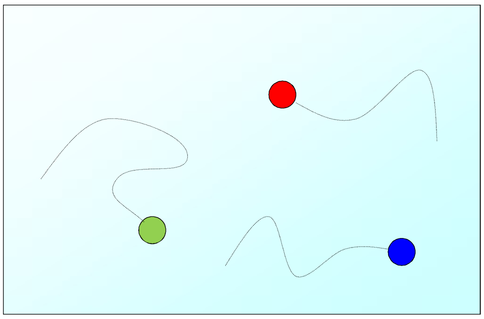
Sample Real-Time Applications

Object animation



General approach

For each object $\mathbf{OBJ_i}$ define:

- data structures and state variables
- task for updating state variables (period $\mathbf{T_i}$)

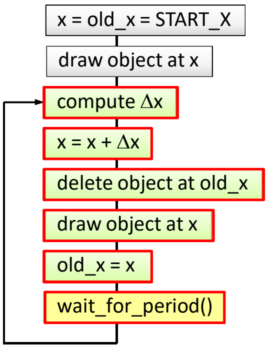
For the whole application define:

- graphic visualization (period $\mathbf{T_g}$)
- User interface & command interpreter (period $\mathbf{T_u}$)

For the graphic display there are two options:

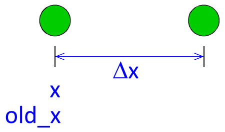
1. Each object draws itself every period.
2. All objects are drawn by a single graphic task.

Case 1: self drawing



x = old_x = START_X
draw object at x
compute Δx
x = x + Δx
delete object at old_x
draw object at x
old_x = x
wait_for_period()

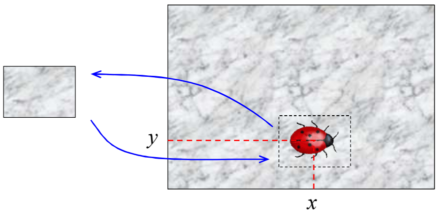
x	current position
old_x	previous position
Δx	position increment



NOTE: If there is a background, it has to be managed.

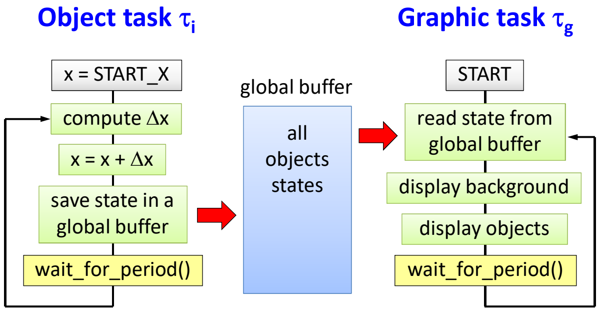
Case 1: handling background

Before drawing the object, we have to save the background in a buffer:



To delete the object, we can just restore the background from the buffer.

Case 2: graphic task



Object task τ_i

x = START_X
compute Δx
x = x + Δx
save state in a global buffer
wait_for_period()

global buffer

all objects states

Graphic task τ_g

START
read state from global buffer
display background
display objects
wait_for_period()

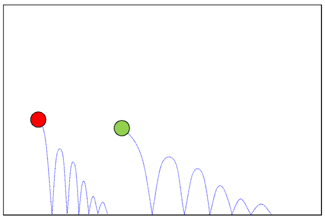
Physical simulations

- **Modeling**
  - Derive the system model expressing the position x as a function of the other state variables.
- **Speed must be independent of the period**
  - compute: `dx = vx * T;`
  - For smooth motion, period should be no larger than 20 ms
  - If T is larger, you can multiply T by a timescale factor.
- **Scalability**
  - Express physical variables in MKS units using floats;
  - Convert to integer coordinates only for displaying;
  - use offset and scale factor for graphic coordinates:

`xg = offset + x * scale;`

Jumping balls

Let us see how to write a program that simulates a number of jumping balls that move in a box following the laws of physics:



Things to define

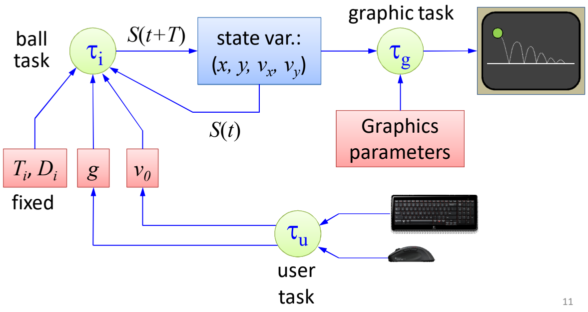
- General description and requirements
- Application structure
- Graphical layout
- User interface
- Motion rules
- Constants of the simulation
- State variables of the ball
- Global variables and data structures
- Auxiliary functions (divide them into categories)

General description

1. Each ball is animated by a dedicated periodic task that updates its state every period.
2. The motion of each ball is independent on the other balls, but depends on its current state.
3. Balls moves within a box and bounce on its borders.
4. Balls lose some energy when bouncing on the floor.
5. The user can interact with the program and can create new balls, make them jumping, vary gravity, etc.
6. The system provides information on: number of active balls, deadline misses, current gravity.

Application structure

The task animating the ball must periodically update the ball state from $S(t)$ to $S(t+T)$:



ball task

τ_i

$S(t+T)$

state var.: (x, y, v_x, v_y)

graphic task

τ_g

$S(t)$

Graphics parameters

T_i, D_i fixed

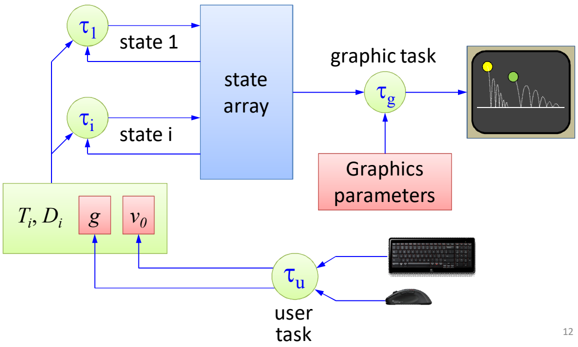
g

v_0

τ_u user task

Application structure

The state of multiple balls is stored in an array:



τ_1

state 1

τ_i

state i

state array

graphic task

τ_g

Graphics parameters

T_i, D_i

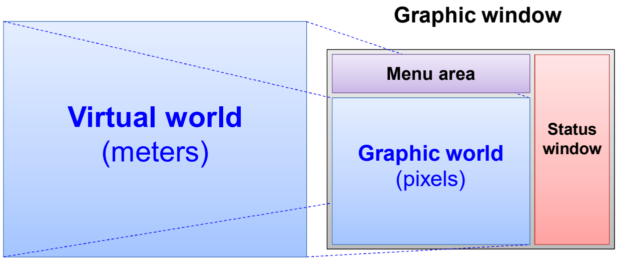
g

v_0

τ_u user task

Graphical layout

Simulations should be done in real units (MKS) and only later converted in pixels.



Graphic window

Virtual world
(meters)

Menu area

Graphic world
(pixels)

Status window

User interface

We have to decide the inputs and the outputs:

Inputs

- What variables can the user change?
- Pressing which keys?
- Is the mouse used?

Outputs

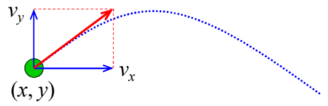
- Which variables are displayed?

User interface

- **SPACE**: creates a new ball with random parameters
- **ESC**: exits the program
- **A**: makes balls jumping (provide extra energy)
- **W**: shows/hides the ball trail
- **LEFT / RIGHT**: decrease/increase trail length
- **UP / DOWN**: increase/decrease gravity

Corresponding values must be shown on the state window

Motion rules



$$\begin{cases} x^{new} = x^{curr} + v_x t \\ y^{new} = y^{curr} + v_y^{curr} t - \frac{1}{2} g t^2 \end{cases} \qquad \begin{cases} v_x = constant \\ v_y^{new} = v_y^{curr} - gt \end{cases}$$

- These equations must be updated every period T_i
- A time scale can be used for the integration interval:

dt = TSCALE * T_i

A closer look to the source code

Makefile

```
#--------------------------------------------------
# Target file to be compiled by default
#--------------------------------------------------
MAIN = balls
#--------------------------------------------------
# CC will be the compiler to use
#--------------------------------------------------
CC = gcc
#--------------------------------------------------
# CFLAGS will be the options passed to the compiler
#--------------------------------------------------
CFLAGS = -Wall -lpthread -lrt
#--------------------------------------------------
# Dependencies
#--------------------------------------------------
$(MAIN): $(MAIN).o ptask.o
	$(CC) $(CFLAGS) -o $(MAIN) $(MAIN).o ptask.o `allegro-config --libs`

$(MAIN).o: $(MAIN).c
	$(CC) -c $(MAIN).c

ptask.o: ptask.c
	$(CC) -c ptask.c
```

Header files

```
//-----------------------------------------------------
// BALLS:      SIMULATION OF JUMPING BALLS
//             with a single display task
//-----------------------------------------------------

#include <stdlib.h>       // include standard lib first
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <sched.h>
#include <allegro.h>
#include <time.h>

#include "ptask.h"        // include own lib later
#include "mylib.h"
```

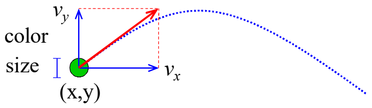
Constants

```
//-----------------------------------------------------
// GRAPHICS CONSTANTS
//-----------------------------------------------------
#define XWIN   640     // window x resolution
#define YWIN   480     // window y resolution
#define BKG    0       // background color
#define MCOL   14      // menu color
#define NCOL   7       // numbers color
#define TCOL   15      // trail color
//-----------------------------------------------------
#define LBOX   489     // X length of the ball box
#define HBOX   399     // Y height of the ball box
#define XBOX   5       // left   box X coordinate
#define YBOX   75      // upper  box Y coordinate
#define RBOX   495     // right  box X coordinate
#define BBOX   475     // bottom box Y coordinate
#define FLEV   5       // floor Y level (in world)
```

Constants

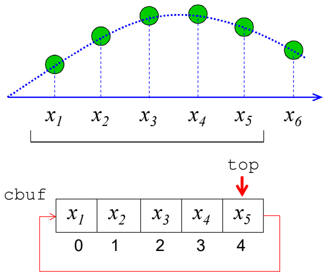
```
//-----------------------------------------------------
// TASK CONSTANTS
//-----------------------------------------------------
#define PER    20              // task period in ms
#define DL     20              // relative deadline in ms
#define PRI    80              // task priority
//-----------------------------------------------------
// BALL CONSTANTS
//-----------------------------------------------------
#define MAX_BALLS     20     // max number of balls
#define G0            9.8    // acceleration of gravity
#define TLEN          30     // trail length
#define HMIN          200    // min initial height
#define HMAX          390    // max initial height
#define VXMIN         20     // min initial hor. speed
#define VXMAX         10     // max initial hor. speed
#define DUMP          0.9    // dumping coefficient
#define TSCALE        10     // time scale factor
```

State variables



```
struct status {       // ball structure
       int    c;      // color [1,15]
       float  r;      // radius (m)
       float  x;      // x coordinate (m)
       float  y;      // y coordinate (m)
       float  vx;     // horizontal velocity (m/s)
       float  vy;     // vertical velocity (m/s)
       float  v0;     // jumping velocity (m/s)
};

struct status ball[MAX_BALLS];   // balls status buffer
```

Storing the trail



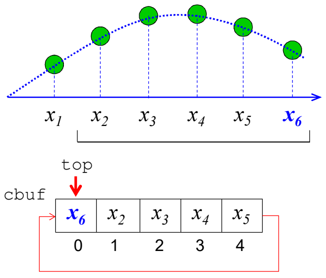
Store a new value x:

```
top = (top + 1) % L;
cbuf[top] = x;
```

It can be done using a circular buffer of length L = max trail length:

- top always points to the most recent data
- next element:
  (top + 1) % L
- previous element:
  (top – 1 + L) % L
- k^{th} previous element:
  (top – k + L) % L

Reading the trail

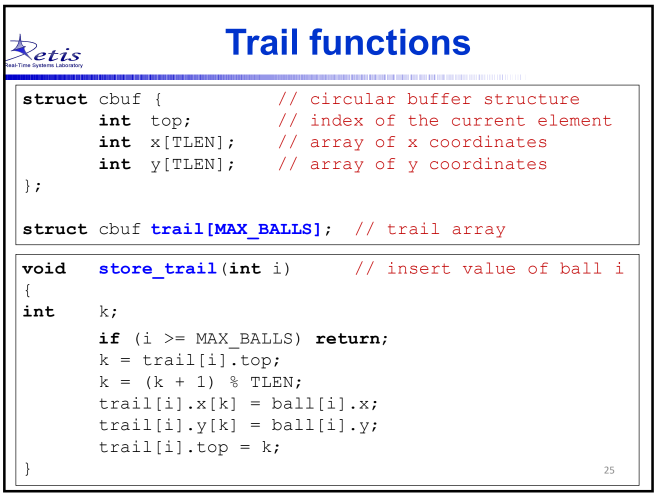


Getting the k^{th} previous value :

```
i = (top - k + L) % L;
value = cbuf[i];
```

It can be done using a circular buffer of length L = max trail length:

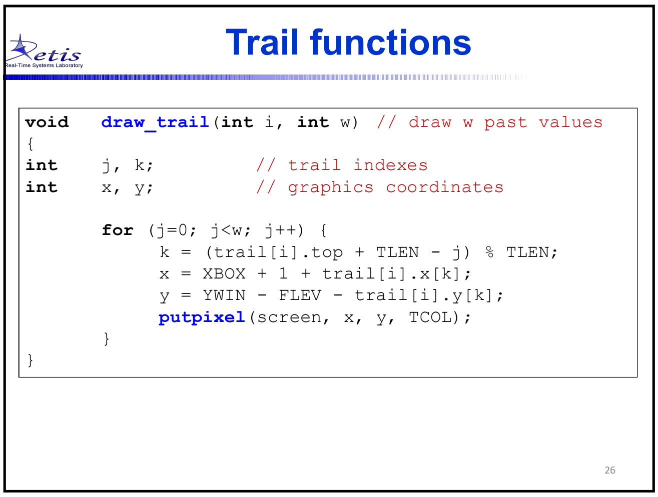
- top always points to the most recent data
- next element:
  (top + 1) % L
- previous element:
  (top – 1 + L) % L
- k^{th} previous element:
  (top – k + L) % L



Trail functions

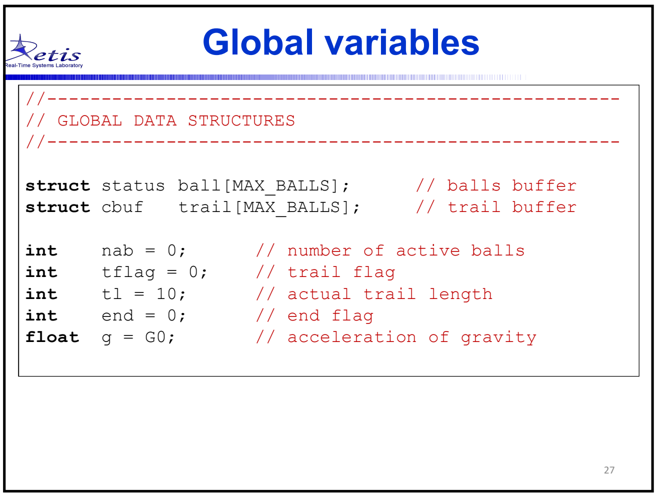
```
struct cbuf {           // circular buffer structure
       int  top;        // index of the current element
       int  x[TLEN];    // array of x coordinates
       int  y[TLEN];    // array of y coordinates
};

struct cbuf trail[MAX_BALLS];  // trail array
```

```
void    store_trail(int i)      // insert value of ball i
{
int     k;
        if (i >= MAX_BALLS) return;
        k = trail[i].top;
        k = (k + 1) % TLEN;
        trail[i].x[k] = ball[i].x;
        trail[i].y[k] = ball[i].y;
        trail[i].top = k;
}
```



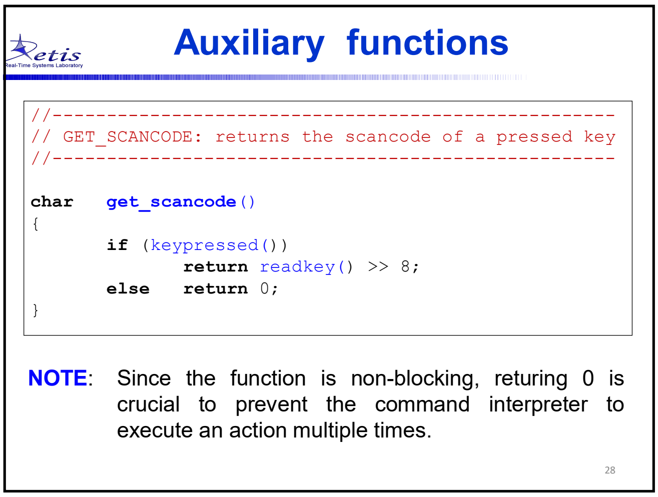
Trail functions

```
void    draw_trail(int i, int w)  // draw w past values
{
int     j, k;           // trail indexes
int     x, y;           // graphics coordinates

        for (j=0; j<w; j++) {
            k = (trail[i].top + TLEN - j) % TLEN;
            x = XBOX + 1 + trail[i].x[k];
            y = YWIN - FLEV - trail[i].y[k];
            putpixel(screen, x, y, TCOL);
        }
}
```



Global variables

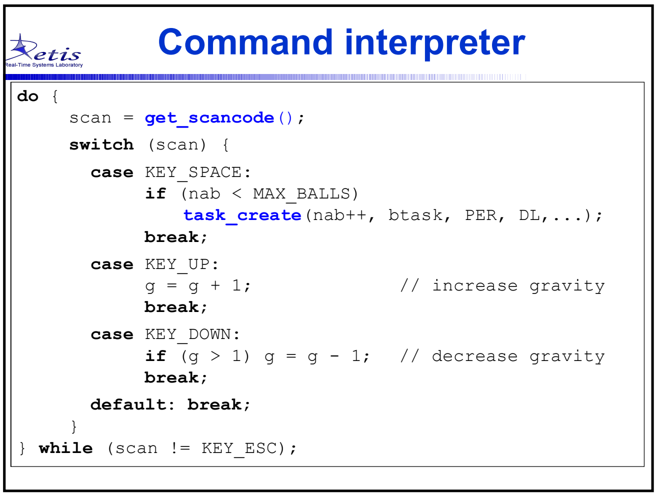
```
//-------------------------------------------------------
// GLOBAL DATA STRUCTURES
//-------------------------------------------------------

struct status ball[MAX_BALLS];      // balls buffer
struct cbuf   trail[MAX_BALLS];     // trail buffer

int     nab = 0;        // number of active balls
int     tflag = 0;      // trail flag
int     tl = 10;        // actual trail length
int     end = 0;        // end flag
float   g = G0;         // acceleration of gravity
```



Auxiliary functions

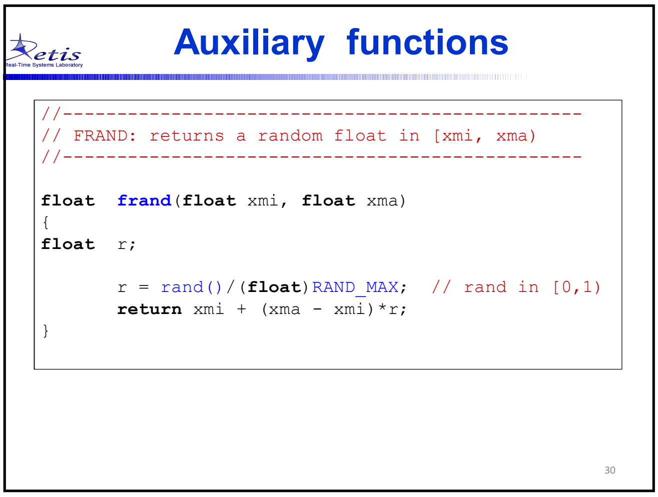
```
//-------------------------------------------------------
// GET_SCANCODE: returns the scancode of a pressed key
//-------------------------------------------------------

char    get_scancode()
{
        if (keypressed())
                return readkey() >> 8;
        else    return 0;
}
```

NOTE: Since the function is non-blocking, returing 0 is crucial to prevent the command interpreter to execute an action multiple times.



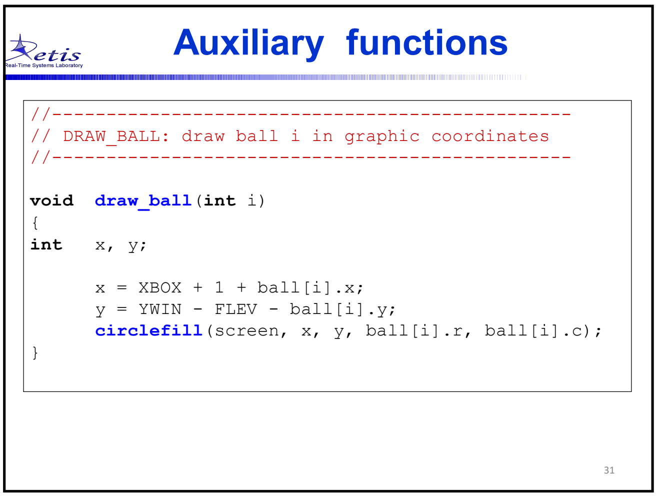
Command interpreter

```
do {
     scan = get_scancode();
     switch (scan) {
       case KEY_SPACE:
            if (nab < MAX_BALLS)
                task_create(nab++, btask, PER, DL,...);
            break;
       case KEY_UP:
            g = g + 1;              // increase gravity
            break;
       case KEY_DOWN:
            if (g > 1) g = g - 1;   // decrease gravity
            break;
       default: break;
     }
} while (scan != KEY_ESC);
```



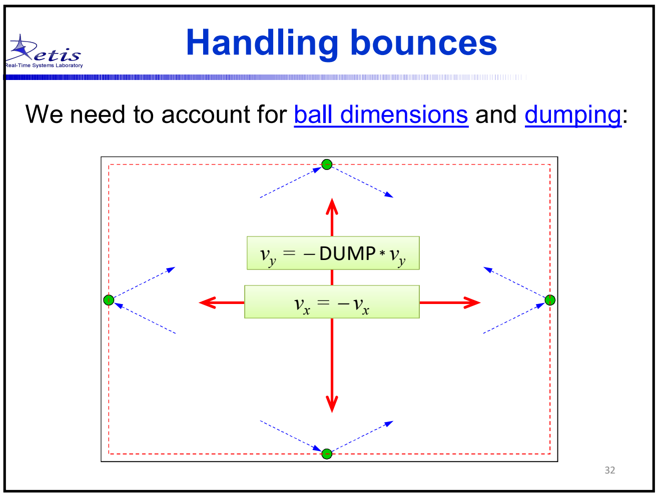
Auxiliary functions

```
//-------------------------------------------------
// FRAND: returns a random float in [xmi, xma)
//-------------------------------------------------

float   frand(float xmi, float xma)
{
float   r;

        r = rand()/(float)RAND_MAX;  // rand in [0,1)
        return xmi + (xma - xmi)*r;
}
```



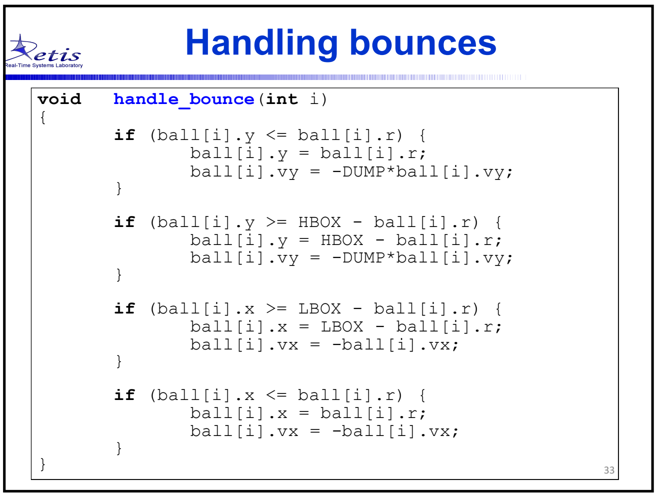
Auxiliary functions

```
//-------------------------------------------------
// DRAW_BALL: draw ball i in graphic coordinates
//-------------------------------------------------

void  draw_ball(int i)
{
int     x, y;

        x = XBOX + 1 + ball[i].x;
        y = YWIN - FLEV - ball[i].y;
        circlefill(screen, x, y, ball[i].r, ball[i].c);
}
```



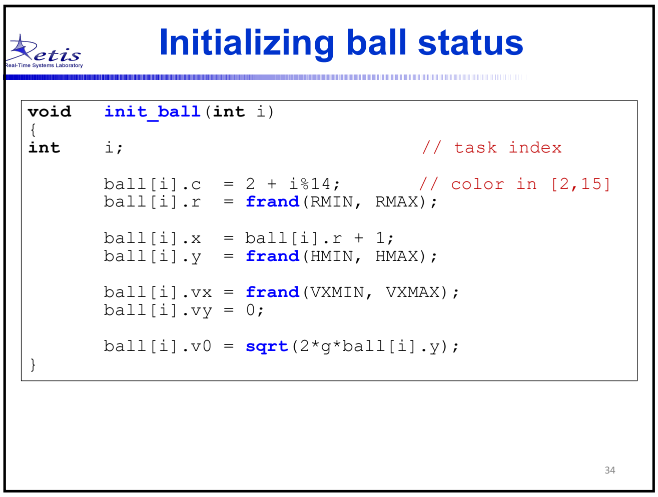
Handling bounces

We need to account for ball dimensions and dumping:



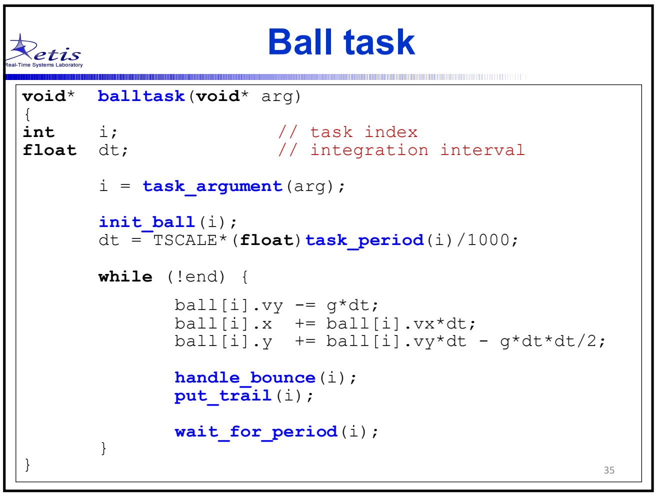
Handling bounces

```
void   handle_bounce(int i)
{
       if (ball[i].y <= ball[i].r) {
              ball[i].y = ball[i].r;
              ball[i].vy = -DUMP*ball[i].vy;
       }

       if (ball[i].y >= HBOX - ball[i].r) {
              ball[i].y = HBOX - ball[i].r;
              ball[i].vy = -DUMP*ball[i].vy;
       }

       if (ball[i].x >= LBOX - ball[i].r) {
              ball[i].x = LBOX - ball[i].r;
              ball[i].vx = -ball[i].vx;
       }

       if (ball[i].x <= ball[i].r) {
              ball[i].x = ball[i].r;
              ball[i].vx = -ball[i].vx;
       }
}
```



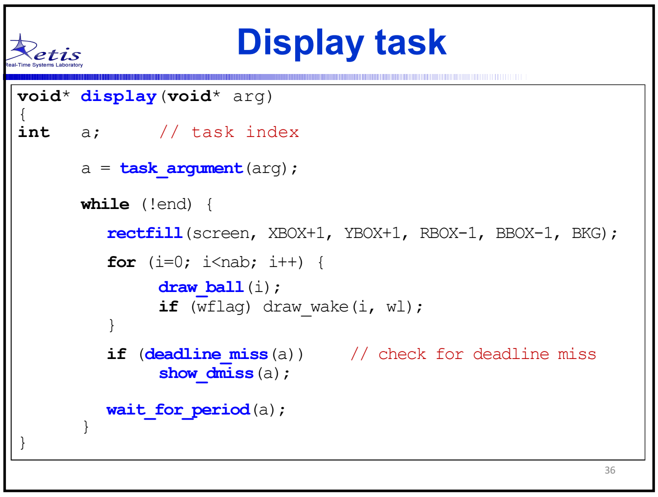
Initializing ball status

```
void   init_ball(int i)
{
int    i;                          // task index

       ball[i].c  = 2 + i%14;      // color in [2,15]
       ball[i].r  = frand(RMIN, RMAX);

       ball[i].x  = ball[i].r + 1;
       ball[i].y  = frand(HMIN, HMAX);

       ball[i].vx = frand(VXMIN, VXMAX);
       ball[i].vy = 0;

       ball[i].v0 = sqrt(2*g*ball[i].y);
}
```



Ball task

```
void*  balltask(void* arg)
{
int    i;               // task index
float  dt;              // integration interval

       i = task_argument(arg);

       init_ball(i);
       dt = TSCALE*(float)task_period(i)/1000;

       while (!end) {

              ball[i].vy -= g*dt;
              ball[i].x  += ball[i].vx*dt;
              ball[i].y  += ball[i].vy*dt - g*dt*dt/2;

              handle_bounce(i);
              put_trail(i);

              wait_for_period(i);
       }
}
```



Display task

```
void* display(void* arg)
{
int    a;      // task index

       a = task_argument(arg);

       while (!end) {

          rectfill(screen, XBOX+1, YBOX+1, RBOX-1, BBOX-1, BKG);

          for (i=0; i<nab; i++) {
                 draw_ball(i);
                 if (wflag) draw_wake(i, wl);
          }

          if (deadline_miss(a))     // check for deadline miss
                 show_dmiss(a);

          wait_for_period(a);
       }
}
```

Main task

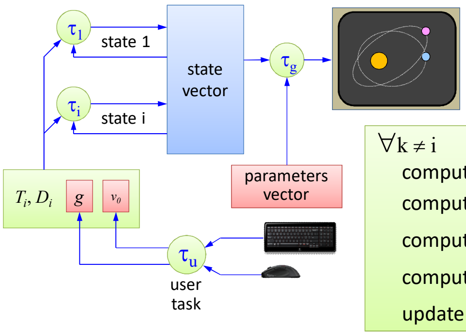
```
int     main(void)
{
int     i;

        init();

        for (i=0; i<=MAX_BALLS; i++)
                wait_for_task_end(i);

        allegro_exit();
        return 0;
}
```

Init function

```
void    init(void)
{
int     i;
char    s[20];

        allegro_init();
        set_gfx_mode(GFX_AUTODETECT_WINDOWED, XWIN, YWIN, 0, 0);
        clear_to_color(screen, BKG);
        install_keyboard();
        srand(time(NULL));        // initialize random generator

        // draw menu area

        // draw box area

        // draw status area

        ptask_init(SCHED_FIFO);
        task_create(MAX_BALLS,   display, PER, DREL, PRI1, ACT);
        task_create(MAX_BALLS+1, interp,  PER, DREL, PRI2, ACT);
}
```

Planets simulation

Using the same programming structure, we can simulate the behavior of N planets subject to a gravitational field:



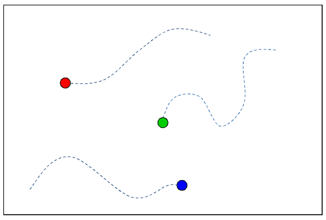
Task τ_i

```
∀k ≠ i
    compute  $d_{ik}$
    compute  $F_{ik} = G \frac{m_i m_k}{d_{ik}^2}$
    compute  $F_i = \sum_{i \neq k} F_{ik}$
    compute  $a_i = F_i / m_i$
    update state $(x_i, v_i)$
```

Simulating pseudo-random motion

Random flies

Let us see how to write a program that simulates the motion of a set of flies that move randomly on a given area of the screen:



General description

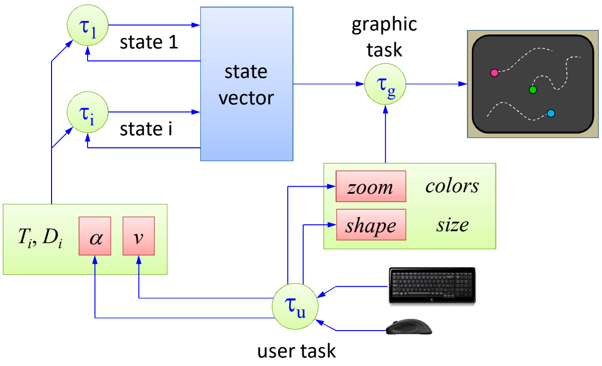
1. Each fly should update its state every period.
2. Fly motion is independent on the other flies, but depends on the current speed and direction.
3. A fly must bounce on the border of the area.
4. The user can create a new fly and vary:
   period, zoom, max deviation, max speed, fly shape.
5. The system provides information on:
   number of created flies, deadline misses, zoom factor.

User interface

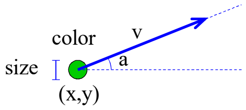
The user can create a new fly and vary:
period, zoom, max deviation, max speed, fly shape.

- **SPACE**: create a new fly;
- **UP/DOWN arrow**: increase/decrease deviation;
- **PAD –**: zoom in; **PAD +**: zoom out
- **F**: change fly shape (toggle)
- **D**: set default parameters
- **ESC** exit the program

Application structure



State variables



```
struct state            // fly structure
{
        int     c;      // color [1,15]
        float   r;      // radius (m)
        float   x;      // x coordinate (m)
        float   y;      // y coordinate (m)
        float   v;      // velocity (m/s)
        float   a;      // orientation angle (rad)
};
```

Global variables

```
struct state fly[MAXFLY];     // fly buffer

int    naf = 0;               // number of active flies

float  scale = 1;             // scale factor

float  dev = MAXDEV;          // deviation factor (deg)

float  v_max = MAXV;          // maximum speed (m/s)

int    period = PER_F;        // period of the fly task

int    end;                   // end flag
```

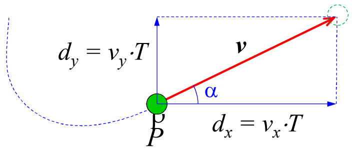
Fly motion rules

Situation at time *t*

$$\vec{P} = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} \qquad \begin{cases} v_x = |\boldsymbol{v}|\cos(\alpha) \\ v_y = |\boldsymbol{v}|\sin(\alpha) \end{cases}$$

Situation at time *t+T*

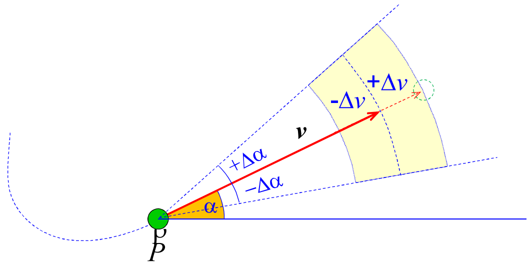
$$\vec{P}' = \vec{P} + \vec{v} \cdot T$$



$$\vec{P}' = \begin{pmatrix} x + d_x \\ y + d_y \end{pmatrix}$$

Fly motion rules

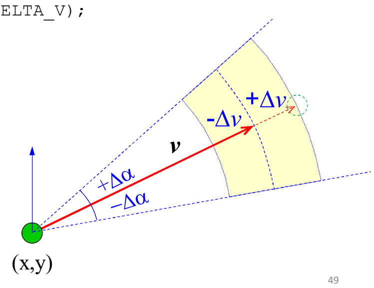
To implement some randomness, $\boldsymbol{v}$ and α can change at every step by a limited amount: $\pm\Delta v$ and $\pm\Delta\alpha$



Fly motion rules

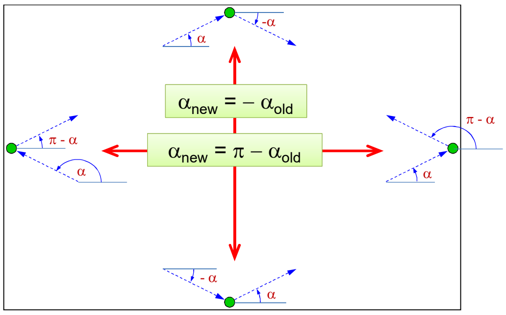
Hence, the position increment can be implemented as follows:

```
da = frand(-DELTA_A, DELTA_A);
dv = frand(-DELTA_V, DELTA_V);

alpha = alpha + da;
v = v + dv;

vx = v*cos(alpha);
vy = v*sin(alpha);

x = x + vx*period;
y = y + vy*period;
```



Handling borders

If flies moves in a box, we need to manage collisions with the borders:

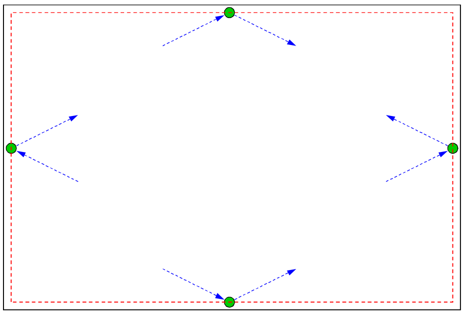


$\alpha_{new} = -\alpha_{old}$

$\alpha_{new} = \pi - \alpha_{old}$

Handling borders

We also need to take fly dimension into account:



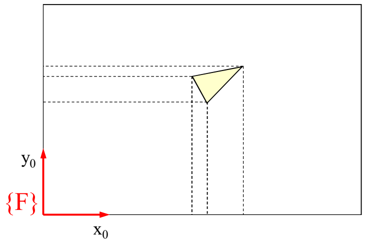
Handling bounces

```
void   handle_bounce(int i)
{
int    outl, outr, outt, outb;

       outl = (fly[i].x <= BOXL+FD);
       outr = (fly[i].x >= BOXR-FD);
       outt = (fly[i].y >= BOXT-FD);
       outb = (fly[i].y <= BOXB+FD);

       if (outl) fly[i].x = BOXL+FD;
       if (outr) fly[i].x = BOXR-FD;
       if (outl || outr) fly[i].a = PI - fly[i].a;

       if (outt) fly[i].y = BOXT-FD;
       if (outb) fly[i].y = BOXB+FD;
       if (outt || outb) fly[i].a = - fly[i].a;
}
```

Auxiliary functions

```
//-----------------------------------------------------
// DRAW_FLY: draws fly i on the screen
//-----------------------------------------------------

void  draw_fly(int i)
{

      // converts world coordinates (x, y, alpha)
      // into display coordinates and draws a fly
      // at position (x,y) with orientation alpha,
      // and color c, taking care of scale factor

}
```

Handling rotations

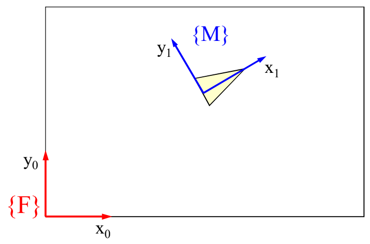


Problem:

To draw the object we have to find the coordinates of each vertex in the fixed reference frame {F}.

Defining a moving frame

As a first step, we have to define a mobile frame {M} attached to the object.

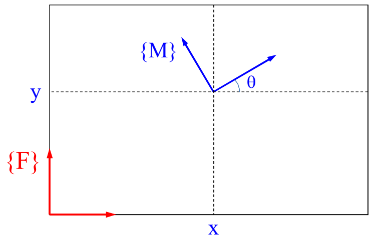


Steps to do

1. Define the variables to express the frame {M}
2. Find the transformation expressing the frame {M} with respect to the frame {F}
3. Express each object vertex with respect to {M}
4. Express each object vertex with respect to {F} using the frame transformation.

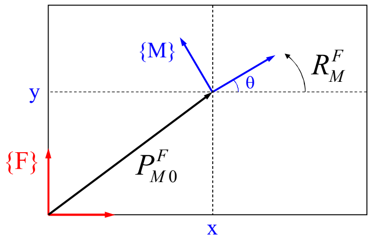
Frame variables

1. Define the variables to express the frame {M}



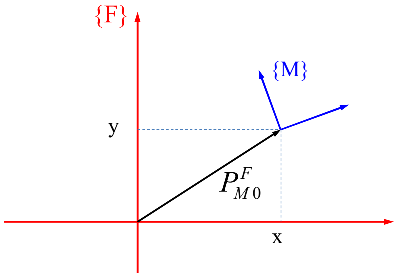
Frame transformation

Frame {M} with respect to {F} can be expressed by a rotation matrix R_M^F and a translation vector P_{M0}^F



Translation vector

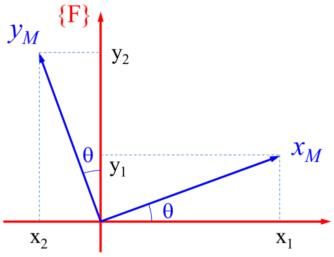
The translation vector expresses the coordinates of the origin of {M} with respect to {F}:



$$P_{M0}^F = \begin{pmatrix} x \\ y \end{pmatrix}$$

Rotation matrix

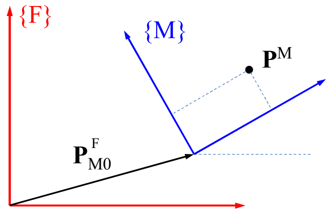
The rotation matrix expresses the coordinates of each versor of {M} with respect to {F}:



$$R_M^F = \begin{array}{c} \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ \begin{array}{cc} \uparrow & \uparrow \\ x_M & y_M \end{array} \end{array}$$

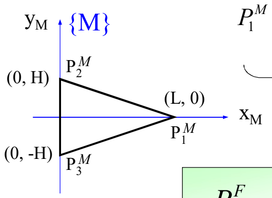
$$R_M^F = \begin{pmatrix} c_\theta & -s_\theta \\ s_\theta & c_\theta \end{pmatrix}$$

Coordinate Transformation



$$P^F = P^F_{M0} + R^F_M \; P^M$$

A simple example

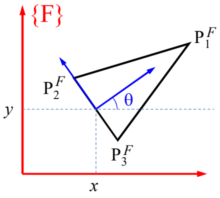


$$\underbrace{P_1^M = \begin{pmatrix} L \\ 0 \end{pmatrix} \quad P_2^M = \begin{pmatrix} 0 \\ H \end{pmatrix} \quad P_3^M = \begin{pmatrix} 0 \\ -H \end{pmatrix}}$$

$$P^F = P^F_{M0} + R^F_M \; P^M$$

$$P^F_{M0} = \begin{pmatrix} x \\ y \end{pmatrix} \qquad R^F_M = \begin{pmatrix} c_\theta & -s_\theta \\ s_\theta & c_\theta \end{pmatrix}$$

A simple example



$$\underbrace{P_1^M = \begin{pmatrix} L \\ 0 \end{pmatrix} \quad P_2^M = \begin{pmatrix} 0 \\ H \end{pmatrix} \quad P_3^M = \begin{pmatrix} 0 \\ -H \end{pmatrix}}$$

$$P^F = P^F_{M0} + R^F_M \; P^M$$

$$\overbrace{P_1^F = \begin{pmatrix} x + Lc_\theta \\ y + Ls_\theta \end{pmatrix} \quad P_2^F = \begin{pmatrix} x - Hs_\theta \\ y + Hc_\theta \end{pmatrix} \quad P_3^F = \begin{pmatrix} x + Hs_\theta \\ y - Hc_\theta \end{pmatrix}}$$

Drawing a triangular fly

```
void  draw_fly(int i)
{
float px1, px2, px3, py1, py2, py3; // world coord.
float ca, sa;

      ca = cos(fly[i].a);
      sa = sin(fly[i].a);

      p1x = fly[i].x + FL*ca; // nose point
      p1y = fly[i].y + FL*sa;
      p2x = fly[i].x - FW*sa; // left wing
      p2y = fly[i].y + FW*ca;
      p3x = fly[i].x + FW*sa; // right wing
      p3y = fly[i].y - FW*ca;

      triangle(screen,
            XCEN+p1x/scale, YMAX-YCEN-p1y/scale;
            XCEN+p2x/scale, YMAX-YCEN-p2y/scale;
            XCEN+p3x/scale, YMAX-YCEN-p3y/scale;
            fly[i].c);
}
```

Initializing fly status

```
void  init_fly(int i)
{
      fly[i].c = 2 + i%14;        // fly color [2,15]
      fly[i].r = FL;              // fly length
      fly[i].x = 0;               // x initial position
      fly[i].y = 0;               // y initial position
      fly[i].v = VEL;             // initial velocity
      fly[i].a = frand(0,2*PI);   // initial orientation
}
```

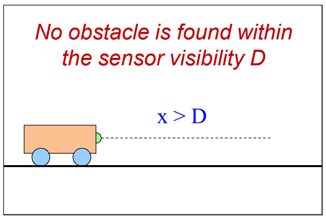
Fly task

```
void*  flytask(void* arg)
{
int    i;                  // task index
float  dt;                 // integration interval

       i = task_argument(arg);

       init_fly(i);
       dt = (float)task_period(i)/1000;

       while (!end) {

            da = frand(-dev,dev);     // var. (deg)
            fly[i].a += da*PI/180;    // fly angle (rad)

            vx = fly[i].v * cos(fly[i].a);
            vy = fly[i].v * sin(fly[i].a);

            fly[i].x = fly[i].x + vx*dt;
            fly[i].y = fly[i].y + vy*dt;

            handle_bounce(i);
            wait_for_period(i);
       }
}
```

Simulating Sensors

Distance sensors

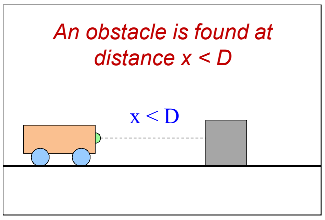
Proximity sensors can be simulated by reading pixels along the sensing direction (up to a maximum distance D) and returning the distance to the first pixel with color ≠ background:



No obstacle is found within the sensor visibility D

Distance sensors

Proximity sensors can be simulated by reading pixels along the sensing direction (up to a maximum distance D) and returning the distance to the first pixel with color ≠ background:

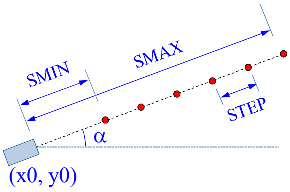


An obstacle is found at distance x < D

Implementation

We have to decide which parameters are variable and which are constant.

Reasonable compromise:



```
#define  SMIN   10        // minimum sensor distance
#define  SMAX   100       // maximum sensor distance
#define  STEP   1         // sensor resolution

int      x0, y0;          // sensor coordinates
float    alpha;           // sensing direction
```

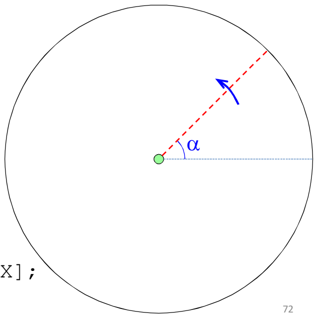
Implementation

```
//----------------------------------------------------
// READ_SENSOR: return the distance of the first
// pixel != BKG found from (x,y) along direction alpha
//----------------------------------------------------

int     read_sensor(int x0, int y0, float alpha)
{
int     c;                          // pixel value
int     x, y;                       // sensor coordinates
int     d = SMIN;                   // min sensor distance

        do {
            x = x0 + d*cos(alpha);
            y = y0 + d*sin(alpha);
            c = getpixel(screen, x, y);
            d = d + SSTEP;
        } while ((d <= SMAX) && (c == BKG));

        return d;
}
```

Simulating a radar

A full radar scan consists of a sequence of line scans, each performed along a different direction α.

If a line scan contains RMAX points and is done with an angular increment dα = 1 deg, then the radar image can be stored into a matrix:



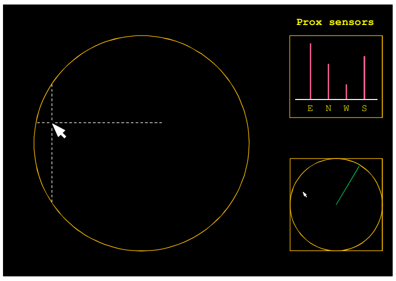
```
#define RMAX  180
#define ARES  360

int     radar[ARES][RMAX];
```

Line scan

```
void    line_scan(int x0, int y0, int a)
{
int     d;
float   alpha;

        alpha = a*PI/180;        // angle in rads

        for (d=RMIN; d<=RMAX; d+=RSTEP) {
                x = x0 + d*cos(alpha);
                y = y0 - d*sin(alpha);
                radar[a][d] = getpixel(screen, x, y);
        }
}
```

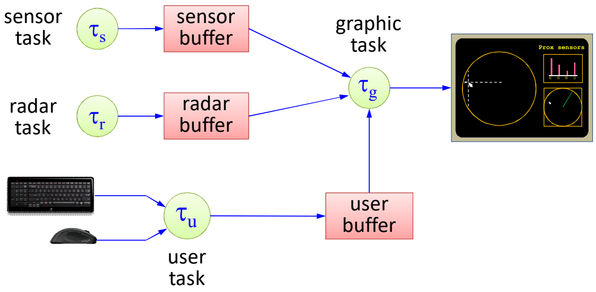
Sensing application

Let us see how to implement an application that uses 4 distance sensors around the mouse and a radar.



Task-resource diagram

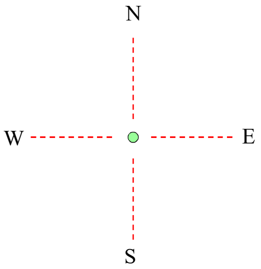
The application can be structured with 4 tasks and 3 shared buffers:



Implementation

An array of integers can be used to store the values of the 4 sensors around the mouse:

```
int    sens[4];      // array for the 4 values (E,N,W,S)
```



Sensor task

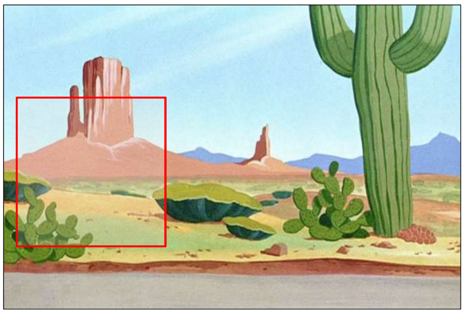
```
void*   sensortask(void* arg)
{
int     i, j;          // task and sensor index
int     x, y;          // sensors coordinates
float   alpha;         // scanning direction

        i = task_argument(arg);

        while (!end) {
                x = mouse_x;
                y = mouse_y;
                for (j=0; j<4; j++) {
                        alpha = j*PI/2;
                        sen[j] = read_sensor(x, y, alpha);
                }
                wait_for_period(i);
        }
}
```

Radar task

```
void*   radartask(void* arg)
{
int     i;              // task index
float   a = 0;          // scanning direction (deg)

        i = task_argument(arg);

        while (!end) {
                line_scan(XRAD, YRAD, a);

                a = a + 1;
                if (a == 360) a = 0;

                wait_for_period(i);
        }
}
```

A video camera

We can simulate a camera by reading a matrix of pixels of given size at a desired location:



Acquiring an image

```
int     image[HRES][VRES];         // global image buffer

//-------------------------------------------------------
// GET_IMAGE   reads an area of the screen centered in
//             (x0,y0) and stores it into image[][]
//-------------------------------------------------------

void    get_image(int x0, int y0)
{
int     i, j;                          // image indexes
int     x, y;                          // video coordinates

        for (i=0; i<HRES; i++)
            for (j=0; j<VRES; j++) {
                x = x0 - HRES/2 + i;
                y = y0 - VRES/2 + j;
                image[i][j] = getpixel(screen, x, y);
            }
}
```

Displaying an image

```
//-------------------------------------------------------
// PUT_IMAGE   displays the image stored in image[][]
//             in an area centered in (x0,y0)
//-------------------------------------------------------

void    put_image(int x0, int y0)
{
int     i, j;                     // image indexes
int     x, y;                     // video coordinates

        for (i=0; i<HRES; i++)
            for (j=0; j<VRES; j++) {
                x = x0 - HRES/2 + i;
                y = y0 - VRES/2 + j;
                putpixel(screen, x, y, image[i][j]);
            }
}
```

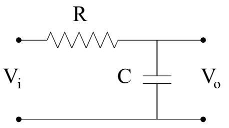
Camera task

```
//-------------------------------------------------------
// Task that periodically gets images from position
// (XCAM,YCAM) and displays them in position (XD,YD)
//-------------------------------------------------------

void*   cameratask(void* arg)
{
int     i;                             // task index

        i = task_argument(arg);

        while (!end) {

            get_image(XCAM, YCAM);
            put_image(XD, YD);

            wait_for_period(i);
        }
}
```

Simulating

Filters and Motors

Low-pass Filter



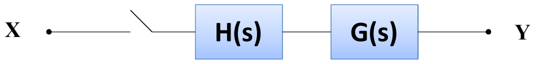
$$\begin{cases} V_o = \dfrac{1}{Cs} I \\ I = \dfrac{V_i - V_o}{R} \end{cases} \qquad V_o = \frac{V_i - V_o}{RCs}$$

$$V_o(RCs+1) = V_i \quad \Rightarrow \quad \frac{V_o}{V_i} = \frac{1}{RCs+1}$$

defining $\alpha = \dfrac{1}{RC}$ we have: $\boxed{G(s) = \dfrac{\alpha}{s+\alpha}}$

Computing the Z-transform

The discrete expression of G(s) can be derived by using a sampling and hold circuit:



$$G(z) = \frac{Y(z)}{X(z)} = Z\left[\left(\frac{1-e^{-st}}{s}\right)G(s)\right] = (1-z^{-1})Z\left[\frac{G(s)}{s}\right]$$

Computing the Z-transform

For the low-pass filter we have :

$$G(z) = (1-z^{-1})Z\left[\frac{\alpha}{s(s+\alpha)}\right]$$

$$G(z) = \frac{(z-1)}{z}\frac{(1-p)z}{(z-1)(z-p)}$$

where

$$\left(p = e^{-\alpha T}\right)$$

$$G(z) = \frac{1-p}{z-p} = \frac{(1-p)z^{-1}}{1-pz^{-1}}$$

Discrete time expression

From the Z-transfer function we derive:

$$\frac{Y(z)}{X(z)} = \frac{(1-p)z^{-1}}{1-pz^{-1}}$$

$$\boxed{Y(z) = pY(z)z^{-1} + (1-p)X(z)z^{-1}}$$

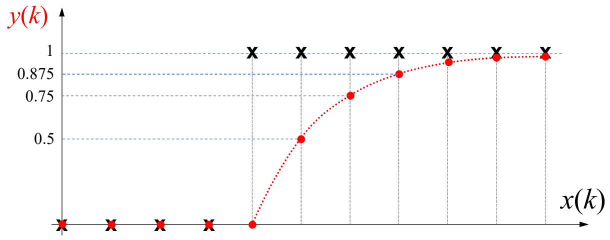
$$\Downarrow$$

$$\boxed{y(k) = py(k-1) + (1-p)x(k-1)}$$

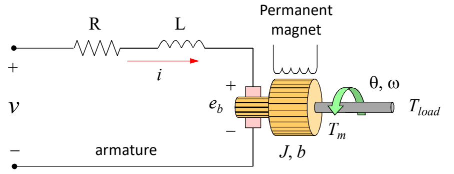
Discrete time expression

$$\boxed{y(k) = py(k-1) + (1-p)x(k-1)}$$

Example with $p = 0.5$

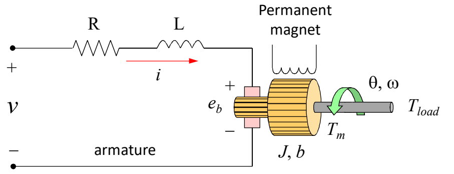


Modeling a DC Motor



Electrical parameters		Mechanical parameters	
v	Input voltage	θ	Motor angle
e_b	Back electromotive force	ω	Motor speed
i	motor current	T_m	Motor torque
R	Electric resistance	J	Moment of inertia
L	Electric inductance	b	Damping coefficient

Electrical equations

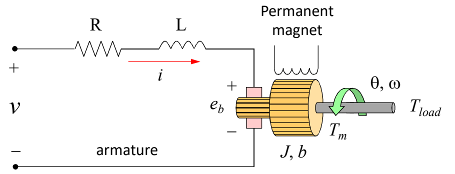


$$v = Ri + L\frac{di}{dt} + e_b$$

$e_b = K_b\dot{\theta}$ K_b = back electromotive force (EMF) constant

$$\boxed{v = Ri + L\frac{di}{dt} + K_b\dot{\theta}} \Rightarrow \boxed{V = (R+Ls)I + K_b\dot{\Theta}}$$

Mechanical equations



$$T_m - T_{load} = J\ddot{\theta} + b\dot{\theta}$$

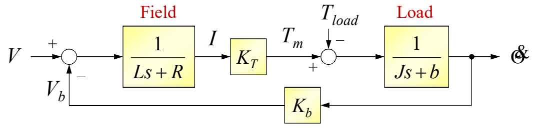
$$T_m = K_T i \qquad K_T = \text{Motor torque constant}$$

$$\boxed{K_T i - T_{load} = J\ddot{\theta} + b\dot{\theta}} \;➡\; \boxed{K_T I - T_{load} = (Js+b)\dot{\theta}}$$

DC Motor block diagram

$$V = (Ls+R)I + K_b\dot{\theta} \;➡\; I = \frac{V - K_b\dot{\theta}}{Ls+R}$$

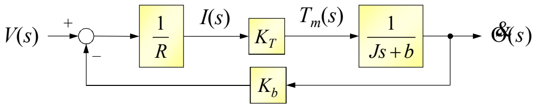
$$K_T I - T_{load} = (Js+b)\dot{\theta} \;➡\; \dot{\theta} = \frac{K_T I - T_{load}}{Js+b}$$



$$\boxed{\dot{\theta} = \frac{K_T V - T_{load}(Ls+R)}{(Ls+R)(Js+b) + K_T K_b}}$$

Simplifying the model

In practice, L can be neglected due to its small value, meaning that the response is dominated by the slow mechanical pole. Also, if there is no disturbance torque ($T_{load} = 0$), we have:

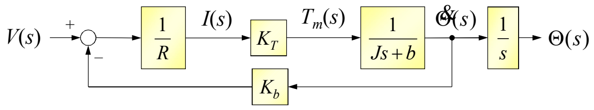


$$\boxed{\frac{\dot{\Theta}(s)}{V(s)} = \frac{K_T}{R(Js+b) + K_T K_b} = \frac{K}{\tau s + 1}}$$

where: $\begin{cases} K = \dfrac{K_T}{Rb + K_T K_b} \\ \tau = \dfrac{RJ}{Rb + K_T K_b} \end{cases}$

Simplifying the model

If we are interested in finding the angular response of the motor, we need to integrate the speed, so obtaining:

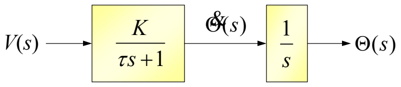


$$\boxed{\frac{\Theta(s)}{V(s)} = \frac{\dot{\Theta}(s)}{sV(s)} = \frac{K}{s(\tau s + 1)}}$$

$$\begin{cases} K = \dfrac{K_T}{Rb + K_T K_b} \\ \tau = \dfrac{RJ}{Rb + K_T K_b} \end{cases}$$

Constant	*Value*	*Units*
K_T	0.05	N m / A
K_b	0.05	V s / rad
R	1	Ω
J	$5\cdot10^{-4}$	N m s²/ rad
b	$1\cdot10^{-4}$	N m s / rad
K	19.23	rad V^{-1} s^{-1}
τ	0.19	s

Summarizing



$$\boxed{G(s) = \frac{\Theta(s)}{V(s)} = \frac{K}{s(\tau s + 1)}}$$

Discretizing using the sampling and hold method, *G(z)* becomes:

$$G(z) = (1 - z^{-1})\,Z\left[\frac{G(s)}{s}\right] = \left(\frac{z-1}{z}\right) Z\left[\frac{K}{s^2(\tau s + 1)}\right]$$

Computing the Z-transform

And since: $Z\left[\dfrac{1}{s^2(\tau s + 1)}\right] = \dfrac{T\,z}{(z-1)^2} - \dfrac{\tau(1-p)z}{(z-1)(z-p)}$

with $\boxed{p = e^{-T/\tau}}$ (if $T = 20\ ms$, then $p = 0.9$)

we have: $G(z) = K\left(\dfrac{z-1}{z}\right)\left[\dfrac{T\,z}{(z-1)^2} - \dfrac{\tau(1-p)z}{(z-1)(z-p)}\right]$

Computing the Z-transform

$$G(z) = K\left[\frac{T}{(z-1)} - \frac{\tau(1-p)}{(z-p)}\right]$$

$$G(z) = K\left[\frac{(T-\tau+p\tau)z+(\tau-pT-p\tau)}{z^2-(1+p)z+p}\right]$$

$$G(z) = \frac{Az+B}{z^2-(1+p)z+p} \quad \text{where: } \begin{cases} A = K(T-\tau+p\tau) \\ B = K(\tau-pT-p\tau) \end{cases}$$

$$G(z) = \frac{Az^{-1}+Bz^{-2}}{1-(1+p)z^{-1}+pz^{-2}}$$

Discrete time expression

$$\frac{\Theta(z)}{V(z)} = \frac{Az^{-1}+Bz^{-2}}{1-(1+p)z^{-1}+pz^{-2}}$$

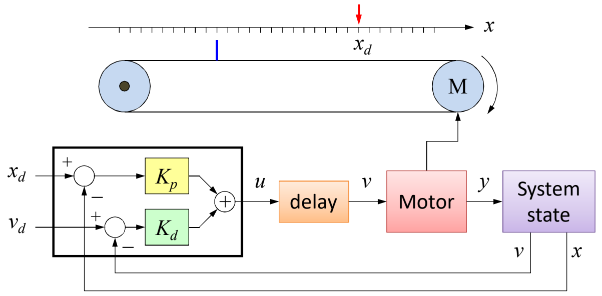
$$\Theta(z)-(1+p)\Theta(z)z^{-1}+p\Theta(z)z^{-2} = AV(z)z^{-1}+BV(z)z^{-2}$$

$$\Downarrow$$

$$\theta(k) = Av(k-1)+Bv(k-2)+(1+p)\theta(k-1)-p\theta(k-2)$$

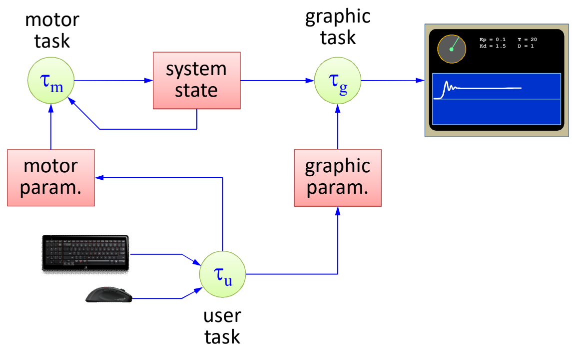
$$\text{where: } \begin{cases} p = e^{-T/\tau} \\ A = K(T-\tau+p\tau) \\ B = K(\tau-pT-p\tau) \end{cases}$$

Motor control application



Note: $\begin{cases} v_d = 0 & \Rightarrow \text{PD position control} \\ K_p = 0 & \Rightarrow \text{P velocity control} \end{cases}$

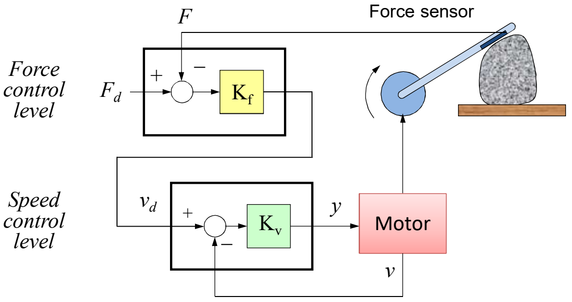
Task-resource diagram



Motor task

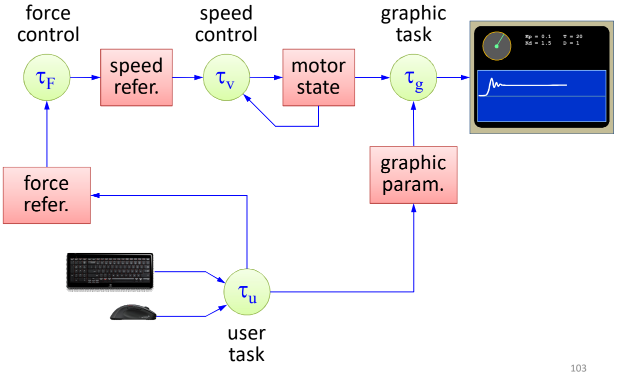
```
void* mototask(void* arg)
{
int     i;          // task index
float   x, v;       // actual  position and speed
float   xd, vd;     // desired position and speed
float   u, v, y;    // temporay variables

        i = task_argument(arg);

        while (!end) {

                get_setpoint(&xd, &vd);
                get_state(&x, &v);
                u = KP*(xd - x) + KD*(vd - v);
                v = delay(u);
                y = motor(v);
                unpdate_state(y);
                wait_for_period(i);
        }
}
```

Two-level control



Note that inner control loop must run faster!

Task-resource diagram



force control

τ_F

speed refer.

speed control

τ_v

motor state

graphic task

τ_g

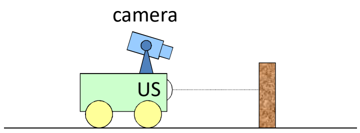
force refer.

graphic param.

τ_u

user task

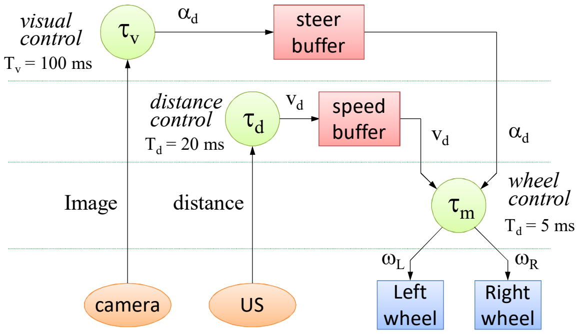
Using visual feedback



Three control loops

1. Steering control (driven by vision)
2. Speed control (driven by distance sensing)
3. Low-level wheel control

Three-level control



visual control

$T_v = 100$ ms

τ_v

α_d

steer buffer

distance control

$T_d = 20$ ms

τ_d

v_d

speed buffer

v_d

α_d

Image

distance

τ_m

wheel control

$T_d = 5$ ms

ω_L

ω_R

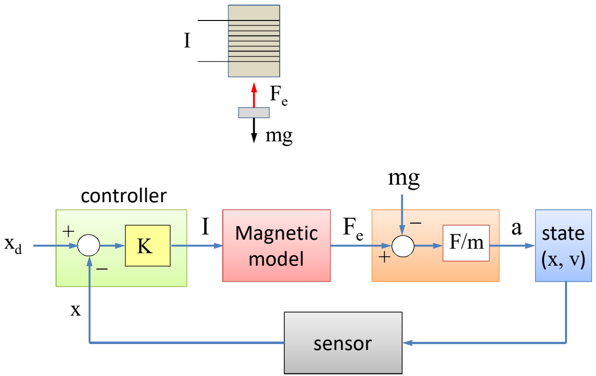
camera

US

Left wheel

Right wheel

Magnetic Levitation



I

F_e

mg

controller

x_d

+

−

K

I

Magnetic model

F_e

mg

−

+

F/m

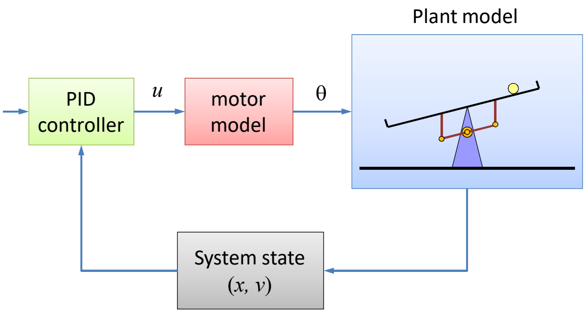
a

state (x, v)

x

sensor

Ball and Beam



Plant model

PID controller

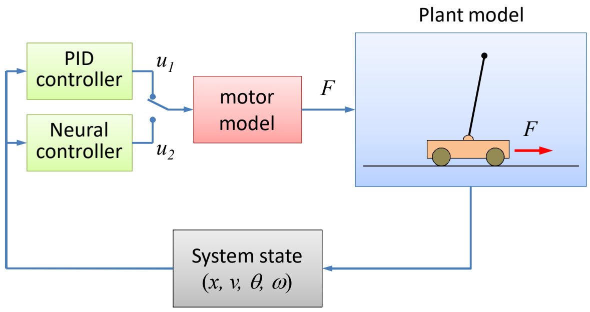
u

motor model

θ

System state (x, v)

Inverted pendulum



Plant model

PID controller

Neural controller

u_1

u_2

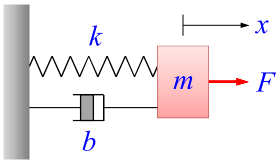
motor model

F

F

System state (x, v, θ, ω)

Mass-spring-damper



m = mass [Kg = Ns^2/m]
k = elastic coefficient [N/m]
b = damping coefficient [Ns/m]

$$F(t) = m\ddot{x} + b\dot{x} + kx$$

$$F(s) = ms^2X(s) + bsX(s) + kX(s)$$

$$G(s) = \frac{X(s)}{F(s)} = \frac{1}{ms^2 + bs + k}$$

Mass-spring-damper

$$G(s) = \frac{X(s)}{F(s)} = \frac{1}{ms^2 + bs + k}$$

Bilinear transform (Tustin):

$$G(z) = G(s)\Big|_{s=\frac{2}{T}\frac{z-1}{z+1}} = \frac{z^2 + 2z + 1}{Az^2 + Bz + C}$$

where
$$\begin{cases} A = \dfrac{4m}{T^2} + \dfrac{2b}{T} + k \\ B = 2k - \dfrac{8m}{T^2} \\ C = \dfrac{4m}{T^2} - \dfrac{2b}{T} + k \end{cases}$$

Mass-spring-damper

$$\frac{X(z)}{F(z)} = \frac{1 + 2z^{-1} + z^{-2}}{A + Bz^{-1} + Cz^{-2}}$$

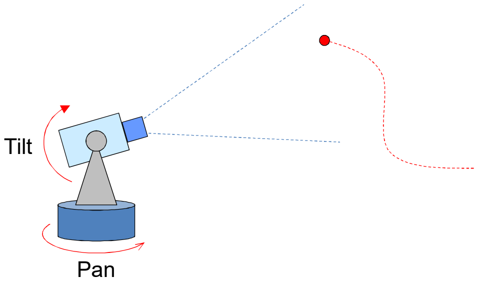
$$X = \frac{1}{A}\left(F + 2Fz^{-1} + Fz^{-2} - BXz^{-1} - CXz^{-2}\right)$$

$$\Downarrow$$

$$x(k) = \frac{1}{A}\left[F(k) + 2F(k-1) + F(k-2) - Bx(k-1) - Cx(k-2)\right]$$

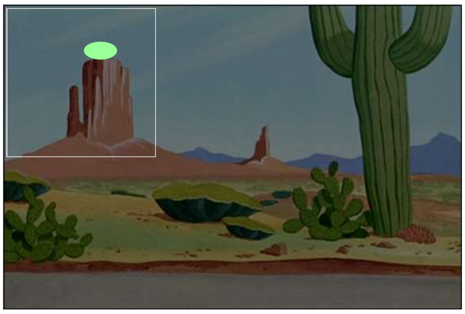
Tracking camera

Now suppose we want to simulate a pan-tilt camera for tracking moving objects:



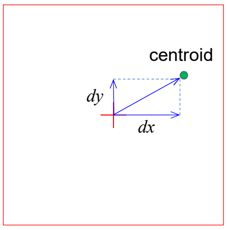
Scanning the image

In a simplified scenario, you can consider bright moving objects on a dark background:



Control reference

To focus the object at the center of the visual field, the camera has to be moved towards the centroid:

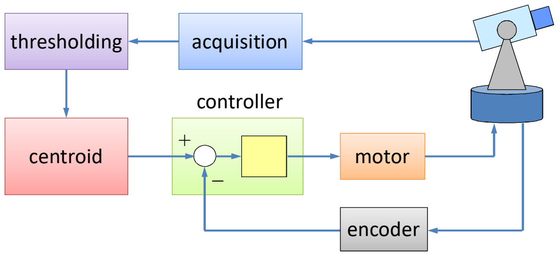


Hence, the centroid coordinates become the set points for the controller

Steps to achieve the goal

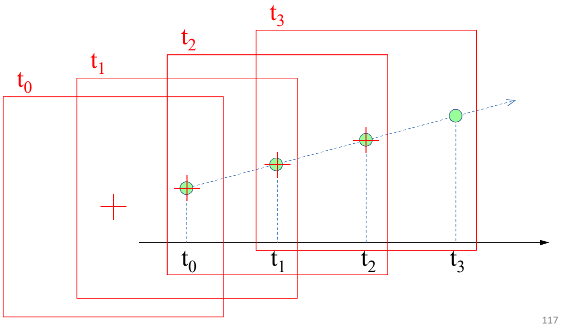
1. Image scanning → Read the pixels color in a given window
2. thresholding → Discard the pixels with dark color
3. centroid computation → Compute the centroid of pixels with light color
4. camera control → Control the camera axes to move to the centroid
5. motor simulation → Simulate the motor to achieve a realistic motion

Camera Tracking



Control reference

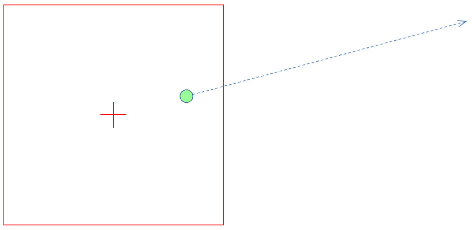
Suppose the object is moving slowly at constant speed v on a straight line:



Control reference

The consequence is that the moving window is always behind the object, with a position error = v T:

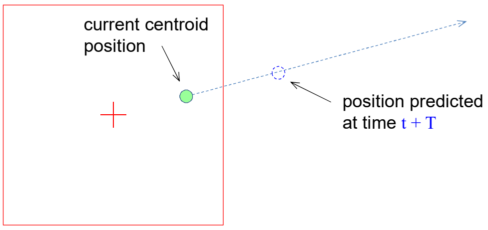
If the object stops,
the error is zero:



Using predictions

We can reduce the error by moving the camera

- NOT at the current centroid position (time t)
- but at the predicted position it will have at t + T:



How to make predictions

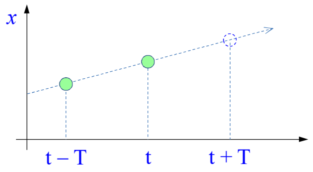
The simplest prediction can be done assuming that the target moves at a constant speed:

$$v(t) = \frac{x(t) - x(t-T)}{T}$$

$$x(t+T) = x(t) + v(t)T$$

$$\Downarrow$$

$$\boxed{x(t+T) = 2x(t) + x(t-T)}$$



A better way is to use a Kalman filter.