INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR SAMARJIT CHAKRABORTY

Embedded Systems Programming Laboratory

Project - Pew pew Asteroids

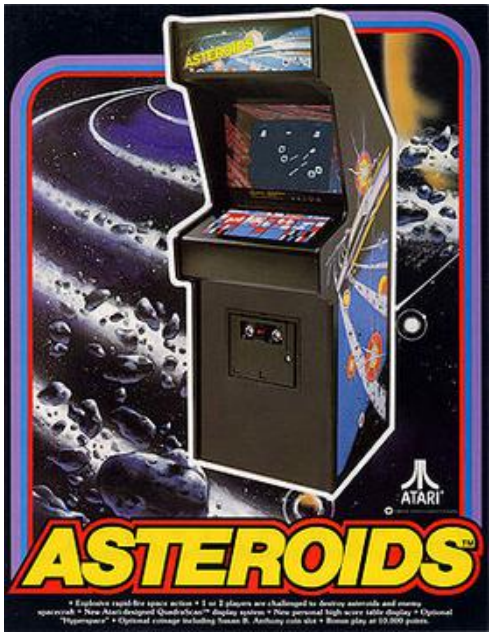
WS 2019/20

Alex Hoffman

November 21, 2019

Project Description

This semester's project will be a re-implementation of the famous Asteroids game which has been published in many different variants. As one example, we will look at the original arcade game. It was designed by Atari, Inc in 1979. You can have a look at the game play on **Youtube**.



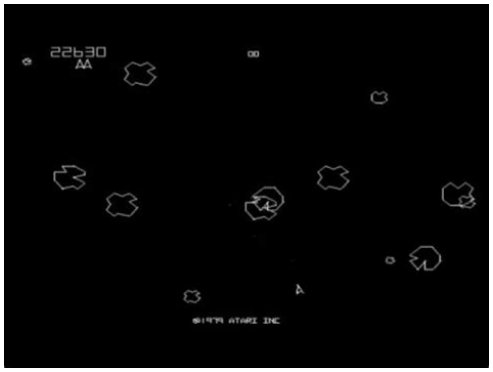


Figure 1: PAC-MAN Source: wikipedia

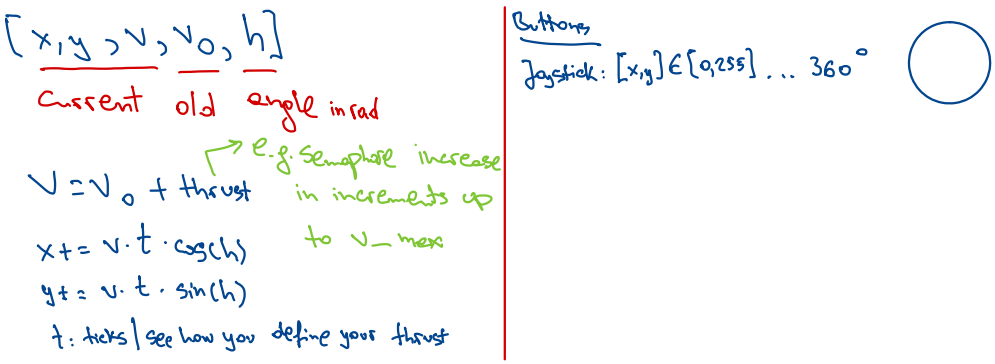
Asteroids

Your task will be to implement the Asteroids game for both one and two player modes. Note that the Asteroids two player mode was not a part of the original game. Please find the game requirements for the successful completion of the lab described below. On purpose, we have not determined all implementation details so that there is room for your creativity. Have a look at the original game for inspirations.

Game Setting

Asteroids gameplay should be modeled after the original game as closley as possible (see **wikipedia** for details). Important features are the following :

- Map should be as close to original as possible.
- The players ship should rotate left and right as well as being able to shoot forwards and thrust forward.
- The ship should have appropriate physics applied such that it continues drifting after a player stops thrusting, this should include the ability to "drift" in curved trajectories.
- Large asteroids should be spawned at the beginning of the game that break up into smaller asteroids when shot, asteroid graphics should not all be the same.



- All objects should wrap around the screen.
- The player's ship should randomly spawn somewhere at the screen's edge when leaving the screen.
- Hitting asteroids should give the player points, which should be displayed on the screen along with the player's remaining lives. The smaller the asteroid the larger the reward should be.
- Twice during each level two flying saucers should appear, see wikipedia for details.
- Once all asteroids have been cleared and the saucers destroyed the player progresses to the next level where the asteroids spawn again, this time with more than the last level.
- The game should contain a minimum of 3 levels.
- The player should gain an extra life after achieving a reasonable score.

Cheats

For testing there should be an option to set cheat values before starting the game. The cheat values should include the following options

- Infinite lives
- Starting score can be set
- Starting level can be set

Two Player Mode

The multiplayer mode will put the two players against each other. One player will control one of the saucers while the other controls the spaceship. Gameplay with be otherwise normal.

Make sure that both your boards are synchronized. This means that both boards show the same game scene at the same time. There should be no noticeable lag between the boards.

High Score

Save the high score for the one and two player mode separately. It is enough to save the high score for one game session. One game session is defined as the phase between turning the RToy on and off.

Menu and Controls

You have learned how to use the joystick and the buttons in the introductory exercise. Choose an appropriate way for the game navigation on your own. Think of a way to design a start menu where you can select the different player modes and the high score. Also keep in mind that your game should be stoppable (pause mode as well as ending the game). Design your game in such a way that a person can learn the controls without needing to read the manual.

Error Handling

You game should be able to handle common errors generated by real world events. Such as the UART cable becoming disconnected during gameplay. This about all the possible errors that could occur and make sure they are handled as you would expect a consumer product to handle them.

Frame Rate

The frame rate for your game should be at least **50 frames per second**. Moreover, the frame rate should be consistent all time during the game play. Display the frame rate which you dynamically calculate during the game play on the screen.

2 Requirements to fulfill

This sections describes the requirements you have to fulfill for your project to be accepted at the end of the semester.

2.1 Functional Requirements

Your game has to work according to the description above, this means:

- Functional one- and two player mode
- Logically structured menu and game play (switching from one program state to another)
- Proper navigation functionality (which part of the controller is doing what and when?)
- Meaningful design
- Run your game at a frame rate of at least **50** frames per second

Moreover, you should think yourself about possible pitfalls that can occur during the game play and come up with a solution, for example:

- The two player mode can only be started with the UART cable attached.
- What happens if the UART cable is removed during the game play?
- etc.

2.2 Structural Requirements

As the goal of this course is to get to know how to use an RTOS, you need to use FreeRTOS in your project:

- Divide your program into meaningful and logical tasks - at least five.
- Think about a proper prioritization for your tasks and implement it. You need to use at least two different priorities.
- Use FreeRTOS synchronization mechanisms for inter-task communication.
- Secure all shared variables with semaphores or use other mechanisms that are appropriate.

2.3 Formal Requirements

Formal requirements mean the outer form of your program as well as delivering it in time:

- Make sure to follow the coding style guidelines which were introduced in the beginning of the lab (exercise 2)! The coding style will be part of your final grade! As reminder, you can find the coding style guidelines at the end of this document.
- Deliver your solution in time! The deadline to present it to the supervisor is **July 17th, 2019** during the lab session. We will fix a time slot for every group. Feel free to present your solution before that. After presenting your code, upload it to the Moodle platform.

3 Programming Style

Despite the correct functionality of the code, you also practice good programming style in this lab. Good programming style is not only important for you to keep your code structured and keep track of what is going on in your program. It is especially important when you are working in a group or have to provide the code to others. Therefore, we pay special attention to writing comments, variable/function naming and modularization of your code.

3.1 Modularization

Write own functions for code which is used multiple times and call them from your tasks. Code should be separated into separate files (.c/.h) to modularize the components of the program. For example, all button code should be in the files buttons.h and buttons.c. Even if the code is executed only once, it is recommended to use functions for logically different work/tasks. They structure your code and make it easier to understand.

3.2 Using constants

When writing a program, you often want to specify either fixed values or protocols. Both involve using some predefined values throughout the program while. Fixed values can be the size of an array, the number of elements in some data structure, etc. Protocols usually use flags to indicate which data set is currently being transmitted. For both use cases, it makes sense to use constants instead of writing down the number which is used in the specific case. There are reasons why using constants is a good choice:

- Using names instead of number makes the code more readable.
- Constants are defined once and used in multiple places.
  - If you change the constant in one place, it is less error prone than changing values spread all over your program.
  - It saves you a lot of time you spend on changing every single value and on looking for errors which occur because you have forgotten to change a value somewhere in your code.

Bad example:

```
switch (E) {
        case A printf("1");
        case B printf("2");
        case C printf("3"));
        case D printf("4");
}
```

Good example (also pay attention to the variable naming):

```
switch (motor_command) {
        case C_NO_SPEED printf("stop");
        case C_LOW_SPEED printf("slow");
        case C_HIGH_SPEED printf("fast"));
```

```
        case C_AVERAGE_SPEED printf("accelerating");
}
```

3.3 Writing comments

Writing comments is important for you to remember after a long time what you actually have programmed. Many programmers ask themselves what the intention of their code was after some time has passed by. Further, comments are essential when you give the code to some other programming team (or your lab tutor/supervisor).

Refer to the **Doxygen** styling for a good, and comprehensive, set of commenting style guidelines.

In the following, we provide some basic rules on writing comments:

1. Always add a header to every .c and .h file you wrote. It should contain at least your name, the date and the project the file belongs to. It should also contain a short description of the file contents.
2. Each module (and sub-module) needs a header with a short description of what it is doing. Further, it needs a description of its input and output parameters if not clear from the naming or some extra explanation is necessary. For example, the unit of a parameter has to be specified (mm, m, km, ms, us, s, min, ...). It has to be clear what the function does with which values.
3. You need to comment the *logic* behind your code. Your algorithms have to become clear.

   ```
   counter = counter + 1;
   ```

   Bad comment: Add 1 to counter.
   Good comment: Increasing the line counter after detecting a new line.
4. Variable and function names should fit the context of your code. They also help to understand what the program is doing. So you should choose proper variable names in a language that everybody understands (for this course English or German).

   ```
   int8t A, B, C; % bad naming
   int8t motor_left, motor_right; % good naming
   ```

3.4 Indentation

To make your code readable to yourself and to others, indentation is very important. Correct indentation makes it easy to identify logically connected code blocks.
Bad examples:

1. No indentation (and no comments either):

   ```
   int main() {
   pololu_platform_init();
   if (!initRTPL()) {
   pololu_beep(2);
   return -1;
   }
   while (isRtplRunning()) {
   pololu_sensors_update();
   ```

```
doCallbacks();
stepRTPL();

bt_poll();
if (BTPacket_In.complete) {
bt_interpret(&BTPacket_In);
BTPacket_In.complete = 0;
BTPacket_In.data_length = 0;
}
}
pololu_beep(10);
return 0;
}
```

2. No new lines:

```
int main() {
pololu_platform_init(); if (!initRTPL()) { pololu_beep(2); return -1; }
while (isRtplRunning()) { pololu_sensors_update(); doCallbacks();        stepRTPL(); bt_poll();
if (BTPacket_In.complete) { bt_interpret(&BTPacket_In); BTPacket_In.complete = 0;
BTPacket_In.data_length = 0; } } pololu_beep(10); return 0; }
```

Good examples:

```
int main() {

        // reset robot, set timers, interrupts, UART, ...
        pololu_platform_init();

        if (!initRTPL()) {
                pololu_beep(2);
                // trying to initialize RTPL kernel failed. This is odd.
                return -1;
        }

        // invoke RTPL kernel again and again...
        while (isRtplRunning()) {
                // fetch values
                pololu_sensors_update();
                // pass certain events to RTPL asynchronously
                doCallbacks();
                // trigger RTPL for next cycle.
                stepRTPL();

                /*****************
                 *  BLUETOOTH
                 ****************/
                bt_poll();
                if (BTPacket_In.complete) {
                        // trigger actions here
                        bt_interpret(&BTPacket_In);
                        BTPacket_In.complete = 0;
                        BTPacket_In.data_length = 0;
                }
        }
```

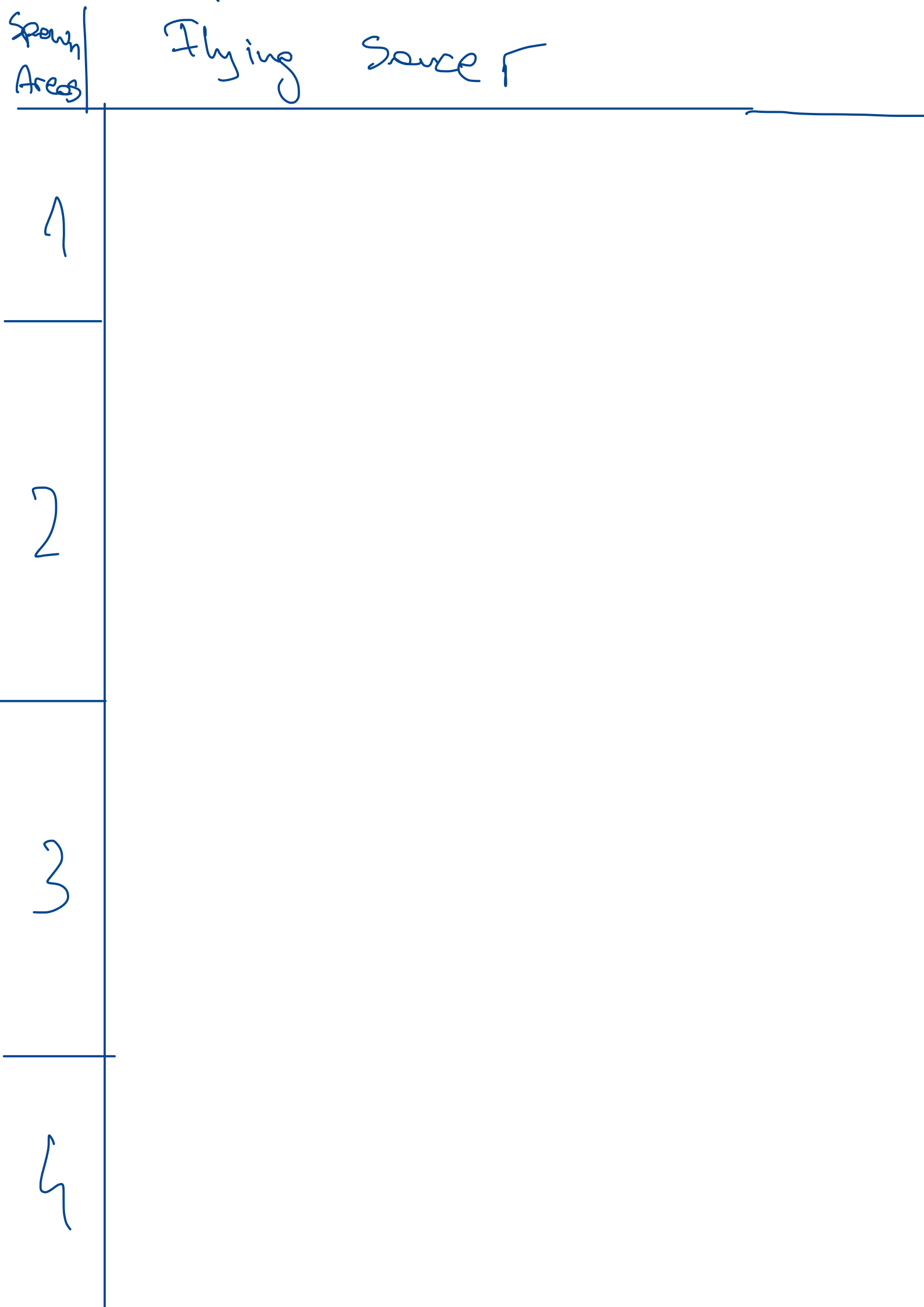
```
        pololu_beep(10);
        // normal, though unwanted exit.
        return 0;
}
```

or

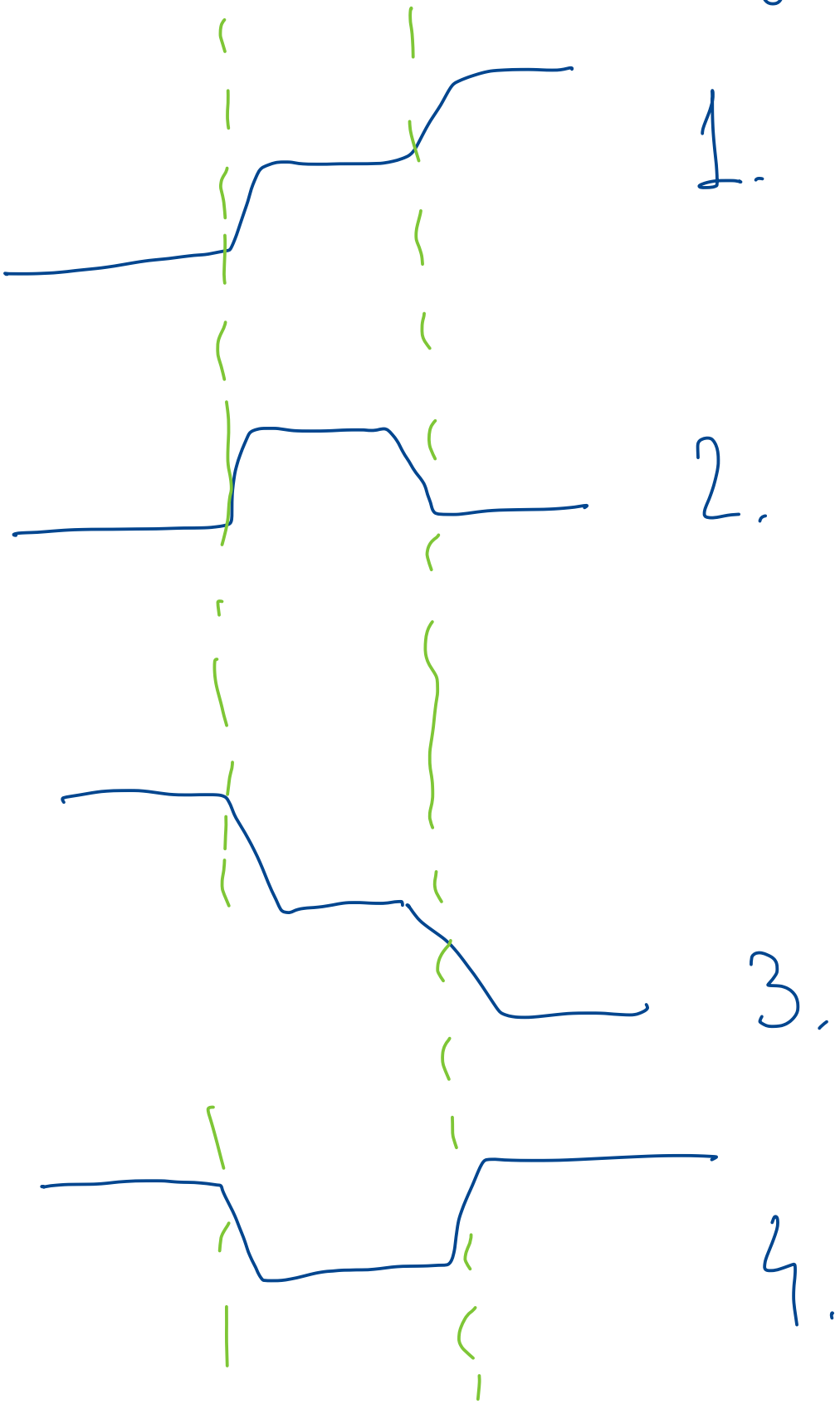
```
void doCallbacks() {
        if (pololu_isWhiteline()) {
                // RTPL callback
                do_OnWhiteline_callback();
        } else {
                // RTPL callback
                do_OnBlackline_callback();
        }
        // RTPL callback: sensors are inverse, i.e. when obstacle=0 then there IS one!
        if(pololu_isObstacleMid())
                do_OnSigMid_callback();
        if(pololu_isObstacleLeft())
                do_OnSigLeft_callback();
        if(pololu_isObstacleRight())
                do_OnSigRight_callback();
        if(pololu_isTac())
                // RTPL callback
                do_OnTac_callback();
}
```

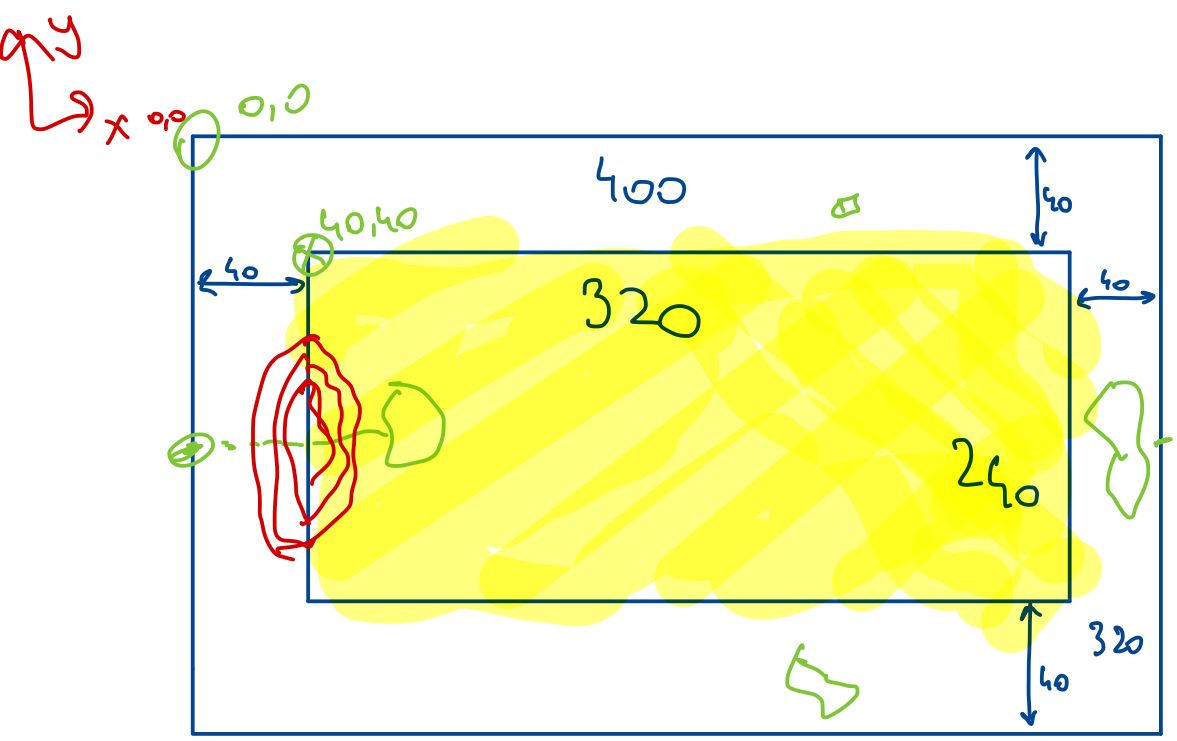
HINT: Make sure that your comments and code still fit on any screen you might use with different computers. If the code fits on a 25 inch monitor, it does not mean that it fits on the screen width of your laptop. Comments should be readable without scrolling to the left and to the right!

HINT: Eclipse provides an auto-format function for C-code. Make use of it! (Ctrl + Shift + f)


Spawn Areas
Flying Saucer
1
2
3
4

Flying Saucer possible trajectories



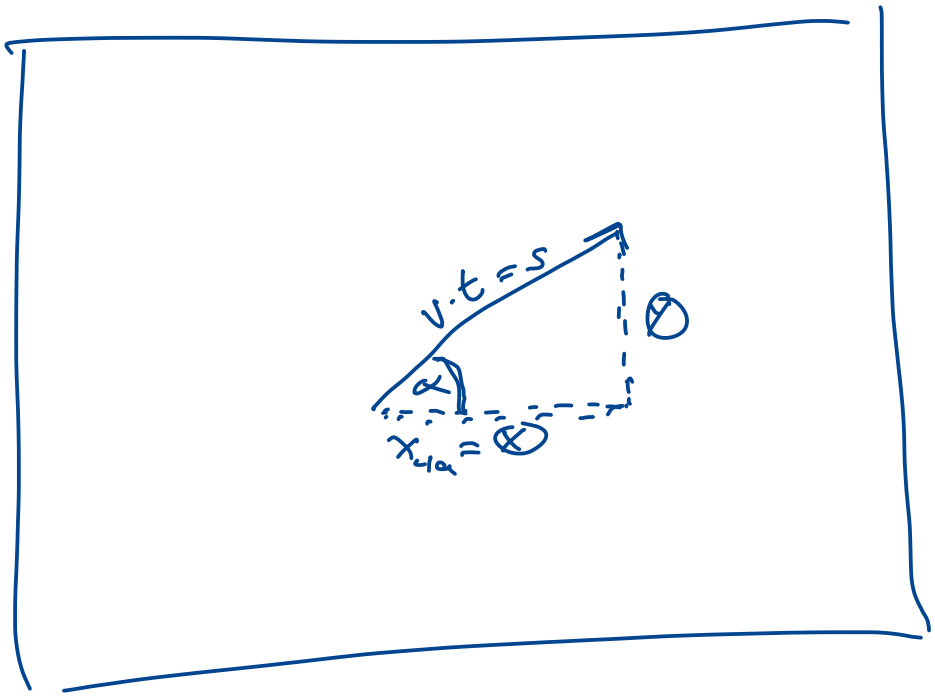


Absolute (x,y) { $0 \le x \le 400$; $0 \le y \le 320$

Screen (x,y) { $0 \le x \le 320$; $0 \le y \le 240$

= Abs - 40 // for both x and y

gets into gdispDraw



α: from joystick values

$x = x_0 + v \cdot t$

$v = v_0 + a \cdot t$

x_0, v_0: Current speeds

x_{new}, x_{old}

y_{new}, y_{old}

Tick: Compute $(x,y)_{new}$

draw $(x,y)_{old}$

Xt=

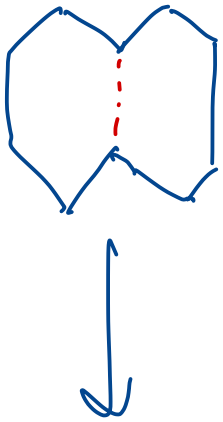
↳ updateTask

---

10 sides max

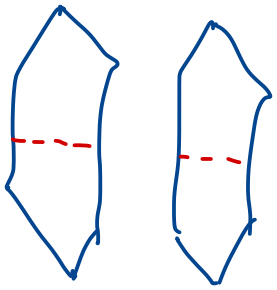
L

10



M

6



S

5