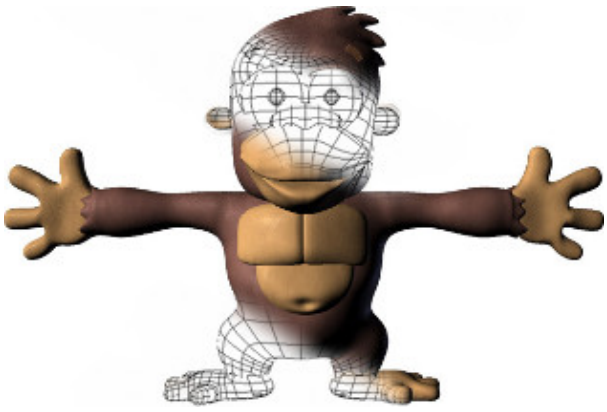
B1 - C Graphical Programming

B-MUL-100

Bootstrap My Hunter

Introduction to events and animations



1.0

PREAMBULE

In order to be able to do that bootstrap correctly you should have entirely finished the first initiation bootstrap.
The exercices asked here assume that you already have a simple window opened which stays like that. Doing the first bootstrap will allow you to have a better comprehension of the following tasks.

EVENTS

The most important thing when you develop a game is the input. Being able to get interactions from the players to the window allows you to make them feel their influence on the course of your game. Basically being able to interact with what is happening is what makes the difference between a video game and cartoon or a movie.

In programming these interactions are called **events**.

With the CSFML we can detect such interactions thanks to the function named `sfRenderWindow_pollEvent`.

Try to find sfRenderWindow_pollEvent in the documentation !

To analyse these events make a function that will be called whenever an events occurs which follows this prototype:

```
void analyse_events(sfRenderWindow *window, sfEvent event);
```

This function will call the appropriate function according to the detected event.

Then make a function `manage_mouse_click` which follows this prototype:

```
void manage_mouse_click(sfMouseButtonEvent event)
```

This function will have to be called whenever a mouse click is detected by `analyse_event` and it will print the position x and y of the mouse cursor the following way:

```
Terminal
~/B-MUL-100> ./my_hunter
Mouse x=45 y=108
Mouse x=34 y=209
Mouse x=42 y=42
Mouse x=21 y=24
Mouse x=19 y=467
```

Now lets make a function name `close_window` which follows this prototype:

```
void close_window(sfRenderWindow *window);
```

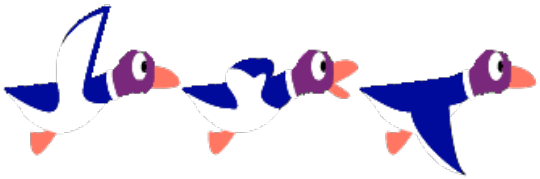
The name being pretty obvious, this function will close the window whenever the event of type `sfEvtClosed` is detected.

ANIMATIONS

Another important thing when you make a game is having animations or at least moving objects.

Weither it is just pixels particles animated by physic calculations or animated sprite made thanks to sprite sheets, animations provide a new dimension to your games.

In this part we will see how to create animation in your program thanks to a 2D sprite sheet. For this use the sprite sheet name spritesheet.png provided on the intranet.



As you can see this sprite sheet is composed of three different parts, each part having the same size wich is 110 pixels by 110 pixels, it makes the sprite sheet being 330 pixels long.

If you try to display the sprite as is, you will find out that the 3 parts appears on your screen. In order to fix that and diplay only one part at a time we will have to use two things from the CSFML library, the `sfIntRect` structure and the function `sfSprite_setTextureRect`.

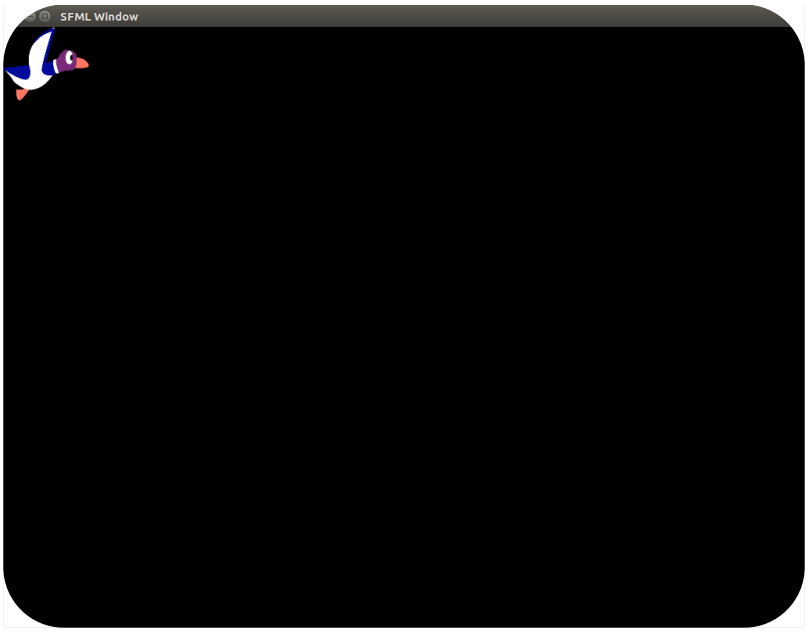
First we need to define a rectagle to move in our sprite sheet, create a variable of type `sfIntRect` the following way:

```
sfIntRect rect;

rect.top = 0;
rect.left = 0;
rect.width = 110;
rect.height = 110;
```

The `top` field defines the vertical position of the rectangle here we put it at 0 cause we want it to be at the top of our sprite sheet, then the `left` field is the horizontal position of the rectangle we also put it at 0 because we what it to be at the left-most position of our sprite sheet.
We set both the `width` and `height` fields because as we said earlier each part of our sprite sheet is 110 pixels by 110 pixels.

Now just before displaying you have to call the function `sfSprite_setTextureRect` with the `sfIntRect` in parameter.
You should see something like this:



Now create a function named `move_rect` which follows this prototype:

```
void move_rect(sfIntRect *rect, int offset, int max_value);
```

This function will be called before setting the texture rectangle, it will change the left position in the rectangle and if that position is higher than or equal to the width of your sprite sheet.
With just that you should obtain an interesting result with your duck flying fast. It is changing according to the frame rate of your program, the higher it is the faster the animation is done.

To avoid that problem there is something called clocks, it will allow you to manage your animations based on time and not on the frame rate of your program. In CSFML you can do that thanks to the data structures `sfClock` and `sfTime`, the first one will allow you to get the time elapsed since the clock started and the second one will just store that time value.

The functions `sfClock_restart` and `sfClock_getElapsedTime`

This is an example of how to use the clocks and time in CSFML

```
sfClock *clock;
sfTime time;
float seconds;

clock = sfClock_create();
while (1)
{
    time = sfClock_getElapsedTime(clock);
    seconds = time.microseconds / 1000000.0;
```

```
    if (seconds > 1.0)
    {
        my_putstr("One more second elapsed...");
        sfClock_restart(clock);
    }
}
```

One second is equal to 1000000 microseconds

From there instead of printing a message every second try to call your `move_rect` function, and you should see your duck flying nicely every second.