power << " , dir: " << angle << " deg" << endl;
#endif
    world = global_world; // Magic
    return send_com_kick(global_world->me->sock, power, angle);
}

```

```

void Command_kick::print () {
    cout << "Kick at power: " << power << " , dir: " << angle << " deg" << endl;
}

Command_catch::Command_catch(World* w, int a) {
    prev = next = NULL;
    world = w;
    angle = a;
}

int Command_catch::send_command() {
#ifdef DEBUG_COM
    cout << "Sending catch at dir: " << angle << " deg" << endl;
#endif
    return send_com_catch(world->me->sock, angle );
}

void Command_catch::print () {
    cout << "Catch at dir: " << angle << " deg" << endl;
}

```

```

// **c++**
// The above line causes xemacs to open this file in c++-mode
#include <sensor.hh>
#include <assert.h>

Self_obj::Self_obj() {
    goalie = 0;
    team = TEAM_US;
    stamina = (int)STAMINA_MAX;
    effort = 1; //!we think this defaults to 1.

    view_width = VIEW_WIDTH_NORMAL;
    view_quality = VIEW_QUALITY_HIGH;

    view_step_clicks = 0;
    view_step_counter = 8;
    should_recieve_pass = FALSE;
    catch_time = 0;
}

Self_obj::Self_obj(Socket* sock_in) {
    goalie = 0;
    stamina = (int)STAMINA_MAX;
    effort = 1; //!we think this defaults to 1.
    team = TEAM_US;
    view_width = VIEW_WIDTH_NORMAL;
    view_quality = VIEW_QUALITY_HIGH;

    view_step_clicks = 0;
    view_step_counter = 8;
    should_recieve_pass = FALSE;
    catch_time = 0;

    sock = sock_in;
}

// Assumes that side and u number are set
void Self_obj::set_starting_position() {
    Pair start_pos;
    start_pos.x = (double)((side == LEFT ? -3 : 3) * u_number);
    start_pos.y = -37.0;
    if (side == RIGHT)
        facing = odo.facing = 180;
    position = start_pos;
    odo_position = start_pos;
}

void Self_obj::fprintf_self(FILE* file) {
    fprintf(file, "Hi! I'm player %d \n", u_number);
    fprintf(file, "\tI'm at (odo) : %f,%f, facing: %d\n",
            odo.position.x, odo.position.y, odo.facing);
    fprintf(file, "\tI'm at (combo) : %f,%f, facing: %d\n",
            position.x, position.y, facing);
    fprintf(file, "\tI am moving at: %f, %f, \n", velocity.x, velocity.y);
}

int Self_obj::compute_position(World* world, Vis_info* vis_info) {
    int out_of_field = FALSE;
    int closest_obj_is_goal = FALSE, second_closest_obj_is_goal = FALSE;
    int new_facing = VIS_DIR_ERR;
    Pair new_position, new_velocity;
    double closest_obj_dist = 0.0;
    Vis_obj* closest_obj = NULL, *second_closest_obj = NULL;

    // Check for out of field
    if ((vis_info->num_lines == 0) || (vis_info->num_lines == 2))
        out_of_field = TRUE;

    // This probably means that we ran off the field and

```

```

// past the external flags and are facing the wrong way.
// It is possible to check out_of_field and be a little bit more intelligent
// about this
if ((vis_info->num_goals == 0) && (vis_info->num_flags == 0)) {
    out_of_field = TRUE;
    world->debug("I don't see nothin'. Canno' cumpoot pozishun.\n");
}

// Find closest object
// May want to keep the two closest ones instead
if (vis_info->num_goals != 0) {
    closest_obj_is_goal = TRUE;
    if (vis_info->num_goals == 2) {
        second_closest_obj_is_goal = TRUE;
        if (vis_info->goals[0].dist < vis_info->goals[1].dist) {
            closest_obj = &vis_info->goals[0];
            second_closest_obj = &vis_info->goals[1];
        } else {
            closest_obj = &vis_info->goals[1];
            second_closest_obj = &vis_info->goals[0];
        }
    } else { // Only one goal seen
        closest_obj = &vis_info->goals[0];
    }
}

Vis_obj* temp;
for(int i = 0; i < NUM_FLAGS; i++) {
    if (vis_info->flags[i].timestamp != vis_info->timestamp) continue;
    if (vis_info->flags[i].in_cone == FALSE) continue;
    if (closest_obj == NULL) {
        closest_obj = &vis_info->flags[i];
        closest_obj_is_goal = FALSE;
    } else if ((second_closest_obj == NULL) ||
               (vis_info->flags[i].dist < second_closest_obj->dist)) {
        second_closest_obj = &vis_info->flags[i];
        second_closest_obj_is_goal = FALSE;
    }
}

if ((second_closest_obj != NULL) && (second_closest_obj->dist < closest_obj->dist)) {
    temp = closest_obj;
    closest_obj = second_closest_obj;
    second_closest_obj = temp;
    second_closest_obj_is_goal = closest_obj_is_goal;
    closest_obj_is_goal = FALSE;
}

//ifdef ASSMONEY
fprintf(global_world->output_file, "%f is closest. %f is second closest.\n",
        closest_obj->dist, second_closest_obj->dist);
#endif
assert(! (second_closest_obj_is_goal && closest_obj_is_goal));

// Compute facing
if (vis_info->num_lines != 0) {
    new_facing = vis_info->lines[0].compute_direction(out_of_field);
} else if (second_closest_obj != NULL) {
    new_facing = closest_obj->compute_direction(closest_obj_is_goal, second_closest_obj);
} else {
    world->debug("No linz. Unlee uno objax. Canno' cumpoot pozishun.\n");
}
return FALSE;

// Compute position
if ((vis_info->num_lines != 0) && (vis_info->lines[0].dist < closest_obj->dist))
    new_position = vis_info->lines[0].compute_position(closest_obj,

```



```

// **c++**
// The above line causes xemacs to open this file in c++-mode
#include <sensor.hh>
#include <actions.hh>
#include <assert.h>

#ifdef DEBUG MODE
int num_visinfos = 0;
#endif

int Vis_info::fill_info(World* world, char* buffer) {
    cerr << " "; // Magic
    Vis_obj* object = new Vis_obj();
    char* next;
    char* param;
    char* obj;
    char* tmp;
    int num = 0;
    char* teamname = world->me->team_name;
    int claimed_us[11];
    int claimed_them[11];
    int our_index;
    int saw_wide_info = FALSE;
    int flag_index = -1;

    //init our claimed arrays.
    for (int i=0;i<11;i++) {
        claimed_us[i] = -1;
        claimed_them[i] = -1;
    }

    //save the original string
    memcpy(original_buffer, buffer, VISBUFSIZE);

    if (buffer == NULL) {
        // fprintf(world->output_file, "buffer is null!");
        return FALSE;
    }

    // pull off the initial see and save the time.
    // fprintf(world->output_file, "%s|", buffer);
    sscanf(buffer, "(see %d", &timestamp);

    buffer = next_token(buffer + 1);
    if (buffer == NULL) {
        //fprintf(world->output_file, "buffer contained |(see| only \n");
        //fprintf(world->output_file, "orig_buffer contained %s \n", original_buffer);
        delete object;
        object = NULL;
        return FALSE;
    }
    buffer = next_token(buffer);

    while (buffer != NULL) {
        next = next_token(buffer++);
        param = next_token(buffer++);

        if (param != NULL) {
            tmp = param;
            param = next_token(tmp);
            object->dist = atof(tmp);
        }
        else
            object->dist = VIS_DIST_ERR;

        if (param != NULL) {
            tmp = param;
            param = next_token(tmp);
            object->dir = atof(tmp);
        }
        else
            object->dir = VIS_DIR_ERR;
    }
}

```

```

if (param != NULL) {
    tmp = param;
    param = next_token(tmp);
    object->dist_change = atof(tmp);
}
else
    object->dist_change = VIS_CHNG_ERR;

if (param != NULL) {
    tmp = param;
    param = next_token(tmp);
    object->dir_change = atof(tmp);
}
else
    object->dir_change = VIS_CHNG_ERR;

if (param != NULL) {
    tmp = param;
    param = next_token(tmp);
    object->facing = atoi(tmp);
}
else
    object->facing = VIS_DIR_ERR;

object->timestamp = world->last_vis_info_time = world->time_now = timestamp;

if (object->dir == VIS_DIR_ERR) {
    /* quality = VIEW_QUALITY_LOW;
    num_players = num_lines = num_goals = 0;
    // We are getting_low quality info
    char temp[100];

    sprintf(temp, "Got low quality visual at time %d.\n", timestamp);
    world->debug(temp);
    if (world->me->view_quality != VIEW_QUALITY_LOW) { // A change view command di
d not get through to the server
        // Send it again, yo!!
        if (world->change_view_act == NULL) {
            world->change_view_act = new Change_view_action(world, world->me->view_wid
th, world->me->view_quality);
        }
    }
    return 2; // Got low quality
    */
    printf("Got VIS_DIR_ERR at time %d.\n", timestamp);
}

obj = next_token(buffer);
if ((abs(object->dir) > 45) && (strcmp(buffer, "line") != 0) &&
    (buffer[0] != 'F') && (buffer[0] != 'G')) && (buffer[0] != 'P') && (buffer[0]
!= 'B')) {
    saw_wide_info = TRUE;
    // _assert(0); // For now we catch this case

    // If the object is in the neighborhood and in the cone of vision,
    // only the lowercased (Cone of vision) entry will be recieved
    if (!strcmp(buffer, "ball")) {
        ball.fill(object);
        ball.in_cone = TRUE;
        ball.ball = world->ball;
    }
    else if (!strcmp(buffer, "Ball")) {
        ball.fill(object);
        ball.in_cone = FALSE;
        ball.ball = world->ball;
    }
    else if (!strcmp(buffer, "player")) {
        object->in_cone = TRUE;
        players[num_players].fill(object);
    }
}

```

```

/*
//dh added - might be useful, but crash is not because of this
assert(players[num_players] != NULL);
char debug_buf[20];
sprintf(debug_buf, "Player Addr: %X\n", players[num_players]);
global_world -> debug(debug_buf);
*/

players[num_players].team = TEAM_UNKNOWN;
players[num_players].unum = VIS_UNUM_ERR;
players[num_players].player = NULL;
if (obj != NULL) { // If we have a team
    buffer = obj;
    obj = next_token(buffer);
    if (buffer != NULL) {
        if (strcmp(buffer, teamname))
            players[num_players].team = TEAM_US;
        else
            players[num_players].team = TEAM_THEM;
    }
}
if (obj != NULL) { // If we have a unum
    (void)next_token(obj);
    players[num_players].unum = atoi(obj);
    // here we construct the vis_obj's Player obj ptr.
    // we know we can because we have the unum.
    // if we have the unum, we must have the team, too.
    our_index = players[num_players].unum - 1;
    if (players[num_players].team == VIS_WI_our) {
        // We want to store our guy at the same index
        // in the world as is his unum
        if (claimed_us[our_index] != -1) {
            // We have a guy whose unum we do not know
            // sitting in the spot we now wish to claim. He's gotta move
            for (int unclaimed = 0; unclaimed < 11; unclaimed++) {
                if (claimed_us[unclaimed] == -1) {
                    // We have found an unclaimed spot
                    players[claimed_us[our_index]].player =
                        world->teammates[unclaimed];
                    claimed_us[unclaimed] = claimed_us[our_index];
                    break;
                }
            }
        }
        players[num_players].player = world->teammates[our_index];
        claimed_us[our_index] = num_players;
    }
    else { // The guy is an opponent
        if (claimed_them[our_index] != -1) { // As above
            for (int unclaimed = 0; unclaimed < 11; unclaimed++) {
                if (claimed_them[unclaimed] == -1) {
                    // We have found an unclaimed spot
                    players[claimed_them[our_index]].player =
                        world->opponents[unclaimed];
                    claimed_them[unclaimed] = claimed_them[our_index];
                    break;
                }
            }
        }
        players[num_players].player = world->opponents[our_index];
        claimed_them[our_index] = num_players;
    }
}
} else if (players[num_players].team == TEAM_US) { // We don't have a unum,
    but we have a team
    //but we still have to handle cases where the unum isn't known.
    //so what we do at the moment is to just assign them to an
    //un-pointed
    //at player.
    //point them at an unclaimed guy on our team.
    for (int i=0; i<11; i++) {
        if (claimed_us[i] == -1) {
            players[num_players].player = world->teammates[i];
            claimed_us[i] = num_players; // store in the claimed_array the
            // index of the player in the player_array who claimed that spot
            // in the world

```

```

        break;
    }
    } else {
        //point them at an unclaimed guy on their team.
        for (int i=0; i<11; i++) {
            if (claimed_them[i] == -1) {
                players[num_players].player = world->opponents[i];
                claimed_us[i] = num_players;
                break;
            }
        }
    }
    } else { // We do not know the team
        if (world->unknowns[num_unknowns] ->is_not_dead) { // Then kill him for a new
            one!
                delete world->unknowns[num_unknowns];
                world->unknowns[num_unknowns] = new Player_obj();
            }
            players[num_players].player = world->unknowns[num_unknowns];
            num_unknowns++;
        }
        num_players++;
    }
    else if (!strcmp(buffer, "Player")) {
        players[num_players].fill(object);
        players[num_players].in_cone = FALSE;
        players[num_players].team = TEAM_UNKNOWN;
        players[num_players].unum = VIS_UNUM_ERR;
        players[num_players].player = world->unknowns[num_unknowns];
        num_unknowns++;
        num_players++;
    }
    else if (!strcmp(buffer, "goal")) {
        goals[num_goals].fill(object, world);
        goals[num_goals].in_cone = TRUE;
        goals[num_goals].side = VIS_SIDE_ERR;
        if (obj != NULL) {
            buffer = obj;
            obj = next_token(buffer);
            if (*buffer == 'l')
                goals[num_goals].side = VIS_SIDE_LEFT;
            else if (*buffer == 'r')
                goals[num_goals].side = VIS_SIDE_RIGHT;
            if (goals[num_goals].side != VIS_SIDE_ERR)
                goals[num_goals].goal = world->field_ptr->goals[num_goals].side-1;
            else
                goals[num_goals].goal = NULL;
        }
        num_goals++;
    }
    else if (!strcmp(buffer, "Goal")) {
        /*
        goals[num_goals].in_cone = FALSE;
        goals[num_goals].side = VIS_SIDE_ERR;
        goals[num_goals].fill(object, world);
        num_goals++;
        */
    }
    else if (!strcmp(buffer, "flag")) {
        object->in_cone = TRUE;
        if (obj != NULL) {
            buffer = obj;
            obj = next_token(buffer);
            if (*buffer == 'l') {
                if (obj != NULL) {
                    buffer = obj;
                    obj = next_token(buffer);
                    if (*buffer == 't') {
                        if (obj != NULL) {
                            buffer = obj;
                            obj = next_token(buffer);
                        }
                    }
                }
            }
        }
    }
}

```

```

if (*buffer == 'l') { //one.
    flag_index = OUT_LEFT_TOP_10_FLAG;
    //
    // flags[OUT_LEFT_TOP_10_FLAG] =
    //     new Flag_vis_obj(object, world, OUT_LEFT_TOP_10_
FLAG);
}
else if (*buffer == '2') {
    flag_index = OUT_LEFT_TOP_20_FLAG;
    //
    // flags[OUT_LEFT_TOP_20_FLAG] =
    //     new Flag_vis_obj(object, world, OUT_LEFT_TOP_20_
FLAG);
}
else if (*buffer == '3') {
    flag_index = OUT_LEFT_TOP_30_FLAG;
    //flags[OUT_LEFT_TOP_30_FLAG] =
    //new Flag_vis_obj(object, world, OUT_LEFT_TOP_30_FLAG);
}
}
else
    flag_index = TOP_LEFT_FLAG;
//flags[TOP_LEFT_FLAG] =
// new Flag_vis_obj(object, world, TOP_LEFT_FLAG);
}
else if (*buffer == 'b') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == 'l') { //one.
            flag_index = OUT_LEFT_BOTTOM_10_FLAG;
            //flags[OUT_LEFT_BOTTOM_10_FLAG] =
            // new Flag_vis_obj(object, world, OUT_LEFT_BOTTOM_10_FLAG);
        }
        else if (*buffer == '2') {
            flag_index = OUT_LEFT_BOTTOM_20_FLAG;
            //flags[OUT_LEFT_BOTTOM_20_FLAG] =
            // new Flag_vis_obj(object, world, OUT_LEFT_BOTTOM_20_FLAG);
        }
        else if (*buffer == '3') {
            flag_index = OUT_LEFT_BOTTOM_30_FLAG;
            //flags[OUT_LEFT_BOTTOM_30_FLAG] =
            // new Flag_vis_obj(object, world, OUT_LEFT_BOTTOM_30_FLAG);
        }
    }
    else
        flag_index = BOTTOM_LEFT_FLAG;
//flags[BOTTOM_LEFT_FLAG] =
//new Flag_vis_obj(object, world, BOTTOM_LEFT_FLAG);
}
else if (*buffer == '0') {
    flag_index = OUT_LEFT_ZERO_FLAG;
    //flags[OUT_LEFT_ZERO_FLAG] =
    //new Flag_vis_obj(object, world, OUT_LEFT_ZERO_FLAG);
}
}
else if (*buffer == 'r') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == 't') {
            if (obj != NULL) {
                buffer = obj;
                obj = next_token(buffer);
                if (*buffer == 'l') { //one.
                    flag_index = OUT_RIGHT_TOP_10_FLAG;
                    // flags[OUT_RIGHT_TOP_10_FLAG] =
                    // new Flag_vis_obj(object, world, OUT_RIGHT_TOP_10_FLAG);
                }
                else if (*buffer == '2') {
                    flag_index = OUT_RIGHT_TOP_20_FLAG;
                    //flags[OUT_RIGHT_TOP_20_FLAG] =
                    //new Flag_vis_obj(object, world, OUT_RIGHT_TOP_20_FLAG);
                }
            }
        }
    }
}

```

```

}
else if (*buffer == '3') {
    flag_index = OUT_RIGHT_TOP_30_FLAG;
    //flags[OUT_RIGHT_TOP_30_FLAG] =
    //new Flag_vis_obj(object, world, OUT_RIGHT_TOP_30_FLAG);
}
}
else
    flag_index = TOP_RIGHT_FLAG;
//flags[TOP_RIGHT_FLAG] =
// new Flag_vis_obj(object, world, TOP_RIGHT_FLAG);
}
else if (*buffer == 'b') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == 'l') { //one.
            flag_index = OUT_RIGHT_BOTTOM_10_FLAG;
            //flags[OUT_RIGHT_BOTTOM_10_FLAG] =
            // new Flag_vis_obj(object, world, OUT_RIGHT_BOTTOM_10_FLAG);
        }
        else if (*buffer == '2') {
            flag_index = OUT_RIGHT_BOTTOM_20_FLAG;
            //flags[OUT_RIGHT_BOTTOM_20_FLAG] =
            // new Flag_vis_obj(object, world, OUT_RIGHT_BOTTOM_20_FLAG);
        }
        else if (*buffer == '3') {
            flag_index = OUT_RIGHT_BOTTOM_30_FLAG;
            //flags[OUT_RIGHT_BOTTOM_30_FLAG] =
            // new Flag_vis_obj(object, world, OUT_RIGHT_BOTTOM_30_FLAG);
        }
    }
    else
        flag_index = BOTTOM_RIGHT_FLAG;
//flags[BOTTOM_RIGHT_FLAG] =
//new Flag_vis_obj(object, world, BOTTOM_RIGHT_FLAG);
}
else if (*buffer == '0') {
    flag_index = OUT_RIGHT_ZERO_FLAG;
    //flags[OUT_RIGHT_ZERO_FLAG] =
    //new Flag_vis_obj(object, world, OUT_RIGHT_ZERO_FLAG);
}
}
else if (*buffer == 'c') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == 't')
            flag_index = TOP_CENTER_FLAG;
        //new Flag_vis_obj(object, world, TOP_CENTER_FLAG);
        else if (*buffer == 'b')
            flag_index = BOTTOM_CENTER_FLAG;
        //flags[BOTTOM_CENTER_FLAG] =
        //new Flag_vis_obj(object, world, BOTTOM_CENTER_FLAG);
    }
    else
        flag_index = CENTER_FLAG;
//flags[CENTER_FLAG] =
// new Flag_vis_obj(object, world, CENTER_FLAG);
}
else if (*buffer == 'p') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == 'l') {
            if (obj != NULL) {

```

```

buffer = obj ;
obj = next_token(buffer) ;
if (*buffer == 't')
    flag_index = LEFT_PBOX_TOP_FLAG;
// new Flag_vis_obj(object, world, LEFT_PBOX_TOP_FLAG);
else if (*buffer == 'c')
    flag_index = LEFT_PBOX_CENTER_FLAG;
// new Flag_vis_obj(object, world, LEFT_PBOX_CENTER_FLAG);
else if (*buffer == 'b')
    flag_index = LEFT_PBOX_BOTTOM_FLAG;
// new Flag_vis_obj(object, world, LEFT_PBOX_BOTTOM_FLAG);
}
}
else if (*buffer == 'r') {
    if (obj != NULL) {
        buffer = obj ;
        obj = next_token(buffer) ;
        if (*buffer == 't')
            flag_index = RIGHT_PBOX_TOP_FLAG;
// new Flag_vis_obj(object, world, RIGHT_PBOX_TOP_FLAG);
else if (*buffer == 'c')
            flag_index = RIGHT_PBOX_CENTER_FLAG;
// new Flag_vis_obj(object, world, RIGHT_PBOX_CENTER_FLAG);
else if (*buffer == 'b')
            flag_index = RIGHT_PBOX_BOTTOM_FLAG;
// new Flag_vis_obj(object, world, RIGHT_PBOX_BOTTOM_FLAG);
}
}
}
else if (*buffer == 'g') {
    if (obj != NULL) {
        buffer = obj ;
        obj = next_token(buffer) ;
        if (*buffer == 'l') {
            if (obj != NULL) {
                buffer = obj ;
                obj = next_token(buffer) ;
                if (*buffer == 't')
                    flag_index = LEFT_GOAL_TOP_FLAG;
// new Flag_vis_obj(object, world, LEFT_GOAL_TOP_FLAG);
else if (*buffer == 'b')
                    flag_index = LEFT_GOAL_BOTTOM_FLAG;
// new Flag_vis_obj(object, world, LEFT_GOAL_BOTTOM_FLAG);
}
}
}
else if (*buffer == 'r') {
    if (obj != NULL) {
        buffer = obj ;
        obj = next_token(buffer) ;
        if (*buffer == 't')
            flag_index = RIGHT_GOAL_TOP_FLAG;
// new Flag_vis_obj(object, world, RIGHT_GOAL_TOP_FLAG);
else if (*buffer == 'b')
            flag_index = RIGHT_GOAL_BOTTOM_FLAG;
// new Flag_vis_obj(object, world, RIGHT_GOAL_BOTTOM_FLAG);
}
}
}
else if (*buffer == 't') {
    if (obj != NULL) {
        buffer = obj ;
}
}

```

```

obj = next_token(buffer) ;
if (*buffer == 'l') {
    if (obj != NULL) {
        buffer = obj ;
        obj = next_token(buffer) ;
        if (*buffer == 'l') { //one.
            flag_index = OUT_TOP_LEFT_10_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_LEFT_10_FLAG);
}
else if (*buffer == '2') {
            flag_index = OUT_TOP_LEFT_20_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_LEFT_20_FLAG);
}
}
else if (*buffer == '3') {
            flag_index = OUT_TOP_LEFT_30_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_LEFT_30_FLAG);
}
else if (*buffer == '4') {
            flag_index = OUT_TOP_LEFT_40_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_LEFT_40_FLAG);
}
else if (*buffer == '5') {
            flag_index = OUT_TOP_LEFT_50_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_LEFT_50_FLAG);
}
}
}
else if (*buffer == 'r') {
    if (obj != NULL) {
        buffer = obj ;
        obj = next_token(buffer) ;
        if (*buffer == 'l') { //one.
            flag_index = OUT_TOP_RIGHT_10_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_RIGHT_10_FLAG);
}
else if (*buffer == '2') {
            flag_index = OUT_TOP_RIGHT_20_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_RIGHT_20_FLAG);
}
}
else if (*buffer == '3') {
            flag_index = OUT_TOP_RIGHT_30_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_RIGHT_30_FLAG);
}
else if (*buffer == '4') {
            flag_index = OUT_TOP_RIGHT_40_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_RIGHT_40_FLAG);
}
else if (*buffer == '5') {
            flag_index = OUT_TOP_RIGHT_50_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_RIGHT_50_FLAG);
}
}
}
else if (*buffer == '0') {
    flag_index = OUT_TOP_ZERO_FLAG;
// new Flag_vis_obj(object, world, OUT_TOP_ZERO_FLAG);
}
}
}
else if (*buffer == 'b') {
    if (obj != NULL) {
        buffer = obj ;
}
}
}

```

```

obj = next_token(buffer) ;
if (*buffer == 'l') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == '1') { //one.
            flag_index = OUT_BOTTOM_LEFT_10_FLAG;
            //flags[OUT_BOTTOM_LEFT_10_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_LEFT_10_FLAG);
        }
        else if (*buffer == '2') {
            flag_index = OUT_BOTTOM_LEFT_20_FLAG;
            //flags[OUT_BOTTOM_LEFT_20_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_LEFT_20_FLAG);
        }
        else if (*buffer == '3') {
            flag_index = OUT_BOTTOM_LEFT_30_FLAG;
            //flags[OUT_BOTTOM_LEFT_30_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_LEFT_30_FLAG);
        }
        else if (*buffer == '4') {
            flag_index = OUT_BOTTOM_LEFT_40_FLAG;
            //flags[OUT_BOTTOM_LEFT_40_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_LEFT_40_FLAG);
        }
        else if (*buffer == '5') {
            flag_index = OUT_BOTTOM_LEFT_50_FLAG;
            //flags[OUT_BOTTOM_LEFT_50_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_LEFT_50_FLAG);
        }
    }
}
else if (*buffer == 'r') {
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == '1') { //one.
            flag_index = OUT_BOTTOM_RIGHT_10_FLAG;
            //flags[OUT_BOTTOM_RIGHT_10_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_RIGHT_10_FLAG);
        }
        else if (*buffer == '2') {
            flag_index = OUT_BOTTOM_RIGHT_20_FLAG;
            //flags[OUT_BOTTOM_RIGHT_20_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_RIGHT_20_FLAG);
        }
        else if (*buffer == '3') {
            flag_index = OUT_BOTTOM_RIGHT_30_FLAG;
            //flags[OUT_BOTTOM_RIGHT_30_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_RIGHT_30_FLAG);
        }
        else if (*buffer == '4') {
            flag_index = OUT_BOTTOM_RIGHT_40_FLAG;
            //flags[OUT_BOTTOM_RIGHT_40_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_RIGHT_40_FLAG);
        }
        else if (*buffer == '5') {
            flag_index = OUT_BOTTOM_RIGHT_50_FLAG;
            //flags[OUT_BOTTOM_RIGHT_50_FLAG] =
            //new Flag_Vis_obj(object, world, OUT_BOTTOM_RIGHT_50_FLAG);
        }
    }
}
else if (*buffer == '0') {
    flag_index = OUT_BOTTOM_ZERO_FLAG;
    //flags[OUT_BOTTOM_ZERO_FLAG] =
    //new Flag_Vis_obj(object, world, OUT_BOTTOM_ZERO_FLAG);
}
}
}
}
}
else {
    // Flag error of some kind should be here
}

```

```

//flags[] =
//new Flag_vis_obj(object, world, VIS_FP_ERR);
}
if (flag_index != -1) {
    flags[flag_index].fill(object, world, flag_index);
    num_flags++;
    //char debug_buf[30];
    //sprintf(debug_buf, "Flag: %X\n", flags [flag_index]);
    //world->debug (debug_buf);
}
}
else if (!strcmp(buffer, "Flag")) {
    //!! We have to be smart about this and figure out which flag this is
    //!! But then the compute function has to know not to use it
    //!! Or to use it only to compute the error bounds
    flags[NUM_FLAGS].fill(object);
    flags[NUM_FLAGS].in_cone = FALSE;
    flags[NUM_FLAGS].flag = NULL;
}
else if (!strcmp(buffer, "line")) {
    object->in_cone = TRUE;
    if (obj != NULL) {
        buffer = obj;
        obj = next_token(buffer);
        if (*buffer == 'l')
            lines[num_lines].fill(object, world, LEFT_LINE);
        else if (*buffer == 't')
            lines[num_lines].fill(object, world, TOP_LINE);
        else if (*buffer == 'b')
            lines[num_lines].fill(object, world, BOTTOM_LINE);
        else if (*buffer == 'r')
            lines[num_lines].fill(object, world, RIGHT_LINE);
        else {
            // Here we need intelligent line-type error handling
            // We prolly won't actually add a line to the array
            num_lines--;
        }
        num_lines++;
    }
    else {
        //!! Here we need intelligent line-type error handling
    }
    else if (!strcmp(buffer, "line")) {
        // Again, we may be able to figure out which line this is,
        // but it prolly won't do us much good, except maybe to bound
        // the error a little better
        assert(0);
        lines[num_lines].fill(object);
        lines[num_lines].in_cone = FALSE;
        lines[num_lines].line = NULL;
        num_lines++;
    }
    else {
        printf("Vis_info::fill_info: unknown obj type\n");
        delete object;
        object = NULL;
        return 0;
    }
    buffer = next;
}
}
}
/*
if ((saw_wide_info) && (world->me->view_quality != VIEW_QUALITY_HIGH)) {
    // A change view command did not get through to the server
    // Send it again, yo!!
    world->change_view_act = new Change_view_action(world, world->me->view_width, world->me->view_quality);
    #ifdef DEBUG_MODE
    fprintf(world->output_file, "!!@@@!!Forcing view back to W:%d Q:%d.\n",
            world->me->view_width, world->me->view_quality);
    #endif
}
}
}

```

```

*/
delete object;
object = NULL;
return i;
}

Vis_info::Vis_info(World *world){
    original_buffer[0] = '\0';
    timestamp = K_NOT_USED;
    num_objects = 0;
    num_players = 0;
    num_unknowns = 0;
    num_lines = 0;
    num_goals = 0;
    num_flags = 0;
    quality = VIEW_QUALITY_HIGH;
    lines = world->seen_lines;
    goals = world->seen_goals;
    players = world->seen_players;
    flags = world->seen_flags;
    ball.timestamp = K_NOT_USED;
#ifdef DEBUG_MODE
    num_visinfos++;
#endif
}

Vis_info::~Vis_info() {
    /*
    int i;

    delete ball;
    for (int i = 0; i < num_players; i++) {
        if (players[i] != NULL)
            delete players[i];
    }
    for (int i = 0; i < NUM_FLAGS+1; i++){
        // // delete flags[i];
    }
    for (int i = 0; i < num_lines; i++){
        if (lines[i] != NULL)
            delete lines[i];
    }
    for (int i = 0; i < num_goals; i++) {
        if (goals[i] != NULL)
            delete goals[i];
    }
    /*
    lines = NULL;
    goals = NULL;
    players = NULL;
    flags = NULL;
#ifdef DEBUG_MODE
    num_visinfos--;
#endif
}

void Vis_obj::compute_delta(int facing){
    double alpha = DEG2RAD * ((int)dir + facing);
    delta.x = dist * cos(alpha);
    delta.y = dist * sin(alpha);
}

// Output operators
ostream& operator<< (ostream& o, const Flag vis_obj& fv)
o << *(fv.flag) << "\t\tDist: " << fv.dist << "\tDir: " << fv.dir;
if (fv.dist_change != VIS_CHNG_ERR)
o << "\tDist chng: " << fv.dist_change;
if (fv.dist_change != VIS_CHNG_ERR)
o << "\tDir chng: " << fv.dir_change;
}

```

```

}
return o;
}

ostream& operator<< (ostream& o, const Vis_info& vi) {
    int i;

    o << "At time " << vi.timestamp << " saw ";
    if (vi.num_lines == 0)
        o << "no lines, ";
    else if (vi.num_lines == 1)
        o << "one line, ";
    else
        o << "two lines, ";
    if (vi.num_goals == 0)
        o << "no goals, ";
    else if (vi.num_goals == 1)
        o << "one goal, ";
    else
        o << "two goals, ";
    o << "and the following flags: " << endl;
    for (i = 0; i < NUM_FLAGS; i++) {
        if (vi.flags[i].timestamp != vi.timestamp) continue;
        o << vi.flags[i] << endl;
    }
    return o;
}

void fprintf_vis_info(FILE* f, Vis_info *vi) {
    int i;
    fprintf(f, "At time %d saw ", vi->timestamp);
    if (vi->num_lines == 0)
        fprintf(f, "no lines, ");
    else if (vi->num_lines == 1)
        fprintf(f, "one line, ");
    else
        fprintf(f, "two lines, ");
    if (vi->num_goals == 0)
        fprintf(f, "no goals, ");
    else if (vi->num_goals == 1)
        fprintf(f, "one goal, ");
    else
        fprintf(f, "two goals, ");
    /*
    fprintf(f, "the following flags: \n");
    for (i = 0; i < NUM_FLAGS; i++) {
        if (vi->flags[i] == NULL) continue;
        vi->flags[i]->flag->fprintf_flag_type(f);
        fprintf(f, "\n");
    }
    */
    fprintf(f, " and %d players\n\n", vi->num_players);
}

//for flags, goals, and lines, the apply fcn is not really meaningful.
//these stationary objects are only used to compute our position.
char* next_token(char *buf)
{
    if (buf == NULL) {
        // return buf;
    }
    int i = 0;
    while (1) {
        if (*buf == '(')
            i++;
    }
}

```

```

else if (*buf == ')') {
    if (i == 0) {
        *buf = '\0';
        return NULL;
    }
    i--;
} else if (*buf == ' ' && i == 0)
    break;
else if (*buf == '\0')
    return NULL;
buf++;
}

*buf = '\0';
buf++;
return buf;
}

// Out of field is a boolean value
int Line_vis_obj::compute_direction(int out_of_field)
{
    if (out_of_field == TRUE) {
        switch(line->line_type) {
            case TOP_LINE :
                return(int) (-dir - ((dir > 0) ? -180 : 0));
                break;
            case BOTTOM_LINE :
                return(int) (-dir - ((dir < 0) ? 180 : 0));
                break;
            case LEFT_LINE :
                return(int) (-dir - ((dir > 0) ? -90 : 90));
                break;
            case RIGHT_LINE :
                return(int) (-dir - ((dir < 0) ? -90 : 90));
                break;
        }
    } else {
        switch(line->line_type) {
            case TOP_LINE :
                return(int) (-dir - ((dir < 0) ? 180 : 0));
                break;
            case BOTTOM_LINE :
                return(int) (-dir - ((dir > 0) ? -180 : 0));
                break;
            case LEFT_LINE :
                return(int) (-dir - ((dir < 0) ? -90 : 90));
                break;
            case RIGHT_LINE :
                return(int) (-dir - ((dir > 0) ? -90 : 90));
                break;
        }
    }
}

Pair Line_vis_obj::compute_position(Vis_obj* vis_obj_ptr,
int vis_obj_is_goal,
int out_of_field,
int facing)
{
    double x_dist_to_obj;
    double y_dist_to_obj;

    Pair ret_val, obj_position;
    // This is the direction to the line in radians
    double rad_dir = Deg2Rad(dir);
    int absolute_obj_ang;

    // Get the object's true position
    if (vis_obj_is_goal)

```

```

obj_position = ((Goal_vis_obj*)vis_obj_ptr)->goal->position;
else
    obj_position = ((Flag_vis_obj*)vis_obj_ptr)->flag->position;

// Get and normalize the angle to the flag (or goal)
// We should now be 180 > a > -180
absolute_obj_ang = (int)vis_obj_ptr->dir + facing;
if (absolute_obj_ang > 180)
    absolute_obj_ang -= 360;
else if (absolute_obj_ang < -180)
    absolute_obj_ang += 360;

switch(line->line_type) {
    case TOP_LINE :
        // We want to compute the y coordinate from the line
        if (out_of_field)
            ret_val.y = -PITCH_WIDTH/2.0 - Abs(dist * sin(rad_dir));
        else
            ret_val.y = -PITCH_WIDTH/2.0 + Abs(dist * sin(rad_dir));

        // Find the x-component of the distance from us to the object
        x_dist_to_obj = sqrt(Abs(Sqr(vis_obj_ptr->dist) -
            Sqr(obj_position.y - ret_val.y)));

        // If we are "looking right"
        if (Abs(absolute_obj_ang) < 90)
            ret_val.x = obj_position.x - x_dist_to_obj;
        else
            ret_val.x = obj_position.x + x_dist_to_obj;
        break;

    case BOTTOM_LINE :
        // We want to compute the y coordinate from the line
        if (out_of_field)
            ret_val.y = PITCH_WIDTH/2.0 + Abs(dist * sin(rad_dir));
        else
            ret_val.y = PITCH_WIDTH/2.0 - Abs(dist * sin(rad_dir));

        // Find the x-component of the distance from us to the object
        x_dist_to_obj = sqrt(Abs(Sqr(vis_obj_ptr->dist) -
            Sqr(obj_position.y - ret_val.y)));

        // If we are "looking right"
        if (Abs(absolute_obj_ang) < 90)
            ret_val.x = obj_position.x - x_dist_to_obj;
        else
            ret_val.x = obj_position.x + x_dist_to_obj;
        break;

    case LEFT_LINE :
        // We want to compute the x coordinate from the line
        if (out_of_field)
            ret_val.x = -PITCH_LENGTH/2.0 - Abs(dist * sin(rad_dir));
        else
            ret_val.x = -PITCH_LENGTH/2.0 + Abs(dist * sin(rad_dir));

        // Find the y-component of the distance from us to the object
        y_dist_to_obj = sqrt(Abs(Sqr(vis_obj_ptr->dist) -
            Sqr(obj_position.x - ret_val.x)));

        // If we are "looking up"
        if (absolute_obj_ang > 0)
            ret_val.y = obj_position.y - y_dist_to_obj;
        else
            ret_val.y = obj_position.y + y_dist_to_obj;
        break;

    case RIGHT_LINE :
        // We want to compute the x coordinate from the line
        if (out_of_field)
            ret_val.x = PITCH_LENGTH/2.0 + Abs(dist * sin(rad_dir));
        else
            ret_val.x = PITCH_LENGTH/2.0 - Abs(dist * sin(rad_dir));

        // Find the y-component of the distance from us to the object
        y_dist_to_obj = sqrt(Abs(Sqr(vis_obj_ptr->dist) -
            Sqr(obj_position.x - ret_val.x)));
}

```

```

    ret_val.x = PITCH_LENGTH/2.0 - Abs(dist * sin(rad_dir));
    // Find the y-component of the distance from us to the object
    Y_dist_to_obj = sqrt(Abs(Sqr(vis_obj_ptr->dist) -
        Sqr(obj_position.x - ret_val.x)));

    // If we are "looking up"
    if (absolute_obj_ang > 0)
        ret_val.y = obj_position.y - Y_dist_to_obj ;
    else
        ret_val.y = obj_position.y + Y_dist_to_obj ;
    break;
}
return ret_val;
}

void Player::vis_obj::apply(World* world) {
    if (player == NULL)
        return;
    double Rmag;
    Pair V,R,W,E; // See the pseudomanual for an explanation
    double Rmag;

    player->is not dead = TRUE;
    player->team = team;
    Self_obj * me = world->me;
    V = me->velocity;
    compute_delta(me->facing);
    R = delta;
    Rmag = R.f();
    E.x = R.x/Rmag;
    E.y = R.y/Rmag;
    player->position = me->position + R;
    if ((dist_change != VIS_CHNG_ERR) && (dir_change != VIS_CHNG_ERR)) {
        W.x = dist_change * E.x - (dir_change * DEG2RAD) * Rmag * E.y;
        W.y = dist_change * E.y + (dir_change * DEG2RAD) * Rmag * E.x;
        player->velocity = V + W;
        player->velocity.error = .01 * dist;
    }
    else {
        player->velocity = ZERO_PAIR;
    }

    if (facing == VIS_DIR_ERR)
        player->facing = VIS_DIR_ERR;
    else
        player->facing = (int)normalize_deg(me->facing + facing);
    player->u.number = (int)timestamp;
    Player->timestamp = timestamp;
}

void Ball_vis_obj::apply(World* world){
    if (ball == NULL)
        return;
    Pair V,R,W,E; // See the pseudomanual for an explanation
    double Rmag;

    Self_obj * me = world->me;
    V = me->velocity;
    compute_delta(me->facing);
    R = delta;
    Rmag = R.f();
    E.x = R.x/Rmag;
    E.y = R.y/Rmag;
    ball->position = me->position + R;
    if ((dist_change != VIS_CHNG_ERR) && (dir_change != VIS_CHNG_ERR) && (dist_change
        != 0.0)) {
        fprintf(world->output_file, "SAW BALL: dist: %f, dir: %f, dist_chg: %f,
            dir_chg: %f.\n",
            //
            // dist_dir, dist_change, dir_change);
        W.x = dist_change * E.x - (dir_change * DEG2RAD) * Rmag * E.y;
        W.y = dist_change * E.y + (dir_change * DEG2RAD) * Rmag * E.x;
        ball->velocity = V + W;

```

```

    ball->velocity.error = .01 * dist;
    ball->odo velocity = ball->velocity;
    } else if (dist_change == 0.0) {
        ball->velocity = Pair(0.0, 0.0);
        ball->odo_velocity = ball->velocity;
    } else {
        ball->get_ball_odo_up_to_speed(world); // Update the velocities from odo
    }
    ball->odo.position = ball->position;
    ball->odo.timestamp = ball->timestamp;
    if (in_cone) {
        ball->see_timestamp = timestamp;
    }
}

int Vis_obj::compute_direction(int closest_obj_is_goal,
    Vis_obj *second_closest_obj,
    int second_closest_obj_is_goal) {
    Pair closest_pos, second_closest_pos;
    if (closest_obj_is_goal) {
        closest_pos = ((Goal_vis_obj*)this)->goal->position;
    } else {
        closest_pos = ((Flag_vis_obj*)this)->flag->position;
    }
    if ( second_closest_obj_is_goal) {
        second_closest_pos = ((Goal_vis_obj*)second_closest_obj)->goal->position;
    } else {
        second_closest_pos = ((Flag_vis_obj*)second_closest_obj)->flag->position;
    }
    return compute_direction_low(closest_pos,
        this->dist,
        this->dir,
        second_closest_pos,
        second_closest_obj->dist,
        second_closest_obj->dir);
}

int compute_direction_low( Pair P1, double R1, double alpha1,
    Pair P2, double R2, double alpha2) {
    double beta = DEG2RAD * (alpha1 - alpha2);
    double gamma = asin(R2 * sin(beta) / (P2-P1).r());
    double eta = asin(R1 * sin(beta) / (P2-P1).r());
    gamma = normalize_rad(pi - eta - beta);
    double zeta = (P2 - P1).angle();
    double facing_rad = normalize_rad( pi - (gamma - zeta) - DEG2RAD * alpha1);
    return (int)(RAD2DEG * facing_rad);
}

// This will compute position from a goal or a flag
Pair Vis_obj::compute_position(int new_facing, int object_is_goal) {
    Pair ret_val;
    Pair obj_pos;
    int absolute_angle; // The world angle of the object
    double absolute_angle_rad; // And the same in radians

    if (object_is_goal) {
        obj_pos = ((Goal_vis_obj*)this)->goal->position;
    } else { // Object is a flag, yo!
        obj_pos = ((Flag_vis_obj*)this)->flag->position;
    }
    absolute_angle = new_facing + (int)dir;
    absolute_angle_rad = Deg2Rad(absolute_angle);
    ret_val.x = obj_pos.x - dist * cos(absolute_angle_rad);
    ret_val.y = obj_pos.y - dist * sin(absolute_angle_rad);
    return ret_val;
}

// Fill in a Body_info with lots of data.

```

```
void World::fill_body_info (char* buf) {
  int timestamp;
  double stamina;
  double effort; //magnitude quantized by .01
  int view_mode;
  int view_quality;
  int view_width;

  int num_kick;
  int num_dash;
  int num_turn;
  int num_say;
  char vqual[6], vwid[8]; // high.low narrow.normal.wide

  // Do this mighty and strong sscanf to parse the entire result in
  // one big gulp.

  sscanf (buf, "(sense_body %d (view_mode %s %[^)]) (stamina %lf %lf) (speed %lf) (k
  ick %d) (dash %d) (turn %d) (say %d)",
          &timestamp, &vqual, &vwid, &stamina, &effort, &speed, &num_kick,
          &num_dash, &num_turn, &num_say);

  // Turn the strings into useful integers.
  if (strcmp (vqual, "high") == 0) {
    view_quality = VIEW_QUALITY_HIGH;
  } else {

    // If the data is bad, assume that the quality is low.
    view_quality = VIEW_QUALITY_LOW;
  }

  if (strcmp (vwid, "narrow") == 0) {
    view_width = VIEW_WIDTH_NARROW;
  } else if (strcmp (vwid, "wide") == 0) {
    view_width = VIEW_WIDTH_WIDE;
  } else {

    // If the data is bad, assume that the view is normal.
    view_width = VIEW_WIDTH_NORMAL;
  }

  time now = timestamp;
  me->view_width = view_width;
  me->view_quality = view_quality;
  me->body_timestamp = timestamp;

  me->stamina = (int)stamina;
  me->effort = effort;
  me->sense_body_speed = speed;

  me->num_kick = num_kick;
  me->num_dash = num_dash;
  me->num_turn = num_turn;
  me->num_say = num_say;
  me->velocity = Polar2Pair (me->sense_body_speed, DEG2RAD * me->facing_at_last_dash);
  //odo velocity = me->velocity;
  // Happy happy all done.
  return;
}
```

```

// Home includes
#include <new_action.hh>
#include <roboocup_com.hh>
#include <world.hh>
#include <roboocup_act.hh>
#include <sensor.hh>
#include <new_move.hh>

#include <libscilient.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

#include <iostream.h>
#include <sstream.h>
#include <utils.hh>

int main(int argc, char **argv) {
    FILE *outfile;
    Socket sock;
    char *server;
    int port;
    InitInfo info;
    char name[16];
    char* cmd;
    Action_Queue *aq = (Action_Queue*) NULL;

    if(argc == 6) {
        strcpy(tname, argv[1]);
        server = argv[2];
        port = atoi(argv[3]);
        //port = 6000;
        strcpy(output_fname, argv[5]);
    } else if (argc == 5) {
        strcpy(tname, argv[1]);
        server = argv[2];
        port = 6000;
        if (strcmp(argv[3], "-f") == 0) {
            strcpy(output_fname, argv[4]);
        } else {
            printf("usage: %s TEAMNAME [HOST [PORT]] [-f OUTPUT_FILENAME] \n", \
                argv[0]);
            exit(0);
        }
    } else if (argc == 4) {
        strcpy(tname, argv[1]);
        if (strcmp(argv[2], "-f") == 0) {
            server = "localhost";
        } else {
            strcpy(output_fname, argv[3]);
            port = 6000;
            server = argv[2];
            port = atoi(argv[3]);
            strcpy(output_fname, "stdout");
        }
    } else if (argc == 3) {
        strcpy(tname, argv[1]);
        server = argv[2];
        port = 6000;
        strcpy(output_fname, "stdout");
    } else if (argc == 2) {
        strcpy(tname, argv[1]);
    }

```

```

        server = "localhost";
        port = 6000;
        strcpy(output_fname, "stdout");
    } else {
        printf("usage: %s TEAMNAME [HOST [PORT]] [-f OUTPUT_FILENAME] \n", argv[0]);
        exit(0);
    }

    if (strcmp(output_fname, "stdout") == 0) {
        strcpy (output_fname, "/dev/tty");
    }

    outfile = fdopen(open(output_fname,
        O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
        S_IRWXU | S_IRWXO | S_IRWXG ), "a");
    setvbuf(outfile, (char*) NULL, _IONBF, 0);

    sock = init_connection(server, port);
    if(sock.socketfd == -1)
        exit(1);

    info = send_com_init(&sock, tname);
    fprintf(outfile, "\ninit!: %d, %d, %s, %d\n\n",
        info.unum, info.playmodestr, info.playmode);

    if (info.side == S_UNKNOWN) {
        fprintf(outfile, "Can't init\n");
        exit(-1);
    }

    World* the_world = new World(&sock);
    the_world->me->sock = &sock; // Don't know who, but this needs to be here
    // a simple init which just makes a World and fills in its socket.
    the_world->fill_world(info.side, tname, info.unum, info.playmodestr, outfile, FALS
E);

    // A new empty queue
    aq = new Action_Queue();

    randomize(); // Seed the random number generator
    // This will put us in a random position on the field
    the_world->send_initial_move();
    // Get some vis info
    Vis info* vi = NULL;
    int got_hear_info;
    while (vi == NULL)
        vi = the_world->get_info(&got_hear_info);
    delete vi;
    vi = NULL;

    // Enqueue find ball
    Find_ball *fba = new Find_ball(the_world);
    aq->insert_first(fba);

    // Move to *mifa = new Move_to(the_world, the_world->me, Pair(0.0,0.0),
    // 60.0, FORMATION_TOLERANCE, FALSE, FALSE);

    Move_to *mifa = new Move_to(the_world); // For new_move
    aq->insert_last(mifa);

    mifa->world = the_world; // Magic
    // This will poll the incoming socket and await the start of the game
    the_world->wait_for_mode_change();

    fprintf(outfile, "Player %d starting timer\n", the_world->me->u_number);
    aq->start_timer(); // This should send the commands
    UpdateInfo *ui;
    aq->update_first_action_only = TRUE;
    send_com_change_view(the_world->me->sock, VIEW_WIDTH_WIDE,
        VIEW_QUALITY_HIGH);
    while ((the_world->mode != BEFORE_KICK_OFF) &&

```

```
(the_world->mode != TIME_OVER) {
    sigpause(SIGUSR1);
    // Count counts alarms since last info
    if (the_world->alarms_since_last_info > 40) {
        close_connection(*the_world->me->sock);
        fprintf(the_world->output_file,
            "Shutting down player %d (server seems DED. . .)\n", the_world->me->u_n
umber);
        goto END;
    }
    if (aq->is_empty()) {
        cout << the_world->me->u_number << "\t" << the_world->num_dest_changes << "\t"
        << the_world->last_dest_time << "\t" << the_world->last_vis_info_time <<
endl;
        goto END;
    }
    // spin_lock(aq->q_lock);
    aq->clean_finished(); // Removes the finished actions
    vi = the_world->get_info(&got_hear_info);
    if (vi != NULL) {
        the_world->alarms_since_last_info = 0;
        the_world->last_vis_info_time = vi->timestamp;
        delete vi;
        vi = NULL;
        ui = aq->update_queue();
        if (ui != NULL) {
            delete ui;
            ui = NULL;
        }
        // release_lock(aq->q_lock);
    }
}

END:
if (the_world != NULL)
    delete the_world;
if (aq != NULL)
    delete aq;
exit(0); //return 0;
}
```

```
//some math routines we might want.
#include <math.h>
#include "trig.hh"
#include <assert.h>

double deg2rad(int theta){
    return ((double)theta * PI/180.0);
}
int rad2deg(double theta){
    return (int)(theta * 180/PI);
}

//wraps a degree value to between -180 and 180
double normalize_deg(double theta){
    int foob = (int)floor (theta / 360);
    theta -= (foob * 360);
    if (theta > 180)
        theta -= 360;
    return theta;
}

//wraps a radian value to between -PI and PI
double normalize_rad(double theta){
    assert (abs (theta) < 10000);
    int foob = (int)floor (theta / TWOPI);
    theta -= (foob * TWOPI);
    if (theta > PI)
        theta -= TWOPI;
    return theta;
}

double abs (double x) {
    return ( (x > 0.0) ? x : (-1) * x);
}

#ifdef IRIX
int ceiling (double x) {
    int y = (int)trunc (x);
    return ( (x > 0) ? y + 1 : y);
}
#endif

#ifdef Solaris
int ceiling (double x) {
    double y = (int)floor (x);
    return (int) ( (x == y) ? y : y + 1);
}
#endif

//int floor (double x) {
//    int y = trunc (x);
//    return ((x >= 0) ? y : y-1);
//}

double x_to_y (double x, int Y) {
    double count = 1;
    for (;y > 0; y--)
        count *= x;
    return count;
}

void swap (double &x, double &y) {
    double t = y;
    y = x;
    x = t;
}
```

```
}
```

```
#include <stdlib.h>
#include <sys/time.h>
#include <utils.hh>
/*
 * =====
 * Random number utility
 * =====
 */
unsigned int randomize(void)
{
    static unsigned int l ;

#ifdef PC98
    static time_t Tp ;
    time( &Tp ) ;
    l = (unsigned int)(Tp) ;
#else
    static struct timeval Tv ;
    static struct timezone Tz ;
    gettimeofday( &Tv , &Tz ) ;
    l = (unsigned int)(Tv.tv_usec) ;
#endif
    srand( l ) ;
    return l ;
}
```

```

#include <sensor.hh>
Vis_obj::Vis_obj() {
}
Player_vis_obj::Player_vis_obj() {
}
Player_vis_obj::Player_vis_obj(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    facing = base->facing;
    this->timestamp = base->timestamp;
}
void Player_vis_obj::fill(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    facing = base->facing;
    this->timestamp = base->timestamp;
}
Ball_vis_obj::Ball_vis_obj() {
}
Ball_vis_obj::Ball_vis_obj(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    this->timestamp = base->timestamp;
}
void Ball_vis_obj::fill(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    this->timestamp = base->timestamp;
}
Flag_vis_obj::Flag_vis_obj() {
}
Flag_vis_obj::Flag_vis_obj(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    flag = NULL;
    this->timestamp = base->timestamp;
}
void Flag_vis_obj::fill(Vis_obj* base) {
    in_cone = base->in_cone;
}

```

```

    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    flag = NULL;
    this->timestamp = base->timestamp;
}
Flag_vis_obj::Flag_vis_obj(Vis_obj* base,
                          World* world,
                          int flag_type) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    flag = world->field_ptr->flags[flag_type];
    this->timestamp = base->timestamp;
}
void Flag_vis_obj::fill(Vis_obj* base,
                      World* world,
                      int flag_type) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    flag = world->field_ptr->flags[flag_type];
    this->timestamp = base->timestamp;
}
/*****
Line_vis_obj::Line_vis_obj() {
}
Line_vis_obj::Line_vis_obj(Vis_obj* base, World* world, int line_type) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    line = world->field_ptr->lines[line_type];
    this->timestamp = base->timestamp;
}
void Line_vis_obj::fill(Vis_obj* base, World* world, int line_type) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    line = world->field_ptr->lines[line_type];
    this->timestamp = base->timestamp;
}
Line_vis_obj::Line_vis_obj(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    line = NULL;
    this->timestamp = base->timestamp;
}
void Line_vis_obj::fill(Vis_obj* base) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    line = NULL;
    this->timestamp = base->timestamp;
}
}

```

```
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    line = NULL;
    this->timestamp = base->timestamp;
}

/*****
Goal_vis_obj::Goal_vis_obj() {
}

Goal_vis_obj::Goal_vis_obj(Vis_obj* base, World* world) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    this->timestamp = base->timestamp;
}

void Goal_vis_obj::fill(Vis_obj* base, World* world) {
    in_cone = base->in_cone;
    dist = base->dist;
    dir = base->dir;
    dist_change = base->dist_change;
    dir_change = base->dir_change;
    this->timestamp = base->timestamp;
}
}

```

```

// **c++**
// The above line causes xemacs to open this file in c++-mode
#include <signal.h>
#include <ctype.h>
#include <sys/time.h>
#include <string.h>
// Our include
#include <sensor.hh>
#include <actions.hh>
#include <new_action.hh>
#include <zones.hh>

// Some other shupoh
#include <time.h>
#include <assert.h>

// These three are for the sleep in synchronize
#include <limits.h>
#define IRIX
#include <sgidefs.h>
#endif
#include <unistd.h>
#include <errno.h>

// New X-Module includes
#include <robocup_com.hh>

World* Global_world = NULL; //icky global-ish variable.
// Necessary for signal handling
Field::Field() {
    int i;
    for(i = 0; i < NUM_FLAGS; i++) {
        flags[i] = new Flag_obj(i);
    }

    for(i = 0; i < 4; i++) {
        //! The lines constructor may need to store more info still
        lines[i] = new Line_obj(i);
    }

    for(i = 0; i < 2; i++) {
        goals[i] = new Goal_obj(i + 1);
    }
}

Field::~Field() {
    for(int i = 0; i < NUM_FLAGS; i++) {
        delete flags[i];
        flags[i] = NULL;
    }

    for(int k = 0; k < 4; k++) {
        //! The lines constructor may need to store more info still
        delete lines[k];
        lines[k] = NULL;
    }

    for(int j = 0; j < 2; j++) {
        delete goals[j]; // The side is 1 for left, 2 for right
        goals[j] = NULL;
    }
}

World::World() {
    assert(0);
    int i;

    ball = new Ball_obj();
    me = new Self_obj();

```

```

    for (i = 0; i < 11; i++) {
        teammates[i] = new Teammate_obj();
        opponents[i] = new Opponent_obj();
    }

    for (i = 0; i < 22; i++) {
        unknowns[i] = new Player_obj();
    }

    for (i = 0; i < NUM_MSGS; i++) {
        messages[i] = NULL;
    }

    field_ptr = new Field();

    // action_q = new Action_Q(this); // Removed for X-Module
    say_act = NULL;
    change_view_act = NULL;

    latest_message = -1;
    mode = BEFORE_KICK_OFF;
    which_side = SIDE_NULL;
    our_score = 0;
    their_score = 0;
    half = 1;
    time_now = 0;
    time_left = HALF_TIME * 5; //!no clue yet why the *5 factor, but it is.

    timer_ticks = -INT_MAX;
    timer_control_on = FALSE;
    action_step_clicks = rounds = 0;
    init_initial_positions();
    queue_mode_on = FALSE;
}

World::World(Socket* sock_in) {
    int i;

    alrms_since_last_info = 0;

    ball = new Ball_obj();
    me = new Self_obj(sock_in);

    for (i = 0; i < 11; i++) {
        teammates[i] = new Teammate_obj();
        opponents[i] = new Opponent_obj();
    }

    for (i = 0; i < 22; i++) {
        unknowns[i] = new Player_obj();
    }

    for (i = 0; i < NUM_MSGS; i++) {
        messages[i] = NULL;
    }

    field_ptr = new Field();

    //action_q = new Action_Q(this);
    say_act = NULL;
    change_view_act = NULL;

    latest_message = -1;
    mode = BEFORE_KICK_OFF;
    which_side = SIDE_NULL;
    global_world = this;
    our_score = 0;
    their_score = 0;
    half = 1;
    time_now = 0;
}

```

```

time_left = HALF_TIME * 5 ;
timer_ticks = -INT_MAX;
timer_control_on = FALSE;
action_step_clicks = rounds = 0;
init_initial_positions();

// Initialize thesis output data
num_dest_changes = 0;
last_dest_time = 0;

//!set output_file?
queue_mode_on = FALSE;
}

World::~World() {
delete ball; ball = NULL;
delete me; me = NULL;
delete field_ptr; field_ptr = NULL;
// delete action_q; action_q = NULL;
// *if ( say_act != NULL)
delete say_act;
if (change_view_act != NULL)
delete change_view_act; *
}

void World::fill_world(int side,
char *team_name,
int u_num,
char *mode_string,
FILE *outfile,
int queue_mode_state) {
which_side = me->side = side; // This is S_LEFT or S_RIGHT
strcpy(me->team_name, team_name, 15);
mode = mode_string_to_number(mode_string);
me->u_number = u_num;
output_file = outfile;
queue_mode_on = queue_mode_state;
me->set_starting_position();
// zone_def = new Zone_Def(side); // Out for X-Module
}

int World::store_message(Hear_info *audio)
{
if (latest_message == -1)
latest_message = 0;
else if (latest_message == (NUM_MSGS-1))
latest_message = 0;
else
latest_message++;
if ( messages[latest_message] != NULL) {
delete messages[latest_message];
messages[latest_message] = NULL;
}
messages[latest_message] = audio;
return(latest_message);
}

void World::fprintf_messages(FILE *f)
{
int i;
if (latest_message != K_NOT_USED) {
for (i = latest_message; i >= 0; i--) {
if (messages[i] != NULL)
messages[i]->fprintf_hear_info(f);
}
for (i = NUM_MSGS - 1; i != latest_message; i--) {
if (messages[i] != NULL)
messages[i]->fprintf_hear_info(f);
}
}
}
}

```

```

Vis_info* World::get_info(int *got_hear_info) {
char buffer[VISBUFSIZE];
char vis_buffer[VISBUFSIZE];
vis_buffer[0] = '\0';
char body_buffer[1024];
int size = sizeof(buffer);
Hear_info* hear_info = NULL;
Vis_info* vis_info = NULL;
// Body info* body_info = NULL; // * goes on the variable type, darnit
// NO, actually it doesn't, cause you can do things like
// int* int_ptr, i; The * in that case would only apply to the int_ptr variable
// So there.
char info_type[16];
int got_vis_info, got_body_info;
*got_hear_info = FALSE;
got_vis_info = FALSE;
got_body_info = FALSE;
while (1) {
if (receive_message(me->sock, buffer, size)) {
// Test to see whether the message is "see", "hear",
// or "sense_body"
sscanf(buffer, "%s ", info_type);
if (!strcmp(info_type, "see")) {
memcpy(vis_buffer, buffer, VISBUFSIZE);
got_vis_info = TRUE;
}
if (!strcmp(buffer, "see")) {
if (!receive_message(me->sock, buffer+4, size-4)) {
fprintf(output_file, "I wanted to, but I just couldn't to do it.
buffer:%s\n", buffer);
} else {
fprintf(output_file, "I wanted to, and I did! buf:%s\n", buffer);
}
}
} else if (!strcmp(info_type, "hear")) {
hear_info = new Hear_info(buffer);
}
}
// This is a HACK. This should be handled more elegantly,
// at least in a separate function
if (hear_info->sender == SENDER_REFEREE) {
if (!strcmp(hear_info->message, "kick_off_1"))
mode = KICK_OFF_L;
if (!strcmp(hear_info->message, "kick_off_r"))
mode = KICK_OFF_R;
if (!strcmp(hear_info->message, "play_on"))
mode = PLAY_ON;
}
store_message(hear_info);
*got_hear_info = TRUE;
} else if (strcmp (info_type, "sense_body") == 0) {
// Make a new body o
memcpy(body_buffer, buffer, 1024);
got_body_info = TRUE;
} else if (strcmp (info_type, "error") == 0) {
// Some unknown string. Perhaps we should do something different here
// ignore this string
#ifdef DEBUG_MODE
fprintf(output_file, "NODA says illegal!\n");
#endif
} else {
#ifdef DEBUG_MODE
fprintf(output_file, "BAD INFO: %s.\n", buffer);
assert(0);
#endif
return NULL;
}
} else
break; // At the end of the messages
}
}

```



```

delete vi2;
vi2 = NULL;
// vi = vi2;
}
// After this procedure vi is the latest vis_info we got.
// if (me->body.timestamp != K_NOT_USED) { // We got a new sense_body
time_now = me->body.timestamp;
// if (got_vis_info != NULL)
// if ((vi != NULL) && (vi->timestamp == me->body.timestamp)) {
// } else {
// if (got_vis_info != NULL) *got_vis_info = TRUE;
// }
// if (got_vis_info != NULL) *got_vis_info = FALSE;
// }
//delete vi;
return me->body.timestamp;
// } else { // We did not get a new sense_body
// if (vi != NULL)
// delete vi;
//me->body.timestamp = save_time_stamp;
//return K_NOT_USED;
// }
}

void World::main_loop() {
// dummy, Yeah?
}

int World::parse_message_array (int current_time) {
int earliest_message = (latest_message + 1) % NUM_MSGS;
int i;
// char mode_str[32];

for (i = earliest_message ; i < NUM_MSGS; i++) {
if (messages[i] == NULL)
continue;
else {
parse_message(i);
}
}
for (i = 0; i < earliest_message ; i++) {
if (messages[i] == NULL)
continue;
else {
parse_message(i);
}
}
return i;

void World::parse_message(int index) {
int matches;
int code, time_of_send;
int target_unum;
int src_unum, src_facing;
double kick_power, kick_angle;
Pair src_pos, target_point;
//char mode_str[64];

if (messages[index]->sender == SENDER_REFEREE) {
// sscanf("referee %s)", mode_str);
mode = mode_string_to_number(messages[index]->message);
delete messages[index];
messages[index] = NULL;
return;
}
if (messages[index]->sender == SENDER_SELF) {
delete messages[index];
messages[index] = NULL;
return;
}
}

```

```

if (messages[index]->sender == SENDER_UNKNOWN) {
matches = sscanf(messages[index]->message,
"%d DRB %d %f %f %d %d %f %f %f %f",
&code, &target_unum, &kick_unum, &kick_power, &kick_angle,
&src_unum, &src_facing, &src_pos.x, &src_pos.y,
&target_point.x, &target_point.y);
if ((matches == 10) && (target_unum == me->u_number)) {
me->should_recieve_pass = TRUE;
me->angle_to_turn = messages[index]->dir;
}
delete messages[index];
messages[index] = NULL;
return;
}
}

//! NIY, this 'simulates the simulator' to update all the state.
void World::step() {
if (time_left != 0) {
time_now++;
time_left--;
//! NIY
ball->step();
me->step();
for (int x=0; x<11; x++){
teammates[x]->step();
opponents[x]->step();
}
}
// } else {
// time_now++;
// time_left = HALF_TIME * 5;
// }
}

void World::fprintf_moving_objs() {
fprintf(output_file, "Saw some of our guys\n");
int i;
for (i = 0; i < 11; i++) {
if (( teammates[i]->position.x == 0) &&
(teammates[i]->position.y == 0) &&
(teammates[i]->timestamp == 0))
continue;
if ( teammates[i]->u_number != VIS_UNUM_ERR ) {
fprintf(output_file, "Number %d\tat time %d at (%f,%f), moving at (%f,%f), fac
ing %d\n",
teammates[i]->u_number,
teammates[i]->timestamp,
teammates[i]->position.x,
teammates[i]->position.y,
teammates[i]->velocity.x,
teammates[i]->velocity.y,
teammates[i]->facing);
}
}
}
fprintf(output_file, "A teammate\tat time %d at (%f,%f), moving at (%f,%f)\n"
,
teammates[i]->timestamp,
teammates[i]->position.x,
teammates[i]->position.y,
teammates[i]->velocity.x,
teammates[i]->velocity.y,
teammates[i]->velocity.y);
}
}
fprintf(output_file, "Saw some of their guys\n");
for (i = 0; i < 11; i++) {
if (( opponents[i]->position.x == 0) &&
(opponents[i]->position.y == 0) &&
(opponents[i]->timestamp == 0))
continue;
if ( opponents[i]->u_number != VIS_UNUM_ERR ) {
fprintf(output_file, "Number %d\tat time %d at (%f,%f), moving at (%f,%f), fac
ing %d\n",

```

```

    opponents[i] -> su_number,
    opponents[i] -> timestamp,
    opponents[i] -> position.x,
    opponents[i] -> position.y,
    opponents[i] -> velocity.x,
    opponents[i] -> velocity.y,
    opponents[i] -> facing;
} else {
    fprintf(output_file, "A baddie\tat time %d at (%f,%f), moving at (%f,%f),\n",
            opponents[i] -> timestamp,
            opponents[i] -> position.x,
            opponents[i] -> position.y,
            opponents[i] -> velocity.x,
            opponents[i] -> velocity.y);
}
}
fprintf(output_file, "And the ball\tat time %d at (%f,%f) moving at (%f,%f),\n",
        ball -> timestamp,
        ball -> position.x,
        ball -> position.y,
        ball -> velocity.x,
        ball -> velocity.y);
fprintf(output_file, "++++\n");
}
}

void World::debug(char* msg) {
#ifdef DEBUG MODE
    fprintf(output_file, "%s", msg);
#endif
}

int mode_string_to_number(char* mode_name) {
    if (!strcmp(mode_name, PLAY_ON_STR)) {
        return PLAY_ON;
    } else if (!strcmp(mode_name, BEFORE_KICK_OFF_STR)) {
        return BEFORE_KICK_OFF;
    } else if (!strcmp(mode_name, TIME_OVER_STR)) {
        return TIME_OVER;
    } else if (!strcmp(mode_name, KICK_OFF_L_STR)) {
        return KICK_OFF_L;
    } else if (!strcmp(mode_name, KICK_OFF_R_STR)) {
        return KICK_OFF_R;
    } else if (!strcmp(mode_name, KICK_IN_L_STR)) {
        return KICK_IN_L;
    } else if (!strcmp(mode_name, KICK_IN_R_STR)) {
        return KICK_IN_R;
    } else if (!strcmp(mode_name, FREE_KICK_L_STR)) {
        return FREE_KICK_L;
    } else if (!strcmp(mode_name, FREE_KICK_R_STR)) {
        return FREE_KICK_R;
    } else if (!strcmp(mode_name, CORNER_KICK_L_STR)) {
        return CORNER_KICK_L;
    } else if (!strcmp(mode_name, CORNER_KICK_R_STR)) {
        return CORNER_KICK_R;
    } else if (!strcmp(mode_name, GOAL_KICK_L_STR)) {
        return GOAL_KICK_L;
    } else if (!strcmp(mode_name, GOAL_KICK_R_STR)) {
        return GOAL_KICK_R;
    } else if (!strcmp(mode_name, GOAL_L_STR)) {
        return GOAL_L;
    } else if (!strcmp(mode_name, GOAL_R_STR)) {
        return GOAL_R;
    } else {
        return PLAY_ON; // Is this a good default?
    }
}

int World::see_ball_now() {
    assert(last_vis_info_time >= ball -> see_timestamp);
    return (last_vis_info_time == ball -> see_timestamp);
}

```

```

/*
Post_info* World::retrieve_post_info(Action_obj* action) {
    Post_info* ret_pi;
    if ((action == NULL) ||
        (action -> post_info == NULL) ||
        (action -> post_info -> expected_round < last_vis_info_time)) {
        ret_pi = new Post_info();
    } else {
        ret_pi = new Post_info(action -> post_info);
    }
    ret_pi -> complete_nulls(this);
    return ret_pi;
}

*/
void World::init_initial_positions() {
    // Goalie
    initial_position[0] = Pair(-PITCH_LENGTH/2 + PENALTY_AREA_LENGTH - 3, 0);
    // Defense
    initial_position[1] = Pair(-PITCH_LENGTH/4, -23);
    initial_position[2] = Pair(-PITCH_LENGTH/4 + 5, -12);
    initial_position[3] = Pair(-PITCH_LENGTH/4 + 5, 12);
    initial_position[4] = Pair(-PITCH_LENGTH/4, 23);
    // Midfield
    initial_position[5] = Pair(-CENTER_CIRCLE_R + 2, -20);
    initial_position[6] = Pair(-CENTER_CIRCLE_R, 0);
    initial_position[7] = Pair(-CENTER_CIRCLE_R + 2, 20);
    // Offense
    initial_position[8] = Pair(0, -15);
    initial_position[9] = Pair(-1, 0);
    initial_position[10] = Pair(0, 15);
}

/*
#If 0 // Out for X-Module
int World::send_initial_move() {
    Pair move_to = initial_position[me -> su_number - 1];
    me -> command_move(this, move_to.x, move_to.y); // Removed for X-Module
    return 1;
}
#endif

*/
// This sends the player to a random position
int World::send_initial_move() {
    Command_move* cm = new Command_move(this,
        drand(-PITCH_LENGTH/2.0, PITCH_LENGTH/2.0),
        drand(-PITCH_WIDTH/2.0, -PITCH_WIDTH/2.0));
    cm -> send_command();
    delete cm;
}
}

```

```

#ifdef ACTION_DOT_HH
#define ACTION_DOT_HH
#define STEAL_RADIUS 5.0

#include "world.h"
#include "self.h"

// The four command types
#define COMMAND_ERROR -1
#define MOVE_COMMAND 0
#define TURN_COMMAND 1
#define KICK_COMMAND 2
#define DASH_COMMAND 3
#define SAY_COMMAND 4
#define SENSE_BODY_COMMAND 5
#define CHANGE_VIEW_COMMAND 6
#define ACTION_COMMAND 7
#define EMPTY_COMMAND 8
#define HALT_COMMAND 9
#define CATCH_COMMAND 10

#define MOVE_ACTION 0
#define TURN_ACTION 1
#define KICK_ACTION 2
#define DASH_ACTION 3
#define SAY_ACTION 4
#define SENSE_BODY_ACTION 5
#define CHANGE_VIEW_ACTION 6
#define MOVE_TO_ACTION 7
#define INTERCEPT_ACTION 8
#define PASS_ACTION 9
#define SHOOT_ACTION 10
#define WATCH_BALL_ACTION 11
#define TURN_BALL_ACTION 12
#define CATCH_ACTION 13

class Action_obj;

class Post_info {
public: // ALWAYS use public
Action_obj *parent_action;
int parent_action_type;
Pair *my_pos;
double *my_facing; // int
Pair *my_vel;
Pair *ball_pos;
Pair *ball_vel;
int expected_round;

Post_info();
Post_info(Post_info *pi); // Copy MOO MOO
Post_info(Action_obj *parent);
~Post_info(); // Destroy OOM
void clear();
void init(void);
void set_my_vel(Pair *my_v);
void set_my_pos(Pair *my_p);
void set_my_facing(double *my_f);
void set_ball_vel(Pair *ball_v);
void set_ball_pos(Pair *ball_p);
void fill(World *world);
void Post_info::complete_nulls(World *world);
};

class Command_obj {
public:
int command_type;
double arg_a;
double arg_b;
char arg_text[MAX_MSG_LENGTH];
Command_obj();

```

```

Command_obj(int type, double arg_a, double arg_b);
Command_obj(int type, char* msg); // this is for say commands.
Command_obj(Action_obj *action); // for actions as commands
Command_obj *send_command(World *world);

~Command_obj();
void fprintf_command(FILE *f);

Action_obj *action;
Command_obj *next;
Command_obj *prev;
char sent;
};

class Command_list {
public:
Command_obj *top;
int command_count;
int unsent_command_count;
Command_obj *last;
Command_obj *nextup;

Command_list(World *w, Command_obj *cmd);
Command_list();
~Command_list();
void empty();
Command_obj *send_next_command(World *w);
Command_obj *append_command(World *w, Command_obj *cmd);
Command_obj *insert_next_command(World *w, Command_obj *cmd);
Command_obj *insert_before(World *w, Command_obj *before, Command_obj *cmd);
void delete_command(Command_obj *obj);
};

// Abstract base class for actions
class Action_obj {
public:
//next 3 variables are replaced by command_list type
// Command_obj **command_list; // Array
//int num_commands; // This is the number of the commands in the whole array
//int next_command;
Command_list command_list;
int action_type;
char name[128]; // For output purposes only

Post_info *post_info;
World *world;
Action_obj *next;
Action_obj *prev;

Action_obj();
virtual Command_obj *send_next_command();
int nextup_is_cvs();
int is_finished();
// Some utilities for examining the command list need
// to be added here.

virtual int update ( void );

virtual void fprintf_action(FILE *output_file);
virtual ~Action_obj();
};

class Move_action : public Action_obj {
private:
double x_coord;
double y_coord;
public:
Move_action(World *w, double x, double y);
// int check(World * world);
};

class Dash_action : public Action_obj {

```

```

private:
  double power;
public:
  Dash_action(World* w, double pow);
  // int check(World* world);
};

class Turn_action : public Action_obj {
  double angle;
public:
  Turn_action (World* w, double ang);
  // int check(World* world);
};

class Kick_action : public Action_obj {
public: //protected: doesn't allow debugger to see it
  double angle;
  double power;
  Kick_action();
public:
  void fill_kick_action (World* w, double pow, double ang);
  Kick_action(World* w, double pow, double ang);
  // int check(World* world)
};

class Catch_action : public Action_obj {
public: //protected: doesn't allow debugger to see it
  int angle;
  Catch_action();
public:
  void fill_catch_action (World* w, int ang);
  Catch_action(World* w, int ang);
};

class Say_action : public Action_obj {
private:
  char msg[MAX_SAY_LENGTH];
public:
  Say_action(World* w, char* msg_in);
  // int check(World* world)
};

class Sense_body_action : public Action_obj {
private:
public:
  Sense_body_action(World* w);
  // int check(World* world)
};

class Change_view_action : public Action_obj {
public:
  Change_view_action(World* w, int width, int quality);
};

// A function used by should get ball to compute how much adv we need
// From how far away the ball is
int compute_great_advantage(double dist_to_ball);

class Watch_ball_action : public Action_obj {
public:
  Self_obj *s;
  int missing_ball;
  int patience;
  void do_watch_ball (World *w, Self_obj *s);
  void find_angle_to_seen_ball (Pair *ball_position, Pair *ball_velocity, Pair *us_p
osition,
                                Pair *us_velocity, double facing);

```

```

void cant_see_ball_panic ();
virtual int update();
/*
  if missing_ball starts at zero, we wait at current facing
  if the ball was in our cone of vision last time we saw it, and it
  might just be too far away to see. We will wait at our current facing
  for a patience rounds number of rounds before getting into a tiff
  about finding the ball.
  if urgent is 1, we go into a tiff right away
  */
  Watch_ball_action (World *w, Self_obj *s, int urgent, int patience_rounds);
};

class Action_Q {
private:
  Action_obj *last_obj; //Is this necessary?
  // It's nice. With it we do not have to traverse the whole list when we enqueue
  int num_actions;
  int queue_locked;
public:
  Action_obj *first_obj;
  Action_Q();
  Action_Q(World* w);
  ~Action_Q();
  void wipe(); //empties queue
  int enqueue(Action_obj *action);
  int insert_first(Action_obj *action);
  int enqueue_before_last(Action_obj* action);
  Command_obj* dequeue();
  /* The index of the first action is zero */
  Action_obj* peek(int index);
  int remove_actions(int start_index);
  int size();
  void fprintf_queue(FILE *output_file);
  void remove_first_action ();
  Action_obj *next_unfinished_action ();
  /* update - runs update on first object in queue */
  void update ();
};
#endif

```

```

#define COMMAND_DOT_HH
#define COMMAND_DOT_HH
// #include <stdlib.h>
#include <assert.h>
#include <iostream.h>
#include <abi_mutex.h>

#define FALSE 0
#define TRUE 1

#define COMMAND_ABC 0
#define COMMAND_WAIT 1
#define COMMAND_NULL 2
// This is the abstract base class for a command
class Command_obj {
public:
    Command_obj *next;
    Command_obj *prev;
    Command_obj();
    virtual ~Command_obj();
    virtual int send_command() {assert(0);}; // const = 0;
    virtual void print();
    virtual int command_type() {return COMMAND_ABC;};
};

// A pair of very simple but potentially useful commands
// Neither actually does anything, however
// A Wait command will simply sit on the action list and
// cause it to do nothing until it is removed from the outside
// (presumably in an update)
// A null command, on the other hand will do nothing over
// a single timer interval, and then the next_up pointer
// will be advanced
class Command_wait : public Command_obj {
public:
    Command_wait() {next = prev = NULL;};
    virtual ~Command_wait(){};
    virtual int send_command() {return 0;};
    virtual int command_type() {return COMMAND_WAIT;};
};

class Command_null : public Command_obj {
public:
    Command_null() {next = prev = NULL;};
    virtual ~Command_null(){};
    virtual int send_command() {return 1;};
    virtual int command_type() {return COMMAND_NULL;};
};

class Command_list {
private:
    Command_obj *first;
    Command_obj *last;
    Command_obj *next_to_send;
    void null_all();
    void insert_into_blank_list(Command_obj *com);
    abilock_t *list_lock;
public:
    Command_list();
    ~Command_list();
    void remove_all();
    Command_obj *send_next_command ();

    void insert_last (Command_obj *com);
    void insert_first (Command_obj *com);
    void insert_as_next (Command_obj *com);
    int is_empty ();
    int is_finished ();
    int is_in_list (Command_obj *com);

```

```

int insert_before (Command_obj *before, Command_obj *com);
int delete_command (Command_obj *com);
void print();
Command_obj *get_next();
int advance_next(int num_advance);
Command_obj *find_with_func(Command_obj *com,
    void *compare_fn(Command_obj *,Command_obj*));
void check_list();
void *Apply( void *applied_fn(Command_obj*));
};
#endif

```

```

#define MOVETO_DOT_HH
#define MOVETO_DOT_HH

#include <action.hh>
/* Moveto_abs_action - action which turns and dashes a player
 * * to within a tolerance distance of an absolute pos on the field,
 * * WARNING - being within the tolerance of the target position circle will create th
 * is action
 * * with a empty command list
 * */

class Moveto_abs_action : public Action_obj {
public:
    Moveto_abs_action ();
    virtual ~Move to() {};
    virtual int is_finished() {return FALSE;};
    virtual void print() {cout << "Move action" << endl;};
    Pair D_last_pos;
    int D_last_round;

    int backwards;
    int backwards_turn_round;
    int watch_ball_too;

    Pair destination; //absolute position to reach
    Pair displacement; //vector from current position to destination
    Pair tangent_disp; //tangential displacement
    //spot we should run towards in order to reach destination,
    //considering tangential velocity */
    Pair disp_minus_tangent;
    // parameter for how close we need to be to destination
    // (creates a destination circle)
    double tolerance;
    int max_power; //max power willing to spend
    Pair dest_destination; //where within tolerance circle we think we'll be
    int missing_turn;
    int less_wasteful_dashes;
    int dest_might_change;

    Self_obj *s; //needs to be stored? we do need to get back facing, velocity, etc f
    rom self class

    Moveto_abs_action (World* w, Self_obj *s, Pair *dest, double max_power, double tol
_dist);
    Moveto_abs_action (World* w, Self_obj *s, Pair *dest, double max_power, double tol
_dist,
                        int less_wasteful_dashes, int dest_might_change);
    Moveto_abs_action (World* w, Self_obj *s, Pair *dest, double max_power, double tol
_dist,
                        int less_wasteful_dashes, int dest_might_change, int watch_ball
_too);

    /* current_speed_for_dashes - amount to base decision on how fast to run
     * comes from this function, basically a transform of our current velocity */
    double current_speed_for_dashes (Pair &velocity, double facing);

    /* recompute existing dashes - redoes all unspent dashes as in fill */
    Command_obj * recompute_existing_dashes (double *dist_remaining, double *radial_sp
eed, double max_power);

    /* delete_remaining_commands - empties command list from first to go to end */
    void delete_remaining_commands (Command_obj *first_to_go);

    virtual void fprintf action (FILE *output file);
    void fill (World* w, Self_obj *s, Pair *dest, double max_power, double tol_dist);
    int run_backwards_to_see_ball (int watch_ball);
    int decide_backwards_when_seeing_ball (Pair &ball_pos, double facing);

    int detect_collision (World* world, Pair self_pos, Pair self_vel, double self_spee

```

```

d, double facing);
    int detect_collision_one_obj (Pair colls_pos, Pair colls_vel, Pair self_pos, Pair
self_vel,
                                double self_speed, double self_faci
ngr);
    int detect_collision_pt (Pair &pos1, Pair &pos2);

    /* compute needed angle - computes how much we need to turn in order to hit destin
ation exactly
    returns 1 if we reach target circle, 0 if not
    sweep_tolerance, 0: +/- allowable to angle of displacement so that we hit targe
t circle
    turn_angle, 0: angle needed to turn to bring us perfectly on course
    char compute_needed_angle (double *sweep_tolerance, double *turn_angle);
    /* figures out if we should immediately turn in order to optimally get to
the destination point
    PostConds: if needed, a turn is inserted at the start of the command list
    missing_turn is ste to 1 if we think we'll need to turn later
    */
    void make_imm_turn (double distance_remaining);

    /* compute_needed_dashes - adds dashes to the command list until we reach destinat
ion. It takes a
    starting speed in the radial direction and the distance to go. Uses class vari
able tolerance.
    Will not create a dash with power greater than max_power, but will create one w
ith less power if
    the larger power will only waste radial velocity to the max speed cap
    */
    void compute_needed_dashes (double *dist_remaining, double radial_speed, double ma
x_power);

    /* goes transformation from max power to speed this power will result in if we das
h at this
    power for an infinite amount of time */
    double compute_runat_speed_from_maxpower (double power);

    /* compute_best_power - returns greatest dash power <= max_power, such that radial
_speed does not go above
    speed maximum after the dash is executed
    */
    double compute_best_power (double radial_speed, double max_power);
    int okay_run_without_turning (double facing, Pair &self_pos, Pair &self_vel);

    void compute_tangential_displacement (); /* computes disp_minus_tangent and tangen
t_disp field; */
    int update_calculation ();
    virtual int update ();
};
#endif

```

```

#ifdef NEW_ACTION_DOT_HH
#define NEW_ACTION_DOT_HH

#include "command.hh"
#include <signal.h>
#include <ctype.h>
#include <sys/time.h>

// This is a forward declaration of something
// that is to be defined at instantiation
class State;

#define SECRET_CODE 11

// Forward declaration needed for Action
class Action_Queue;

class Action {
public:
    // An array of pointers to command lists
    Command_list **c_lists;
    void *pre_state;
    void *post_state;
    int action_type;

    Action_Queue* aq; // The action queue that the action is part of
    // Declared void because of fwd declaration problems

    Action *prev;
    Action *next;

    int action_done; // An action can indicate that it has completed successfully
    int num_command_lists;
    Action(int num_lists = 1);
    virtual ~Action();
    // This will attempt to acquire the pre_state structure for the action
    // from the previous action or from the action queue if necessary
    // Note that the pre_state is just a pointer to a data structure that
    // is stored elsewhere. Modifying it is not the business of this action
    void acquire_pre_state_ptr();
    Command_obj *send_next_command_from_list(int list_id);
    virtual int is_finished(int list_id = 0); // No longer used
    virtual int update() {return 0;}; // Default returns success
    virtual void print();
    // UPDATE RETURN VALUES
    //0 => Action is in progress and there is no error
    //1 => It is currently impossible to update this action
    // and/or calculate post_state (check again later)
    //2 => This action needs to be rebuilt from scratch
    //3 => This action cannot complete with the current post_state
    //4 => This action needs to be removed from the queue
    //11 => Secret code. Re-do a move-to action
};

class Update_info {
public:
    Update_info();
    ~Update_info();
    int is_filled();

    int code;
    Action *last_action;
};

class Action_Queue {
private:
    Action *first;
    Action *last;
    State *world_state;

```

```

// These structures run the queue's timer
struct itimerval itv;
struct sigaction alarm_action;

void insert_into_blank_list(Action *act);
public:
    int update_first_action_only;
    int timer_cycle;
    abilock_t *q_lock;

    Action_Queue();
    ~Action_Queue();
    void insert_last(Action *act);
    void insert_first(Action *act);
    int insert_before(Action *before, Action *act);
    int is_empty();
    int is_in_list(Action *act);
    int remove_action(Action *act);
    Action *get_first() {return first;};
    // int update_first_queue();
    // int update_first_Only();
    send_next_command_from_list(int list_id);
    void clean_finished();
    void start_timer();
    int send_next_command();
};

#endif

```

```
#ifndef MOVE_TO_DOT_HH
#define MOVE_TO_DOT_HH
#include <world.hh>
#include <new_action.hh>

//f 0
class Move_to : public Action {
private:
    Pair formation_positions[PLAYERS_IN_FORMATION];
    int position_taken[PLAYERS_IN_FORMATION];
    float position_dist[PLAYERS_IN_FORMATION];
    int destination_idx;
    int formation_positions_set;
public:
    World *world;
    Move_to() { world = NULL; };
    virtual ~Move_to() {};
    virtual int is_finished() {return FALSE;};
    virtual void print() {cout << "Moveto" << endl;};
    Pair destination;

    Self obj *s; //needs to be stored? we do need to get back facing, velocity, etc f
    rom self class
    Move_to:(Move_to(World *w);
    virtual int update ();
    void grab_closest();
    void set_player_closest_position(int);
    // int get_closest_free_pos_index(Pair pos);
    void compute_destination();
    Pair get_dest();
    int move_to_complete();
};
//endif
#endif
```

```

// *-c++*-
// The above line causes xemacs to open this file in c++-mode
#ifdef OBJECT_H
#define OBJECT_H

#include <limits.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>

#include "pair.hh"
#include "libscclient.h"
#include "param.hh"

#define K_NOT_USED -1
#define NUM_FLAGS 53
// Here are the constants for the different types of flags
// The 12 old flags (pre server version 4.0)
#define TOP_LEFT_FLAG 0
#define TOP_RIGHT_FLAG 1
#define BOTTOM_LEFT_FLAG 2
#define BOTTOM_RIGHT_FLAG 3
#define TOP_CENTER_FLAG 4
#define BOTTOM_CENTER_FLAG 5
#define LEFT_PBOX_TOP_FLAG 6
#define LEFT_PBOX_BOTTOM_FLAG 7
#define LEFT_PBOX_CENTER_FLAG 8
#define RIGHT_PBOX_TOP_FLAG 9
#define RIGHT_PBOX_BOTTOM_FLAG 10
#define RIGHT_PBOX_CENTER_FLAG 11
#define PROX_CENTER_FLAG 7

// The gajillion and a half new flags
#define CENTER_FLAG 12
#define LEFT_GOAL_TOP_FLAG 13
#define LEFT_GOAL_BOTTOM_FLAG 14
#define RIGHT_GOAL_TOP_FLAG 15
#define RIGHT_GOAL_BOTTOM_FLAG 16

// The new flags that are outside the field
#define OUT_TOP_ZERO_FLAG 17
#define OUT_TOP_LEFT_10_FLAG 18
#define OUT_TOP_LEFT_20_FLAG 19
#define OUT_TOP_LEFT_30_FLAG 20
#define OUT_TOP_LEFT_40_FLAG 21
#define OUT_TOP_LEFT_50_FLAG 22
#define OUT_TOP_RIGHT_10_FLAG 23
#define OUT_TOP_RIGHT_20_FLAG 24
#define OUT_TOP_RIGHT_30_FLAG 25
#define OUT_TOP_RIGHT_40_FLAG 26
#define OUT_TOP_RIGHT_50_FLAG 27
#define OUT_BOTTOM_ZERO_FLAG 28
#define OUT_BOTTOM_LEFT_10_FLAG 29
#define OUT_BOTTOM_LEFT_20_FLAG 30
#define OUT_BOTTOM_LEFT_30_FLAG 31
#define OUT_BOTTOM_LEFT_40_FLAG 32
#define OUT_BOTTOM_LEFT_50_FLAG 33
#define OUT_BOTTOM_RIGHT_10_FLAG 34
#define OUT_BOTTOM_RIGHT_20_FLAG 35
#define OUT_BOTTOM_RIGHT_30_FLAG 36
#define OUT_BOTTOM_RIGHT_40_FLAG 37
#define OUT_BOTTOM_RIGHT_50_FLAG 38
#define OUT_RIGHT_ZERO_FLAG 39

```

```

#define OUT_RIGHT_TOP_10_FLAG 40
#define OUT_RIGHT_TOP_20_FLAG 41
#define OUT_RIGHT_TOP_30_FLAG 42
#define OUT_RIGHT_BOTTOM_10_FLAG 43
#define OUT_RIGHT_BOTTOM_20_FLAG 44
#define OUT_RIGHT_BOTTOM_30_FLAG 45
#define OUT_LEFT_ZERO_FLAG 46
#define OUT_LEFT_TOP_10_FLAG 47
#define OUT_LEFT_TOP_20_FLAG 48
#define OUT_LEFT_TOP_30_FLAG 49
#define OUT_LEFT_BOTTOM_10_FLAG 50
#define OUT_LEFT_BOTTOM_20_FLAG 51
#define OUT_LEFT_BOTTOM_30_FLAG 52
// Here are the constants for the line types
#define NUM_LINES 4
#define TOP_LINE 0
#define BOTTOM_LINE 1
#define LEFT_LINE 2
#define RIGHT_LINE 3
#define SIDE_NULL 0
#define LEFT 1
#define RIGHT 2
#define TEAM_UNKNOWN 0
#define TEAM_US 1
#define TEAM_THEM 2

class Facing {
public:
    int facing;
    int err_up;
    int err_down;
    Facing();
    Facing(int dir);
};

class State_data {
public:
    Pair position;
    Pair velocity;
    Facing facing;
    int timestamp;

    State_data();
};

//! how do we handle openness? is that stored as a list with each object, or a disjunct
//! matrix elsewhere? is it even stored at all?
//this class is the ABC for on-field objects.
//it is extended by various inheritors which represent more specific objects.
class Object {
public:
    Pair position;
};

//also ABC

```

```

class Moving_obj : public Object {
public:
    int see_timestamp;

    Moving_obj ();
    //odo data
    Pair odo_position;
    Pair odo_velocity;
    int odo_facing;
    int odo_timestamp;
    double odo_accuracy;

    //combined data. (position inherited from Object.)
    Pair velocity;
    int facing;
    int timestamp;
    double accuracy;
    void step();
};

//also ABC
class Stationary_obj : public Object {
};

class Line_obj : public Stationary_obj {
public:
    int line_type; //should be one of the #defined types above.
};

Line_obj (int type);
};

class Flag_obj : public Stationary_obj {
public:
    int flag_type; //should be one of the #defined types above.
    Flag_obj(int type); // Constructor
    int is_outside_flag();
    void fprintf_flag_type(FILE* f);
};

extern ostream& operator<<(ostream &o, const Flag_obj& f);

class Goal_obj : public Stationary_obj {
public:
    int side;
};

Goal_obj (int side);
};

//ABC
class World;
class Player_obj : public Moving_obj {
public:
    int u_number;
    int is_not_dead;
    int team;

    Player_obj();
    int rounds_to_point (Pair *pt, double tolerance);
    int rounds_to_point (Pair *pt, double tolerance, double max_power);
    int hypothetical_rounds_to_point (Pair *pt, double tolerance, Pair *position, Pair
*velocity,
                                double facing, double max_power);
    Pair* should_get_ball (World *world,
                          Pair ball_pos,
                          Pair ball_vel,
                          int max_power_to_exert,
                          int *adv_over_opponent,
                          //for official sserver-sanctioned goalies.

```

```

    int *adv_over_ball,
    int *time_to_reach_ball);

Pair* should_get_ball (World *world,
                      Pair ball_pos,
                      Pair ball_vel,
                      int max_power_to_exert,
                      double steal_radius,
                      int *adv_over_opponent,
                      int *adv_over_ball,
                      int *time_to_reach_ball);

/* pass option at player: determines the best way that this player
can kick the pass to player <to>.
Inputs:
    world *
Returns:
    Player *to is the recipient of the pass
    Returns true if this player thinks the pass will be successful,
    0 if he thinks it will fail
Passed return values:
    dest is the target point we're kicking at,
    kickpower and kickangle define the optimal way to kick
    round advantage is the amount of rounds that the recipient should
    have until an opponent will get the ball
    optimal_angle_abs is the angle along which the kick should be sent
*/
int pass_option_at_player (World *w, Player_obj *to, Pair *dest,
                          Pair &self_pos,
                          Pair &ball_pos, Pair &base_ball_vel,
                          double my_facing,
                          double *kickpower,
                          double *kick_angle, int *round_advantage,
                          double *optimal_angle_abs);

/* does noda computation on the dist and dir to ball to get speed
from the power we're kicking it; second function is the inverse of the first */
double speed_from_kickpower (World *w, double kickpower);
double kickpower_from_speed (World *w, double speed);
double speed_from_kickpower (const Pair &self_pos, const Pair &ball_pos, double ki
ckpower);
double kickpower_from_speed (const Pair &self_pos, const Pair &ball_pos, double sp
eed);

/* computes angle to pass to server in order to turn at desired angle */
double compute_needed_angle_at_speed (double desired_angle, double speed);
double compute_turned_angle_at_speed (double server_angle, double speed);

/* finds 1/2 sweep angle (max sweep) under which we boot the ball into ourself */
double find_kick_into_self_angle (double distance_from_player_to_ball);

/* computes the absolute angle on which you have to kick the ball in order to get
the resultant ball velocity to be along test_angle. ball angle and ball initial sp
eed
are the direction and angle of the ball's current velocity */
int compute_needed_kick_direction (double added_ball_speed, double test_angle, dou
ble ball_angle, double ball_initial_speed, double *your_angle);
};

class Opponent_obj : public Player_obj {
public:
    Opponent_obj ();
};

class Teammate_obj : public Player_obj {
public:
    // int role; //this may specify our field position
    Player_obj* captain;
    Teammate_obj ();
};

//for official sserver-sanctioned goalies.

```

```
//we could have a subclass for the opponents too, but we have no reliable way
//of telling if an opponent is the goalie.
class Goalie_obj : public Teammate_obj {
};

//also remember there's a class self_obj : public player .

class Ball_obj : public Moving_obj {
public:
    int see_timestamp;
    Player_obj* possessor; //is this useful? how would we update it?
    /* *****
    * compute ball steps:
    * computes discrete timesteps of ball -
    * Returns array, s.t. array[i] = distance ball will be at time i
    * last entry k in array is the one s.t. array[k+1] > max_distance
    * NOTE: don't forget to delete the returned array when you're done
    * using it
    */
    double *compute_ball_steps (double speed,
                                double max_distance,
                                int max_rounds,
                                double min_speed,
                                int *size); /* 0: size of returned array */

    void get_ball_odo_up_to_speed(World* world);
    void noda_step (int cur_timestamp);
    Ball_obj();
};

#endif
```

```
// **c++**
// The above line causes xemac to open this file in c++-mode
// this file contains a definition and fcn prototypes for the Pair class.
#ifdef PAIR_H
#define PAIR_H

#include <trig.hh>
#include <iostream.h>

#define ZERO_PAIR Pair(0.0,0.0)

class Pair {
public:
    double x;
    double y;
    double error;

    Pair();
    Pair(const double vx, const double vy);
    int equals_pair(Pair b);
    Pair add_pair(Pair b);
    Pair subtract_pair(Pair b);
    Pair delta_pair(double dist, double theta);
    Pair delta_pair(Pair b);
    void operator = (const Pair& v);
    void operator += (const Pair& v);
    void operator -= (const Pair& v);
    void operator *= (const double& a);
    void operator /= (const double& a);
    inline double r() { return sqrt(square(x)+square(y)); };
    inline double th() { return Atan(y,x); };
    friend Pair operator +(const Pair& a, const Pair& b);
    friend Pair operator -(const Pair& a, const Pair& b);
    friend Pair operator *(const Pair& a, double b);
    friend Pair operator *(double aa, const Pair& a);
    void normalize(const double& l = 1.0);
    double distance(const Pair& orig);
    double angle();
    double angle(const Pair& dir);
    void rotate(const double& ang);
    double vangle(const Pair& target, const Pair& origin);
    double vangle(const Pair& target, const double& origin);
    inline double dot (Pair &u) {return ((x * u.x) + (y*u.y))}; /* takes dot produ
ct of 2 vectors */
};

inline Pair Polar2Pair(double r, double ang)
{
    return Pair(r * cos(ang), r * sin(ang));
}

extern ostream& operator<< (ostream& o, const Pair& v);

#endif
```

```

// --c++--
// The above line causes xemacs to open this file in c++-mode
/* ** Mode: C **
*Header:
*File: param.h
*Author: Noda Itsuki
*Date: 1996/02/23
*EndHeader:
*/

#ifdef PARAM_H
#define PARAM_H
/* for simulator
*/
#define SIMULATOR_STEP_INTERVAL_MSEC 100 /* milli-sec */
#define UDP_RECV_STEP_INTERVAL_MSEC 20 /* milli-sec */
#define UDP_SEND_STEP_INTERVAL_MSEC 150 /* milli-sec */

#define IMPARAM 5.0 /* Inertia-Moment Parameter */

#define TIMEDELTA 50 /* polling interval
[milli-sec] */

/* for network
*/
#define DEFAULT_PORT_NUMBER 6000
#define COACH_PORT_NUMBER 6001
#define MaxMsg 2048
#define SEND 1
#define RECV 2

#define NO_INFO 0
#define SHOW_MODE 1
#define MSG_MODE 2
#define DRAW_MODE 3
#define BLANK_MODE 4

#define DrawClear 0
#define DrawPoint 1
#define DrawCircle 2
#define DrawLine 3

#define MSG_BOARD 1
#define LOG_BOARD 2

/* Objects
*/
#define MaxObject 128

#define GOAL_L_NAME "(goal l)"
#define GOAL_R_NAME "(goal r)"

#define LINE_L_NAME "(line l)"
#define LINE_R_NAME "(line r)"
#define LINE_T_NAME "(line t)"
#define LINE_B_NAME "(line b)"

#define FLAG_C_NAME "(flag c)"
#define FLAG_CT_NAME "(flag ct)"
#define FLAG_CB_NAME "(flag cb)"
#define FLAG_LT_NAME "(flag lt)"
#define FLAG_LB_NAME "(flag lb)"
#define FLAG_RT_NAME "(flag rt)"
#define FLAG_RB_NAME "(flag rb)"

#define FLAG_PLT_NAME "(flag plt)"
#define FLAG_PLC_NAME "(flag plc)"
#define FLAG_PLB_NAME "(flag plb)"
#define FLAG_PRT_NAME "(flag prt)"
#define FLAG_PRC_NAME "(flag prc)"

```

```

#define FLAG_PRB_NAME "(flag pr b)"
#define FLAG_GLT_NAME "(flag gl t)"
#define FLAG_GLB_NAME "(flag gl b)"
#define FLAG_GRT_NAME "(flag gr t)"
#define FLAG_GRB_NAME "(flag gr b)"

#define FLAG_TL50_NAME "(flag t l 50)"
#define FLAG_TL40_NAME "(flag t l 40)"
#define FLAG_TL30_NAME "(flag t l 30)"
#define FLAG_TL20_NAME "(flag t l 20)"
#define FLAG_TL10_NAME "(flag t l 10)"
#define FLAG_T0_NAME "(flag t 0)"
#define FLAG_T10_NAME "(flag t r 10)"
#define FLAG_TR10_NAME "(flag t r 10)"
#define FLAG_TR20_NAME "(flag t r 20)"
#define FLAG_TR30_NAME "(flag t r 30)"
#define FLAG_TR40_NAME "(flag t r 40)"
#define FLAG_TR50_NAME "(flag t r 50)"

#define FLAG_BL50_NAME "(flag b l 50)"
#define FLAG_BL40_NAME "(flag b l 40)"
#define FLAG_BL30_NAME "(flag b l 30)"
#define FLAG_BL20_NAME "(flag b l 20)"
#define FLAG_BL10_NAME "(flag b l 10)"
#define FLAG_B0_NAME "(flag b 0)"
#define FLAG_BR10_NAME "(flag b r 10)"
#define FLAG_BR20_NAME "(flag b r 20)"
#define FLAG_BR30_NAME "(flag b r 30)"
#define FLAG_BR40_NAME "(flag b r 40)"
#define FLAG_BR50_NAME "(flag b r 50)"

#define FLAG_LT30_NAME "(flag l t 30)"
#define FLAG_LT20_NAME "(flag l t 20)"
#define FLAG_LT10_NAME "(flag l t 10)"
#define FLAG_L0_NAME "(flag l 0)"
#define FLAG_LB10_NAME "(flag l b 10)"
#define FLAG_LB20_NAME "(flag l b 20)"
#define FLAG_LB30_NAME "(flag l b 30)"

#define FLAG_RT30_NAME "(flag r t 30)"
#define FLAG_RT20_NAME "(flag r t 20)"
#define FLAG_RT10_NAME "(flag r t 10)"
#define FLAG_R0_NAME "(flag r 0)"
#define FLAG_RB10_NAME "(flag r b 10)"
#define FLAG_RB20_NAME "(flag r b 20)"
#define FLAG_RB30_NAME "(flag r b 30)"

#define PLAYER_NAME_FORMAT "(player %s %d)"
#define PLAYER_NAME_FAR_FORMAT "(player %s)"
#define PLAYER_NAME_TOOFAR_FORMAT "(player)"

#define O_TYPE_FLAG_NAME "(Flag)"
#define O_TYPE_GOAL_NAME "(Goal)"
#define O_TYPE_BALL_NAME "(Ball)"
#define O_TYPE_PLAYER_NAME "(Player)"

#define SAY_MESSAGE_SCAN_FORMAT "(%s %[-0-9a-zA-Z ( ) .+*/?<>_]"

#define TEAM_L_DIRECTION 0
#define TEAM_R_DIRECTION PI
#define SideDirection(side) (((side) == LEFT) ? TEAM_L_DIRECTION : TEAM_R_DIRECTION
ION)

// From here till the next mark the parameters are official.
// - Tim
#define BALL_NAME "(ball)"
#define BALL_SIZE 0.085
#define BALL_DECAY 0.94
#define BALL_RAND 0.05
#define BALL_WEIGHT 0.2
#define BALL_T_VEL 0.001
#define BALL_SPEED_MAX 2.7
#define KICK_DECAY BALL_DECAY

```

```

#define REFEREE_NAME      "referee"
#define PLAYER_SIZE      .8
#define PLAYER_WIDGET_SIZE 1.0
#define PLAYER_DECAY     0.4
#define PLAYER_RAND      0.1
#define PLAYER_WEIGHT    60.0
#define PLAYER_SPEED_MAX 1.0

// Stamina stuph
#define STAMINA_MAX      2000.0
#define STAMINA_INC_MAX 20.0

// Recovery stuph
#define RECOVERY_DEC_THR 0.3
#define RECOVERY_DEC     0.002
#define RECOVERY_MIN     0.5

// Effort stuph
#define EFFORT_DEC_THR   0.5
#define EFFORT_DEC      0.02
#define EFFORT_MIN      0.5
#define EFFORT_INC_THR  0.8
#define EFFORT_INC      0.02

// Hear stuph
#define HEAR_MAX 2
#define HEAR_INC 1
#define HEAR_DECAY 2

#define GOALIE_CATCHABLE_POSSIBILITY 1.0
#define GOALIE_CATCHABLE_AREA_LENGTH 2.0
#define GOALIE_CATCHABLE_AREA_WIDTH 1.0
// Below this mark the parameters may not be
#define VisibleAngle 90.0
#define VisibleDistance 3.0
#define AUDIO_CUT_OFF_DIST 50.0
#define MAX_MSG_LENGTH 512

#define POWERRATE 0.01
#define MAXPOWER 100.0
#define MINPOWER -30.0
#define KICK_POWER_RATE 0.016

#define KICK_RANDOM_RATE 0.1
#define KICKABLE_MARGIN 1.0
#define CONTROL_RADIUS 2.0
#define KICKABLE_AREA KICKABLE_MARGIN+PLAYER_SIZE+BALL_SIZE

#define NormalizebashPower(p) (max(min((p),MAXPOWER), MINPOWER))
#define ReducebashPower(p) (power * stadium->qorate)
#define Sign(x) (((x) > 0.0) ? 1.0: ((x) == 0) ? 0 : -1.0)
#define NormalizekickPower(p) (max(min((p),MAXPOWER), MINPOWER))
/*
#define NormalizekickPower(p) (max(min((p),stadium->maxp),stadium->minp) * stadium->kprate)
*/
#define MAXMOMENT 180
#define MINMOMENT -180
#define NormalizeMomentum(p) Deg2Rad(max(min((p), stadium->maxxm),stadium->minxm))

#define LEFT_STR "l"
#define RIGHT_STR "r"
#define SideStr(side) (((side) == LEFT) ? LEFT_STR : RIGHT_STR )

#define DEF_SAY_MSG_SIZE 256

#define DIST_OSTEP 0.1
#define LAND_OSTEP 0.01

```

```

#define MOMENT_INERTIA 5.0
/*
*****
*Part: for COACH
*****
*/
#define BALL_POS_INFO_STRINGS {"", "in_field", "goal_l", "goal_r", "out_of_field", }

enum BallPosInfo {
    BPI_Null,
    BPI_InField,
    BPI_GoalL,
    BPI_GoalR,
    BPI_OutOfField,
    BPI_MAX
};

/*
*****
*Part: for WEATHER
*****
*/
#define WIND_DIR 0.0
#define WIND_FORCE 10.0
#define WIND_RAND 0.3
#define WIND_WEIGHT 10000.0

/*
*****
*Part: Field Parameter
*****
*/
#define PITCH_LENGTH 105.0
#define PITCH_WIDTH 68.0
#define PITCH_MARGIN 5.0
#define CENTER_CIRCLE_R 9.15
#define PENALTY_AREA_LENGTH 16.5
#define PENALTY_AREA_WIDTH 40.32
#define GOAL_AREA_LENGTH 5.5
#define GOAL_AREA_WIDTH 18.32
#define GOAL_WIDTH 14.02
#define GOAL_DEPTH 2.44
#define PENALTY_SPOT_DIST 11.0
#define CORNER_ARC_R 1.0
#define KICK_OFF_CLEAR_DISTANCE CENTER_CIRCLE_R

#define CORNER_KICK_MARGIN 1.0
#define LENGTH_MAGNIFY 6.0
#define SHOWINFO_SCALE 16.0

#define MAX_PLAYER 11
#define HALF_TIME 600
#define AFTER_GOAL_WAIT 50
#define AFTER_OFFSIDE_WAIT 30

#define MaxStringsSize 4096
#define MAX_FILE_LEN 256
#define UNUM_FAR_LENGTH 20.0
#define UNUM_TOOFAR_LENGTH 40.0
#define TEAM_FAR_LENGTH 40.0
#define TEAM_TOOFAR_LENGTH 60.0

#define BOARD_ROWS 6
#define VIEW_FOR 1
#define VIEW_REV -1

#define OFFSIDE_ACTIVE_AREA_SIZE 9.15

```

#endif

```

#define ROBOCUP_ACT_DOT_HH
#define ROBOCUP_ACT_DOT_HH
// #include <moveto.hh>
#include <world.hh>
#include <new_action.hh>

#define FORMATION_RADIUS 13.0
#define PLAYERS_IN_FORMATION 11
#define FORMATION_TOLERANCE 3.0
#define FORMATION_TURN_TOL_DEG 5.0

/*****
 * Action types
 *****/
class Find_ball: public Action {
private:
    void enqueue_turn_270();
public:
    World *world;

    Find_ball(World *w);
    virtual ~Find_ball();
    virtual int update();
};

class Kick_ball_action: public Action {
public:
    World *world;

    Kick_ball_action(World *w);
    virtual ~Kick_ball_action() {};
    virtual int update();
};

# ifdef CHASER // For new move
class Move_into_formation: public Action {
private:
    Pair formation_positions[PLAYERS_IN_FORMATION];
    int destination_idx;
    void construct_from_scratch();
    double compute_destination();
public:
    World *world;

    Pair get_dest();
    Move_into_formation(World *w);
    virtual ~Move_into_formation();
    virtual int update();
};

class Move_to : public Action {
private:
    Pair formation_positions[PLAYERS_IN_FORMATION];
    int position_taken[PLAYERS_IN_FORMATION];
    float position_dist[PLAYERS_IN_FORMATION];
    int destination_idx;
    int formation_positions_set;
public:
    World *world;
    Move_to();
    virtual ~Move_to() {};
    virtual int is_finished() {return FALSE;};
    virtual void print() {cout << "Moveto" << endl;};

    Pair D_last_pos;
    int D_last_round;

```

```

    int backwards;
    int backwards_turn_round;
    int watch_ball_too;

    Pair destination; //absolute position to reach
    Pair displacement; //vector from current position to destination
    Pair tangent_disp; //tangential displacement
    Pair disp_minus_tangent; //spot we should run towards in order to reach destination
n,
    // considering tangential velocity */
    double tolerance; //parameter for how close we need to be to destination (creates
    a destination circle)
    int max_power; //max power willing to spend
    Pair est_destination; //where within tolerance circle we think we'll be
    int missing_turn;
    int less_wasteful_dashes;
    int dest_might_change;

    Self_obj *s; //needs to be stored? we do need to get back facing, velocity, etc f
    rom self class

    Move_to (Move_to *mt); // Copy constructor
    Move_to (World* w, Self_obj *s, Pair dest, double max_power, double tol_dist);
    Move_to (World* w, Self_obj *s, Pair dest, double max_power, double tol_dist,
            int less_wasteful_dashes, int dest_might_change);

    Move_to (World* w, Self_obj *s, Pair dest, double max_power, double tol_dist,
            int less_wasteful_dashes, int dest_might_change, int watch_ball
            _too);

    /* current_speed_for_dashes - amount to base decision on how fast to run
    comes from this function, basically a transform of our current velocity */
    double current_speed_for_dashes (Pair &velocity, double facing);

    /* recompute existing dashes - redoes all unspent dashes as in fill */
    Command_obj * recompute_existing_dashes (double *dist_remaining, double *radial_sp
    eed, double max_power);

    /* delete remaining_commands - empties command list from first to go to end */
    void delete_remaining_commands (Command_obj *first_to_go);

    //virtual void fprintf_action (FILE *output_file);
    void fill (World* w, Self_obj *s, Pair dest, double max_power, double tol_dist);

    int run_backwards_to_see_ball (int watch_ball);
    int decide_backwards_when_seeing_ball (Pair &ball_pos, double facingr);

    int detect_collision (World* world, Pair self_pos, Pair self_vel, double self_spee
    d, double facingr);
    int detect_collision_one_obj (Pair collis_pos, Pair collis_vel, Pair self_pos, Pair
    self_vel,
                                double self_speed, double self_faci
    ngr);
    int detect_collision_pt (Pair &pos1, Pair &pos2);

    /* compute needed angle - computes how much we need to turn in order to hit destin
    ation exactly
    Returns 1 if we reach target circle, 0 if not
    sweep_tolerance, 0: +/- allowable to angle of displacement so that we hit targe
    t circle
    turn_angle, 0: angle needed to turn to bring us perfectly on course
    */
    char compute_needed_angle (double *sweep_tolerance, double *turn_angle);

    /* figures out if we should immediately turn in order to optimally get to
    the destination point
    PostConds: if needed, a turn is inserted at the start of the command list
    missing_turn is ste to 1 if we think we'll need to turn later
    */
    void make_imm_turn (double distance_remaining);

```

```
/* compute_needed_dashes - adds dashes to the command list until we reach restinat
ion. It takes a
starting speed in the radial direction and the distance to go. Uses class vari
able tolerance.
Will not create a dash with power greater than max_power, but will create one w
ith less power if
the larger power will only waste radial velocity to the max speed cap
*/
void compute_needed_dashes (double *dist_remaining, double radial_speed, double ma
x_power);

/* goes transformation from max power to speed this power will result in if we das
h at this
power for an infinite amount of time */
double compute_runatspeed_from_maxpower (double power);

/* compute_best_power - returns greatest dash power <= max_power, such that radial
_speed does not go above
speed maximum after the dash is executed
*/
double compute_best_power (double radial_speed, double max_power);
int okay_run_without_turning (double facingr, Pair &self_pos, Pair &self_vel);

void compute_tangential_displacement (); /* computes disp_minus_tangent and tangen
t_disp field; */
int update_calculation ();
virtual int update ();

void grab_closest ();
void set_player_closest_position(int);
int get_closest_free_pos_index(Pair pos);
void compute_destination();
int all_taken ();
Pair get_dest ();
};
#endif // Move to out for now for new_move
#endif
```

```

#define ROBOCUP_COM_DOT_HH
#define ROBOCUP_COM_DOT_HH
#include <command.hh>
#include <world.hh>

#define COMMAND_DASH 10
#define COMMAND_TURN 11
#define COMMAND_MOVE 12
#define COMMAND_KICK 13
#define COMMAND_CATCH 14

class Command_dash : public Command_obj {
public:
    Command_dash(World* w, double p);
    virtual ~Command_dash() {world = (World*)NULL;};
    virtual int send_command() ;
    virtual void print();
    virtual int command_type() {return COMMAND_DASH;};
    double get_power() {return power;};
    void set_power(double p) {power = p;};
private:
    World* world;
    double power;
};

class Command_turn : public Command_obj {
public:
    Command_turn(World* w, int a);
    virtual ~Command_turn() {world = (World*)NULL;};
    virtual int send_command() ;
    virtual void print();
    virtual int command_type() {return COMMAND_TURN;};
private:
    World* world;
    int angle;
};

class Command_move : public Command_obj {
public:
    Command_move(World* w, double xc, double yc);
    virtual ~Command_move() {world = (World*)NULL;};
    virtual int send_command() ;
    virtual void print();
    virtual int command_type() {return COMMAND_MOVE;};
private:
    World* world;
    double x;
    double y;
};

class Command_kick : public Command_obj {
public:
    Command_kick(World* w, double power, int dir);
    virtual ~Command_kick() {world = (World*)NULL;};
    virtual int send_command() ;
    virtual void print();
    virtual int command_type() {return COMMAND_KICK;};
private:
    World* world;
    double power;
    int angle;
};

class Command_catch : public Command_obj {
public:
    Command_catch(World* w, int ang);
    virtual ~Command_catch() {world = (World*)NULL;};
    virtual int send_command() ;
    virtual void print();
    virtual int command_type() {return COMMAND_CATCH;};
};

```

```

private:
    World* world;
    int angle;
};
#endif

```

```

// --c++--
// The above line causes xemacs to open this file in c++-mode
// the self class.
#ifdef SELF_H
#define SELF_H

#include "object.hh"
#include <stdio.h>
#include <stdlib.h>

#define VIEW_WIDTH_NARROW 0
#define VIEW_WIDTH_NORMAL 1
#define VIEW_WIDTH_WIDE 2

#define VIEW_QUALITY_HIGH 0
#define VIEW_QUALITY_LOW 1

class World;
class Vis_info;

//this class holds all the data we have about ourself. this class is intended to be
//instantiated only once.
//note. we inherit from Player_obj, not Teammate_obj.

class Self_obj : public Player_obj {
public:
    /* remember, there is data from superclasses:
    position
    velocity
    timestamp
    accuracy

    odo_position
    odo_velocity
    odo_timestamp
    odo_accuracy

    int facing;
    int u_number;
    char* team_name;
    */

    //this data all public for easy access.
    int facing_at_last_dash;
    //id data
    int side; // LEFT or RIGHT
    int role; //this may specify our field position
    int group_number; //we might use these.
    Player_obj* captain;
    int goalie; //am I an official soccerserver-sanctioned goalie?
    int catch_time;

    char team_name[32];
    int body_timestamp;
    int stamina;
    double effort; //the new 4.16 number that says the extent to which
    //we can translate dash/kick power into actual motion. based to some extent
    //on stamina.

    // A number of unknown usefulness and rowhere else to go.
    double sense_body_speed;

    int view_width;
    int view_quality;
    int view_step_clicks;
    int view_step_counter;

    //utility objects.
    Socket* sock; //the socket we use to communicate.

```

```

    int num_kick;
    int num_dash;
    int num_turn;
    int num_say;

    // Pass info
    int should_recieve_pass;
    double angle_to_turn;

    Self_obj();
    Self_obj(Socket* sock_in);
    void fprintf_self(FILE* file);
    void fprintf_body_data(FILE* file);

    void set_starting_position();
    int compute_position(World* world, Vis_info* vis_info);
    void noda_step(int cur_timestamp);

    int command_dash(World* world, double power);
    int command_turn(World* world, int angle);
    //int command_move(World* world, double x,double y);
    int command_kick(World* world, double power,int dir);
    int command_say(World* world, char* msg);
    int command_change_view(World* world, int width, int quality);
    //int command_sense_body(World* world);
    int command_catch(World* world, int ang);
    // Pair* should_get_ball(World* world, int max_power_to_exert, int *num_rounds_to
    _ball);
    int can_get_ball(World* world);
    int get_turn_to_goal();
};

#endif

```

```

// *-c++-*-
// The above line causes xemacs to open this file in c++-mode//this file contains c1
ass defs and prototypes for our sensor info data.
#define SENSOR_H
#define SENSOR_H

// Homebrewn includes
#include <world.hh>

// System includes
#include <iostream.h>
#include <limits.h>

#define VIS_DIST_ERR -1
#define VIS_DIR_ERR -181
#define VIS_CHNG_ERR -FLT_MAX
#define VIS_WT_ERR -1
#define VIS_UNUM_ERR -1
#define ERROR_PAIR Pair(-LONG_MAX, -LONG_MAX)

#define VIS_WT_our 1
#define VIS_WT_opponent 0

#define VIS_SIDE_ERR -1
#define VIS_SIDE_RIGHT 2
#define VIS_SIDE_LEFT 1
#define VIS_FP_ERR -1
#define VIS_LP_ERR -1

int compute_direction_low(Pair P1,
                          double R1,
                          double alpha1,
                          Pair P2,
                          double R2,
                          double alpha2
                          );

#define MAXSENNOBJ 72
// This got bigger now that we can see more flags
#define VISBUFSIZE 4096

class Vis_info {
public:
    char original_buffer[VISBUFSIZE];

    int timestamp;
    int num_objects;
    int num_lines;
    int num_flags;

    // We never see more than two lines
    Line_vis_obj* lines; // Pointer to an array
    int num_goals;
    // It is actually possible to see both goals
    Goal_vis_obj* goals; // Pointer to an array

    // This array should be indexable by flag_type
    // The extra slot is for a flag whose type we do not know
    Flag_vis_obj* flags; // Pointer to an array

    Ball_vis_obj ball;

    // We can never see ourselves
    Player_vis_obj* players; // Pointer to an array
    // This is used to add to the players array
    int num_players;
    // Number of players added to the world's unknown array
    int num_unknowns;
    int quality;

    Vis_info(World *world);

```

```

-Vis_info();
int fill_info(World* world, char* buffer);
void stamp_em_baby();
};

extern ostream& operator<< (ostream& o, const Vis_info& v);
void fprintf_vis_info(FILE* f, Vis_info *vi);

// Sender types
#define SENDER_UNKNOWN -1
#define SENDER_REFEREE 0
#define SENDER_SELF 1

class Hear_info {
public:
    int timestamp;
    int sender;
    int dir;
    char message[MAX_SAY_LENGTH];

    Hear_info(char *buf);
    void fprintf_hear_info(FILE* f);
};

class Body_info {
public:
    int timestamp;
    double stamina;
    double effort;
    double speed; //magnitude quantized by .01
    int view_mode;
    int view_quality;
    int view_width;

    int num_kick;
    int num_dash;
    int num_turn;
    int num_say;

    Body_info(char* buf);
    void apply(World* world);
};

*/
char *next_token(char *buf);

#endif

```

```

// **C++**
// The above line causes xemacs to open this file in c++-mode
#ifdef TRIG_H
#define TRIG_H

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define EPS 1.0e-10
#define INFINITE 1.0e10
#define PI M_PI
#define max(x,y) (x>y?x:y)
#define min(x,y) (x>y?v:x)
#define Abs(x) ((x) > 0.0)?(x):(-(x))
#define Atan(y,x) ((x==0.0 && y==0.0)?(0.0):atan2(y,x))
#define RAD2DEG (180.0/PI)
#define DEG2RAD (PI/180.0)
#define TWOPI (2*PI)
#define P5PI (PI/2)
#define P25PI (PI/4)
#define P125PI (PI/8)
#define Rad2Deg(a) (int)(Rad2Deg(a))
#define Quantize(v,q) ((rint((v)/(q)))*(q))

#define Pow(x) ((x)*(x))
#define square(x) ((x)*(x))

#define RANDOMBASE 1000
#define IRANDOMBASE 31

#define drand(h,l) (((((h)-(l)) * ((double)rand()/%RANDOMBASE) / (double)RANDOM
BASE))) + (l))
#define irand(x) ((rand() / IRANDOMBASE) % (x))

#define ROOT2 1.4142136
#define MAX_TURN_ERROR_PERCENT .1

/* PLAYER_DIST_FROM_VEL_FACTOR is the equivalent of the integral of velocity from t
ime = 0 to infinity,
with degradation. Because the timestep does discrete math,
total distance = sum of integers i over [0, infinity) of [ (.4)^i * velocity] =
velocity / (1-.4) = velocity * 1.666667
*/

#define PLAYER_DIST_FROM_VEL_FACTOR (1 / (1 - PLAYER_DECAY)) // 5/3 for .4 decay
#define PLAYER_DECAYED_DIST (1 / (1 - PLAYER_DECAY))

#define BALL_DIST_FROM_VEL_FACTOR (1 / (1 - KICK_DECAY)) // 50/3 for .94 decay
#define BALL_DECAYED_DIST (1 / (1 - KICK_DECAY))

double abs (double x);
double deg2rad(int theta);
int rad2deg(double theta);
double normalize_deg(double theta);
double normalize_rad(double theta);
int ceiling (double);
//int floor (double);
double x_to_y (double x, int y);
void swap (double &x, double &y);
#endif

```

```
// For utilities -Tim
// Seeds the random number generator off the timer.
//Returns value used to seed;
unsigned int randomize(void);
```

```

#define VISOBJ_DOT_H
#define VISOBJ_DOT_H

// This count does not include the player himself,
// only the people he can see on the field
#define NUM_PLAYERS 21
#include <pair.hh>
#include <object.hh>

#define NUM_PLAYERS 21

class Vis_obj {
public:
    int timestamp;

    int in_cone;
    double dist;
    double dir;
    double dist_change;
    double dir_change;
    int facing;

    Pair delta; //this is a vector from us to the object, computed from dist,dir.
    //from delta we will compute the objects world-relative position.
    //from dist_change and dir_change, we will compute the object's
    //world-relative velocity.

    Vis_obj ();
    void compute_delta(int facing); //does what it says.
    Pair compute_position(int facing, int object_is_goal);
    int compute_direction(int closest_obj_is_goal, obj,
        Vis_obj *second_closest, obj,
        int second_closest_obj_is_goal);
};

class Player_vis_obj : public Vis_obj {
private:
    Player_vis_obj(Vis_obj* base);
public:
    int unum;
    int team;
    Player_obj* player;
    Player_vis_obj();
    void fill(Vis_obj* base);
    void apply(World* world);
};

class Ball_vis_obj : public Vis_obj {
public:
    Ball_obj* ball;
    Ball_vis_obj();
    Ball_vis_obj(Vis_obj* base);
    void fill(Vis_obj* base);
    void apply(World* world);
};

class Flag_vis_obj : public Vis_obj {
private:
    Flag_vis_obj(Vis_obj* base);
    Flag_vis_obj(Vis_obj* base, World* world, int flag_type);
public:
    Flag_obj* flag;
    int fpos;
    Flag_vis_obj();
    void fill(Vis_obj* base);
    void fill(Vis_obj* base, World* world, int flag_type);
};

extern ostream& operator<< (ostream& o, const Flag_vis_obj& v) ;

```

```

class Goal_vis_obj : public Vis_obj{
private:
    Goal_vis_obj(Vis_obj* base, World* world);
public:
    Goal_obj* goal;
    int side;
    Goal_vis_obj();
    void fill(Vis_obj* base, World* world);
};

class Line_vis_obj : public Vis_obj{
private:
    Line_vis_obj(Vis_obj* base);
    Line_vis_obj(Vis_obj* base, World* world, int line_type);
public:
    Line_obj* line;
    int lpos;
    Line_vis_obj();
    void fill(Vis_obj* base);
    void fill(Vis_obj* base, World* world, int line_type);
    // These functions are called from Self::compute_position
    int compute_direction(int out_of_field);

    Pair compute_position (Vis_obj* vis_obj_ptr,
        int vis_obj_is_goal,
        int out_of_field,
        int facing );
};

#endif

```

```

// **c++**
// The above line causes xemaps to open this file in c++-mode//these are the instant
iate-once whole-world data structures.
#ifdef WORLD_H
#define WORLD_H

#define BEFORE_KICK_OFF 0
#define TIME_OVER 1
#define PLAY_ON 2
#define KICK_OFF_L 3
#define KICK_OFF_R 4
#define KICK_IN_L 5
#define KICK_IN_R 6
#define FREE_KICK_L 7
#define FREE_KICK_R 8
#define CORNER_KICK_L 9
#define CORNER_KICK_R 10
#define GOAL_KICK_L 11
#define GOAL_KICK_R 12
#define GOAL_L 13
#define GOAL_R 14

#define BEFORE_KICK_OFF_STR "before_kick_off"
#define TIME_OVER_STR "time_over"
#define PLAY_ON_STR "play_on"
#define KICK_OFF_L_STR "kick_off_l"
#define KICK_OFF_R_STR "kick_off_r"
#define KICK_IN_L_STR "kick_in_l"
#define KICK_IN_R_STR "kick_in_r"
#define FREE_KICK_L_STR "free_kick_l"
#define FREE_KICK_R_STR "free_kick_r"
#define CORNER_KICK_L_STR "corner_kick_l"
#define CORNER_KICK_R_STR "corner_kick_r"
#define GOAL_KICK_L_STR "goal_kick_l"
#define GOAL_KICK_R_STR "goal_kick_r"
#define GOAL_L_STR "goal_l"
#define GOAL_R_STR "goal_r"

#define NUM_MSGS 5
#define MAX_SAY_LENGTH 512
#define ARG_ERR - INT_MAX

// #define VIS_TIMING_DEBUG_MODE

#include <self.hh>
#include <visobj.hh>

class Zone_Def;

// #include <zones.hh>

// this contains the stationary objects on the field.
class Field {
public:
    Flag_obj* flags[NUM_FLAGS+1]; //there are now many flags.
    Line_obj* lines[4];
    Goal_obj* goals[2]; //we may not use these anymore.
    Field();
    ~Field();
};

class Hear_info;
class Action_Q;
class Change_view_action;
class Say_action;
class Post_info;
class Action_obj;

```

```

extern World *global_world;
//this structure contains the 23 moving objects.
class World {
public:
    Ball_obj* ball;
    Self_obj* me;
    Teammate_obj* teammates[11];
    Opponent_obj* opponents[11];
    Player_obj* unknowns[22];

    Line_vis_obj seen_lines[2];
    Goal_vis_obj seen_goals[2];
    Player_vis_obj seen_players[NUM_PLAYERS];
    Flag_vis_obj seen_flags[NUM_FLAGS+1];

    Field* field_ptr;
    Hear_info* messages[NUM_MSGS];
    Zone_Def* zone_def;

    // Action_Q *action_q;
    Say_action* say_act;
    Change_view_action* change_view_act;

    Pair initial_position[11];

    //status-like things.
    int latest_message;
    int mode; //this refers to the before_kick_off, play_on, etc.
    int which_side; //are we right or left?
    int our_score;
    int their_score;
    int half;
    int time_now;
    int time_left;

    int alrms_since_last_info;

    // For thesis output
    int num_dest_changes;
    int last_dest_time;

    World();
    World(Socket* sock_in);
    ~World();
    void fill_world(int side, char* team_name, int u_num,
                   char* mode_name, FILE *output_file, int queue_mode_state);
    int store_message(Hear_info *audio);
    void fprintf_messages(FILE *f);
    Vis_info* get_info(int *got_hear_info);
    Vis_info* parse_vis_info(char *buffer);
    int parse_message_array(int current_time);
    void parse_message(int index);
    int get_sense_body();
    int synchronize_and_start_timer();
    void main_loop();

    //will's zone-testing loop.
    void zone_loop();

    int last_vis_info_time;
    int see_ball_now();

    int rounds;
    int action_step_clicks; // Controls when the next action is sent
    int timer_ticks; // How long the timer should tick for
    //void start_timer();

    void fill_body_info (char *we_r_buff);

    //updates the world doing simualtor-simulation.
    void step();

```

```
Post_info* retrieve_post_info(Action_obj *action);
//utility shite.
int queue_mode_on;
// This is a boolean which determines whether or not
// willclient is to enqueue commands or send them right away
FILE* output_file;
//debugging fcn. do nothing if DEBUG_MODE is not defined.
void debug(char* msg);
// test shite
void test_queue();
int timer_control_on; //Boolean
void fprintf_moving_objs();
// Initial move
int send_initial_move();
void init_initial_positions();
// Intercept routines (intercept.cc)
int get_min_teammate_time_to_point(Pair *dest_pt, double tolerance);
int get_min_opponent_time_to_point(Pair *dest_pt, double tolerance);
// Some helper routines for X-Module stuff (-Tim)
void wait_for_mode_change();
};
//these use that evil global_world pointer.
//void sense_module(void);
//void actuate_module(void); // Only this one is actually used
// A little utility is good once in awhile
int mode_string_to_number(char* mode_name);
#ifdef CHASER
void prepare_starting_queue(World *the_world, void *aq_v);
#endif
#endif
```