System Programming
Holiday Homework
-
8th Laboratory (14th .. 17th April 2020)

In this laboratory students will start working in the foundations and libraries necessary for implementing the project.
As a support for this exercises 2 different applications are provided:

- App1 randomly places lemons on the screen and allows the user to click on window to clear the leamon.
- App2 application prints various graphical elements on the screen, retrieves the mouse movement and reads the keyboard.

Exercise I

Compile each of the provided application using the gcc:

- `gcc app1.c -o app1 UI_library.c -lSDL2 -lSDL2_image -lpthread`
- `gcc app2.c -o app2 UI_library.c -lSDL2 -lSDL2_image -lpthread`

if the compiler gives the following error:

- …`bin/ld: cannot find -lSDL2`
- …`bin/ld: cannot find -lSDL2_image`

It will be necessary to install the SDL2 and SDL2_image library:

- ○ https://wiki.libsdl.org/Installation

Linux
In Linux (or WSL) it will be necessary to use the correct package manager:

- Ubuntu: `sudo apt-get install libsdl2-dev`
- Ubuntu: `apt-get install libsdl2-image-dev`
- OpenSUSE: `sudo zypper install SDL2-devel`
- OpenSUSE: `sudo zypper install SDL2_image-devel`

More information can be retrieved in the following pages:

- https://lazyfoo.net/tutorials/SDL/01_hello_SDL/linux/index.php
- https://lazyfoo.net/tutorials/SDL/06_extension_libraries_and_loading_other_image_formats/linux/index.php

MacOS X with XCode
In MacOS X you should follow the next tutorial:

- http://lazyfoo.net/tutorials/SDL/01_hello_SDL/mac/index.php
- http://lazyfoo.net/tutorials/SDL/06_extension_libraries_and_loading_other_image_formats/mac/index.php
-

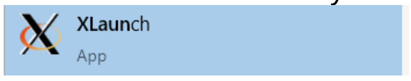
MacOS X with gcc in terminal
In MacOS X you should follow the next tutorial:

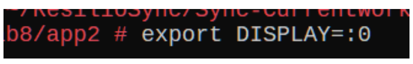
- http://lazyfoo.net/tutorials/SDL/01_hello_SDL/mac/index.php
- http://lazyfoo.net/tutorials/SDL/06_extension_libraries_and_loading_other_image_formats/mac/index.php
- https://www.youtube.com/watch?v=BCcvwoIOHVI

After installing all the libraries the compilation will produce the desired application.

Exercise II

Try to execute the compiled programs.
In WSL it will be necessary to run the X server before launching the applications:





```
b8/app2 # export DISPLAY=:0
```

App1 creates random lemons and when the user clicks in a aplace clears it.
App2 shows how to print lemons, cherries and bricks, along a few other interactions:

- A monster follows the mouse cursor
- When the user clicks on the mouse left button the color of the moster changes
- When the user presses the left arrow key a pacman apears on a random place

UI_library.c

In order to update the screen a set of auxiliary functions are available in the files `UI_library.c` and `UI_library.h` .
The list of functions available for the development of the project is the following:

- **`int create_board_window(int dim_x, int dim_y);`**
  - This function initializes the graphical environment and creates a window representin a board with dim_x by dim_y places.

- **`void get_board_place(int mouse_x, int mouse_y, int * board_x, int *board_y);`**
  - When a user clicks on the windows or moves the mouse the SDL functions return the cursor position in the window. This function transforms that pixel position in the window (mouse_x and mouse_y) into the coordinate of a place in the board (the arguments board_x and board_y are passed to the function as references).

- **`void close_board_windows();`**
  - This function closes the board windows before the games terminates

- **`void paint_pacman(int  board_x, int board_y , int r, int g, int b);`**
  - This function prints a pacman on the defined coordinates (board_x, board_y) with a certain color (r, g and b).

- **`void paint_monster(int  board_x, int board_y , int r, int g, int b);`**
  - This function prints a monster on the defined coordinates (board_x, board_y) with a certain color (r, g and b).

- **`void paint_lemon(int  board_x, int board_y );`**
  - This function prints a lemon on the defined coordinates (board_x, board_y).

- **`void paint_cherry(int  board_x, int board_y);`**
  - This function prints a cherry on the defined coordinates (board_x, board_y).

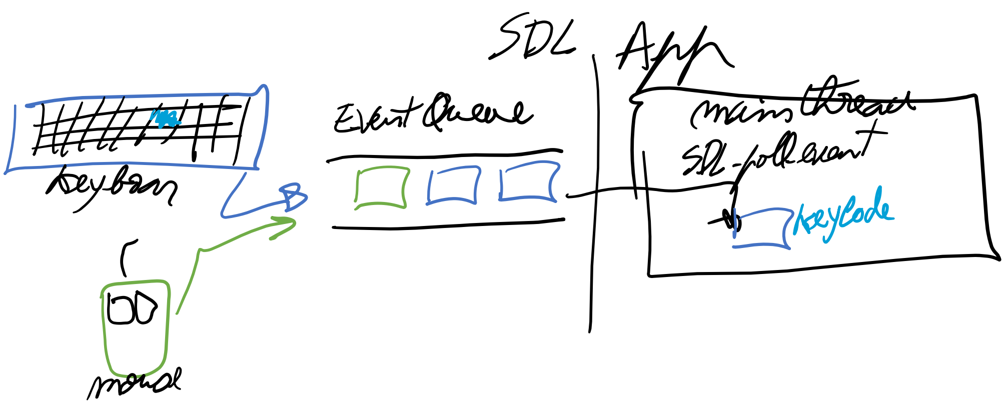
- **`void paint_brick(int  board_x, int board_y );`**
  - This function prints a brick on the defined coordinates (board_x, board_y).

- **`void clear_place(int  board_x, int board_y);`**
  - This function clears a place (paints it white).

These functions can only be called from the main thread!!!!.

SDL events

SDL based programs are built around a main loop that waits for events stored in the event queue. These events are generated by the various input devices (mouse, click, mouse movement, key strokes…) and contain relevant information.



The way to receive and handle such events is presented in the following example code:

```
while (!done){
  while (SDL_PollEvent(&event)) {
    if(event.type == SDL_QUIT){      event handler
      done = SDL_TRUE;
    }
    if(event.type == SDL_MOUSEBUTTONDOWN){
      int board_x, board_y;
      int window_x, window_y;
      window_x = event.button.x;
      window_y = event.button.y;                                          event handler
      printf("clicked on pixel x-%d y-%d\n", window_x, window_y);
      get_board_place(window_x, window_y, & board_x, &board_y);
      printf("clicked on board x-%d y-%d\n", board_x, board_y);    }
  }
  …
}
```

There are multiple types of events. Each type of these events should be handled in a part of the mail loop by its own event handler. This can be accomplished with a series of if's whose condition compare `event.type` with the various constants.
The most important types of events (and corresponding constants are) are:

- SDL_QUIT
- SDL_MOUSEBUTTONDOWN
- SDL_MOUSEMOTION
- SDL_KEYDOWN

Each one of this type of event includes relevant and different information that can be retrieved from inside the event variable.

SDL_QUIT
This event is created when the user presses `CTRL-C` in the windows or click the close button on the windows bar.

SDL_KEYDOWN
This event is generated whenever the user presses one of the keyboard keys.
The identity of the pressed key (Keycode) can be retrieved from the field `event.key.keysym.sym`.
The list of possible **SDL_Keycode** values is available in:

- https://wiki.libsdl.org/SDL_Keycode

SDL_MOUSEMOTION
This event is generated whenever the user moves the mouse (on top of the SDL window).
The location of the pixel on which the mouse is can be retrieved from the fields event.motion.x and event.motion.y .

SDL_MOUSEBUTTONDOWN
This event is generated whenever the user clicks with the mouse on the SDL window).
The location of the pixel on which the mouse is can be retrieved from the fields `event.button.x` and `event.button.y` . It is possible to know what button was pressed accessing the field `event.button.button .`

Window board coordinates
The events `SDL_MOUSEMOTION` and `SDL_MOUSEBUTTONDOWN` provide to the programmer the co-ordinates (inside the window) where the mouse is (`event.button.x/y` or `event.button.x/y`).
In order to convert these coordinates to the game coordinates (corresponding place in the board the function `get_board_place` should be used.

Information about events:

- https://wiki.libsdl.org/SDL_Event
- https://wiki.libsdl.org/SDL_PollEvent
- https://wiki.libsdl.org/SDL_KeyboardEvent
- https://wiki.libsdl.org/SDL_MouseMotionEvent
- https://wiki.libsdl.org/SDL_MouseButtonEvent
- http://lazyfoo.net/tutorials/SDL/03_event_driven_programming/index.php

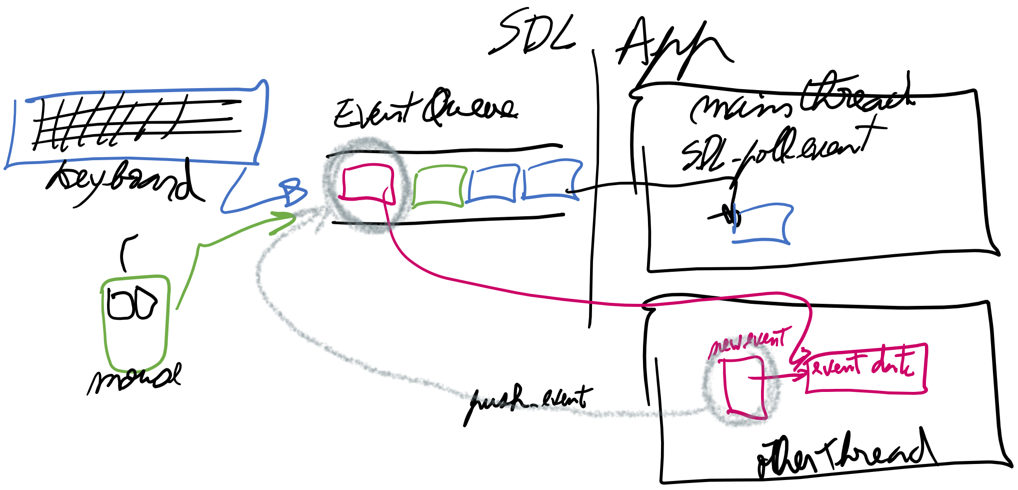
Only the main thread can retrieve events from the queue !!!!.

Communication with main thread

Since only the main thread can write/draw on the graphical window but sometimes other threads need to update the information of such window it is necessary to perform communication between other threads and the main thread.
This communication can be done using the SDL event queue.
To accomplish this communication a few steps should be followed:

- A new event type should be defined
- Other threads create such new events
- Other threads push these events into que queue
- Main thread retrieves the new event
- Main thread processes the information transmitted with the event.

App1 is multithreaded SDL application where there is a thread (`thread_lemon`) that continuously decides the position for a new lemon and pushes a new events into the event queue with such information. The various steps are explained next:



Definition of new event type

To create a new event type it is necessary to define:

- a data structure that will store the information to transmit to the main thread
- and a new event identifier.

In app1 the data structure that will contains the information to be transmitted is defined as:

```
typedef struct Event_ShowLemon_Data{
  int x, y;
} Event_ShowLemon_Data;
```

It is also necessary to assign a new identifier (equivalent to the `SDL_MOUSEBUTTONDOWN` or `SDL_QUIT`) that will be used in the main thread to distinguish this new event type.
The function that assigns this new identifier is `SDL_RegisterEvents(1)`, in this example we assign the return of this function to the global variable `Event_ShowLemon`:

```
    Event_ShowLemon =  SDL_RegisterEvents(1);
```

After this assignment all the threads know the new event type identifier.

Creation of new event

After the definition of the event type it is possible to create new events of this type. This is accomplished with the following code:

```
event_data = malloc(sizeof(Event_ShowLemon_Data));
event_data->x = x;
event_data->y = y;
```

```
    SDL_zero(new_event); // reset event structure
    new_event.type = Event_ShowLemon; //set event type
    new_event.user.data1 = event_data; // set event data
```

The first three lines create and initialize the structure of type `Event_ShowLemon_Data` that will contain the new event actual data.
The other lines initialize the new event structure with its type

```
        new_event.type = Event_ShowLemon)
```

and with the data to be transmitted

        new_event.user.data1 = event_data.

The event variable can contain other information, namely a integer code or another void * (data2).

Pushing the event into queue
After the initialization of the new event structure and the data to be transmitted, the thread can now push the new event into the queue:

```
        SDL_PushEvent(&new_event);
```

After the completion of this function, the main thread will be able to retrieve it.

Retrieving the new event
This new event will be available to the main thread as any regular event.
The `SDL_PollEvent` function will return and the main thread should now verify this new event type:

```
        if(event.type == Event_ShowLemon){
```

Processing the information
Since the other thread sent a pointer to a `Event_ShowLemon_Data` on the `event.user.data1` field, it is possible for the main thread to retrieve the pointer and the pointed data:

```
        Event_ShowLemon_Data * data = event.user.data1;
        data->x ... data-> y
```

The overall code for retrieving this new event type and its information is presented here:

```
        while (!done){
          while (SDL_PollEvent(&event)) {
            …
            if(event.type == Event_ShowLemon){
              // we get the data (created with the malloc)
              Event_ShowLemon_Data * data = event.user.data1;
              // retrieve the x and y
              int x = data->x;
              int y = data->y;
              // we paint a lemon
              paint_lemon(data->x, data->y);
              free(data);
            }
            ...
          }
        }
```

After processing the event it is possible to free the event data that was allocated in the `lemon_thread`.

Information about user events:
https://wiki.libsdl.org/SDL_PushEvent
https://wiki.libsdl.org/SDL_RegisterEvents
https://wiki.libsdl.org/SDL_UserEvent

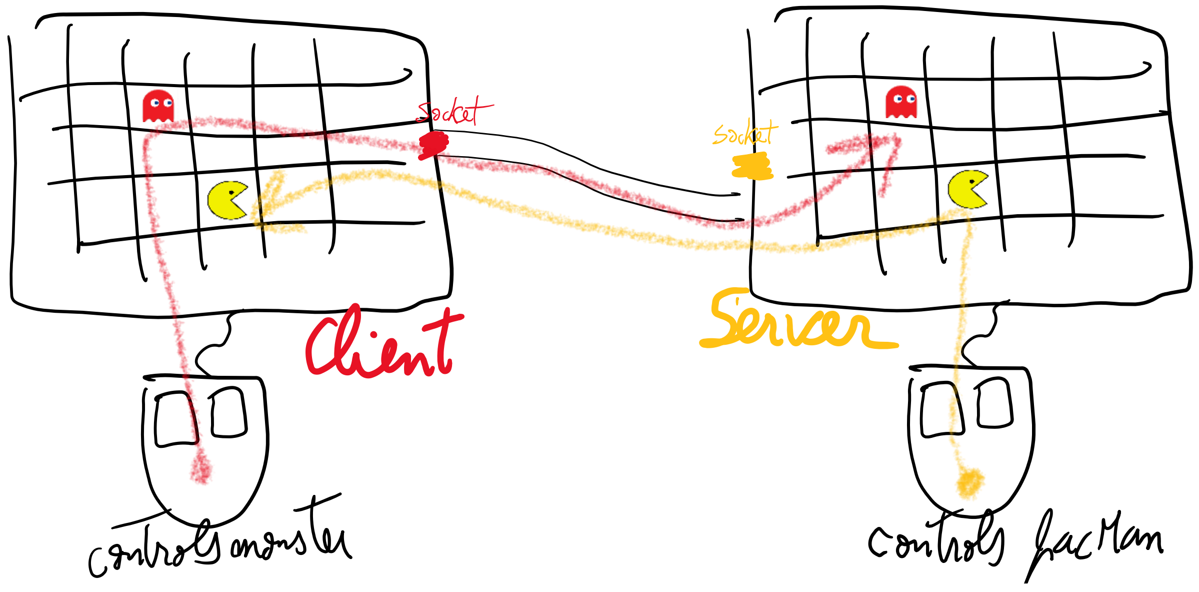
Exercise III

Modify **app1** to that implement the following behavior

- A new thread will start generating cherries in random locations
  - This first new functionality will require a new thread, the change of the event that communicates new cherries/lemons and the handling of this event in the main thread.
- Clicking on a place will put a monster in it
  - The second functionality will require the event handler for mouse clicks to print a monster.
- Moving the mouse will clear the case
  - This functionality will require a new handler for the mouse movement event. Students can use the app2 code to implement this functionality.

Exercise IV

In this exercise student will implement one application that will allow two users (one monster and one pacman) to interact with each other.
One user will control the pacman while the other user will control the monster.
The applications are connected using a INET stream socket and use that socket to transmit the new position of the pacman/monster to the other program.



Student should write one single application that can run as a server (that creates a socket, accepts connections and controls the pacman) or a client (that connects to the server and controls the monster).
If the application receives as arguments the address (xxx.xxx.xxx.xxx) and a port number it becomes a client and will try to connect to such server. If the application is started without any argument it will act as a server.
The user on the server controls the yellow pacman, the user on the client controls a red monster.
The application should implement the following architecture:

- The main thread will be responsible for:
  - Receiving the mouse events
  - Transmit the new position to the other application
  - Print the monster and packman
- Other thread should be responsible for receiving in the socket the updates sent by the other application.

Student should decide the structure of the messages exchanged between the client and the server.