ASTEROID INVASION

Proyecto de la 2.ª evaluación

Ander Aguinaga San Sebastián

MAIS 1.º A, en U-Tad

Proyectos

10/06/2021

Contenido

Introducción

En este proyecto de temática libre, he decidido crear un pequeño videojuego, inspirado en Debris Infinity. Existen dos mecánicas principales para la nave del jugador: movimiento y disparo, cada uno controlado por un *joystick* distinto.

El objetivo del usuario es destruir la mayor cantidad asteroides como le sea posible antes de acabar con todas sus vidas, para conseguir así una mejor puntuación final. Para tener conocimiento de la cantidad de vidas que tiene disponibles, el jugador podrá mirar al controlador, donde hay instalado un *display* de siete segmentos que mostrará esta información en todo momento.

Para la realización de este proyecto, se ha utilizado la librería SDL 2.0.5, con las extensiones SDL_image 2.0.5 (para poder importar *sprites*) y SDL_ttf 2.0.15 (para poder importar fuentes con formato *TTF*).

También se ha hecho uso de la librería SerialPort desarrollada por Manash Kumar Mandal (https://github.com/manashmandal/SerialPort).

El proyecto se ha compilado utilizando el compilador MSCV 14.28.29910 y /*std:c++latest*, para poder utilizar las adiciones que trae C++20.

Componentes

Microcontrolador

Se ha utilizado una placa Elegoo Uno R3, programada mediante el lenguaje de programación Arduino con uso de la IDE de Arduino.

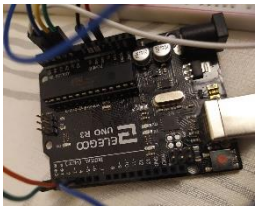


Ilustración 1: Microcontrolador.

Protoboard

El *display* de siete segmentos, así como el *shift register* que lo controla, están conectados a una Breadboard MB-102 de Stellarsource Electronic.

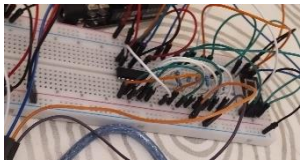
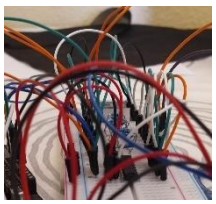


Ilustración 2: Componentes conectados a una protoboard.

Cables

Se encargan de transportar la energía eléctrica.



Resistencias

Componentes diseñados para limitar la corriente que circula por un circuito eléctrico.

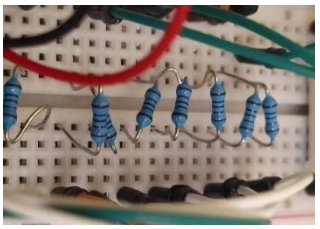


Ilustración 4: Resistencias.

Shift register

Ilustración 5: Shift register (74HC595N).

En mi caso, he utilizado el 74HC595N concretamente. Nos sirve para reducir drásticamente el número de pines digitales que utilizamos en el microcontrolador (solo utilizamos 3 en total, siendo que el *display* utiliza 7+1 pines.

Display de siete segmentos

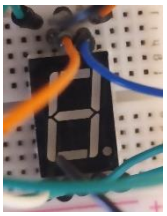


Ilustración 6: Display *de siete segmentos.*

Dispositivo electrónico capaz de reproducir, generalmente, valores numéricos, de una forma relativamente simple. Tiene siete segmentos para el dígito en sí, mientras que abajo a la derecha tiene un puntito de más, lo que hace que, en total, utilice hasta ocho pines. Este dispositivo es utilizado para mostrar las vidas que quedan restantes al jugador.

Joysticks

Ilustración 7: Joysticks.

Dispositivos de entrada analógica que nos permite obtener la dirección hacia la que el *joystick* apunta. Uno de los *joysticks* es utilizado para el movimiento del jugador; el otro, para la dirección hacia la que dispara el mismo.

Código

arduino.ino

En este archivo se encuentra todo el código que se instala en el microcontrolador de Arduino. Su funcionalidad es simplemente recibir datos de las entradas analógicas (los *joysticks*), y enviárselos al programa escrito en C++ gracias a la comunicación serie. Por otra parte, también recibe datos desde el programa en C++ (la vida del jugador, más concretamente), y la utiliza para imprimirla en el *display* de siete segmentos.

```
int clockPin = 4;
int latchPin = 3;
int dataPin = 2;

int numbers[10] = //values necessary for the 7-segment display to show the correct number
{
  0b00111111, //0
  0b00000110, //1
  0b01011011, //2
  0b01001111, //3
  0b01100110, //4
  0b01101101, //5
  0b01111101, //6
  0b00100111, //7
  0b01111111, //8
  0b01100111  //9
};

void setup() {
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);

  Serial.begin(9600);
}

void loop()
{
  char incomingData[2];

  incomingData[0] = Serial.read();
  if(incomingData[0] == 'C') //the first byte of data being 'C' means this information was sent from C++
  {
    incomingData[1] = Serial.read(); //reads the number of lives the player has left

    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, MSBFIRST, numbers[incomingData[1]]);
    digitalWrite(latchPin, HIGH);
  }

  char data[5];
  data[0] = 'A';

  int moveX = analogRead(A0);
  int moveY = analogRead(A1);
  data[1] = map(moveX, 0, 1023, 0, 255);
  data[2] = map(moveY, 0, 1023, 0, 255);

  int shootX = analogRead(A2);
  int shootY = analogRead(A3);
  data[3] = map(shootX, 0, 1023, 0, 255);
  data[4] = map(shootY, 0, 1023, 0, 255);
  Serial.write(data, 5); //reads all the values from analog input pins, and sends them to the Serial buffer.

  delay(450);
}
```

main.cpp

El archivo donde se encuentra la función *main*, el punto de comienzo del programa.

Mediante la librería *thread* de C++, se crean dos hilos: uno se encargará de la comunicación con el microcontrolador de Arduino: el otro se ocupará del bucle principal del juego. Resulta de ayuda el tener varios hilos en este caso, puesto que el bucle de la comunicación con Arduino itera mucho más despacio que el bucle principal del juego.

La función que controla la conexión serie utiliza la librería SerialPort. Funciona de forma similar al archivo .ino visto anteriormente; recibe datos del Arduino y los inserta en los objetos del juego; por otra parte, también recoge datos de estos objetos (la vida del jugador), y se los manda al Arduino.

En este último, también se controla la aparición de asteroides por la pantalla; y es que los asteroides aparecen cada 20 *frames* del juego.

```cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <algorithm>

#include "RNG.hpp"

#include "SDL.h"
#include "SerialPort.hpp"
#include "Game.hpp"

#undef main

int convert(char input) //convert from unsigned char to signed int
{
    int output;
    if (input < 0)
    {
        output = input + 256;
    }
    else
    {
        output = input;
    }
    output -= 127;
    if (output >= -3 && output <= 3) //as joysticks are not flawless, it's common for them to have an error of 1-2 in their values.
                                     //As a result, the player would be moving even when not touching the joystick. To avoid this, values near 0
                                     //are rounded to 0.
    {
        output = 0;
    }
    return output;
}

//This function will communicate with Arduino. It'll be responsible for both sending to and receiving data from it.
void serial_connection(Game& game)
{
    std::string port("\\\\.\\COM3");

    SerialPort* arduino = new SerialPort(port.c_str());

    while (!arduino->isConnected()) { //keeps trying until manages to connect with Arduino.
        delete arduino;
        Sleep(200);
        arduino = new SerialPort(port.c_str());
    }

    while (arduino->isConnected() && game.running)
    {
        char incomingData[5];
```

```
        int readResult = arduino->readSerialPort(incomingData, 5); //reads 5 bytes from the Serial buffer

        if (incomingData[0] == 'A') //the first byte of data being 'A' means this information was sent from Arduino
        {
            game.player->set_direction(convert(incomingData[1]), convert(incomingData[2])); //updates player's movement direction

            int shootX = convert(incomingData[3]);
            int shootY = convert(incomingData[4]);
            game.player->set_shoot(shootX, shootY); //updates player's shooting direction
            if (shootX || shootY)
            {
                game.generate_bullet(shootX, shootY); //if the shooting direction's joystic is actually moved, a bullet will be generated
            }
        }

        char numberData[2];
        numberData[0] = 'C';
        numberData[1] = char(game.player->get_health());
        arduino->writeSerialPort(numberData, 2); //sends player's health to Arduino

        Sleep(450);
    }
    game.running = false;
    delete arduino;
}

void play(Game& game)
{
    game.init();
    int asteroidClock = 0;
    while (game.running)
    {
        if (++asteroidClock == 20)
        {
            game.generate_asteroid(); //every 20 frames, a new asteroid spawns
            asteroidClock = 0;
        }
        game.update();
        game.handle_events();
        game.render();
        Sleep(50);
    }
    std::cout << "Score: " << game.score; //when the game ends, the final score will be shown in the console
    game.clean();
}

int main()
{
    Game game;

    //multi-threading, so as for the Serial connection and the actual videogame to run independently.
    std::thread connectionThread(serial_connection, std::ref(game));
    std::thread gameThread(play, std::ref(game));

    connectionThread.join();
    gameThread.join();

    return 0;
}
```

Game.hpp & Game.cpp

Esta clase contiene las funciones a las que se llama desde el bucle principal de main.cpp:

- *init*: sirve para inicializar el programa en sí: crear la ventana, crear el renderizador, crear el objeto del jugador, etc.
- *handle_events* nos permite, como su nombre indica, controlar algunos eventos. En este caso, al estar utilizando el controlador de Arduino, no lo utilicé tanto (en otras ocasiones, se suele usar para la lectura de pulsaciones de teclado, etc.). De todas formas, sí le damos un uso: poder cerrar la ventana mediante el botón de cierre que ofrece Windows.
- *update*: desde aquí se hacen diversas cosas. Por una parte, se llama al *update* del jugador, de los asteroides y de las balas (lo que, entre otras cosas, hace que tengan movimiento). Por otra parte, se hace un control de colisiones con los asteroides.

Asimismo, elimina del vector correspondiente los asteroides y las balas inactivas. Por último, se termina el juego si el jugador tiene vidas negativas.

- Lo que hace la función *render* es renderizar las imágenes; hacerlas visibles en la ventana. Renderizamos primero el fondo; después el jugador, los asteroides y las balas; y, por último, la puntuación.
- *clean* es una función que permite liberar parte de la memoria alocada dinámicamente durante el transcurso del juego.
- *generate_asteroid* y *generate_bullet*, respectivamente, crean un asteroide y una bala.

```
1   #pragma once
2
3   #include <iostream>
4   #include "SDL.h"
5
6   #include <vector>
7   #include "Asteroid.hpp"
8   #include "Bullet.hpp"
9   #include "Player.hpp"
10
11  class Game
12  {
13  public:
14      void init();
15      void update();
16      void handle_events();
17      void render();
18      void clean();
19      void generate_asteroid();
20      void generate_bullet(int inputX, int inputY);
21
22      static inline SDL_Window* window;
23      static inline SDL_Renderer* renderer;
24
25      SDL_Texture* texture;
26
27      std::vector<std::unique_ptr<Asteroid>> asteroids;
28      std::vector<std::unique_ptr<Bullet>> bullets;
29
30      bool running = true;
31      Player* player;
32      int score;
33  };
34
35
```

```cpp
#include "Game.hpp"
#include "SDL.h"
#include "SDL_ttf.h"
#include <sstream>

void Game::init()
{
    if (!SDL_Init(SDL_INIT_EVERYTHING))
    {
        window = SDL_CreateWindow("AsteroidInvasion", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 1080, 720, 0); //initialise window
        renderer = SDL_CreateRenderer(window, -1, 0); //initialise renderer
    }

    TTF_Init(); //initialise fonts' extension

    player = new Player(renderer);
    score = 0;

    SDL_Surface* surface = IMG_Load("media/background.png");
    texture = SDL_CreateTextureFromSurface(renderer, surface); //make a texture for the background
    SDL_FreeSurface(surface);
}

void Game::handle_events()
{
    SDL_Event event;
    SDL_PollEvent(&event);

    switch (event.type)
    {
    case SDL_QUIT: //window's X button (top righthand corner)
        running = false;
    }
}

void Game::update()
{
    player->update(); //update player

    for (auto& a : asteroids) //update all asteroids
    {
        a->update();
    }
    for (auto& b : bullets) //update all bullets
    {
        b->update();
    }

    for (auto& a : asteroids)
    {
        for (auto& b : bullets)
        {
            if (b->collide(a->get_dest())) //check for collision between asteroids and bullets
            {
                --a->health; //if a collision happened, asteroid's health is decreased
                if ((++score) % 20 == 0 && player->get_health() < 9)
                {
                    player->heal_up(); //if the shot was successful, the score is increased by 1
                }
            }
        }
        if (player->collide(a->get_dest())) //check for collision between asteroids and the player
        {
            a->active = false; //if a collision happened, the asteroid becomes inactive
        }
    }

    //if an asteroid or a bullet is not active, delete theme and erase them from the vector

    asteroids.erase(std::remove_if(asteroids.begin(), asteroids.end(), [](const std::unique_ptr<Asteroid>& a) {
            return !a->active;
        }),
        asteroids.end());
    bullets.erase(std::remove_if(bullets.begin(), bullets.end(), [](const std::unique_ptr<Bullet>& b) {
            return !b->active;
        }),
        bullets.end());

    if (player->get_health() < 0)
    {
        running = false; //if the player's health is bellow 0, the game finishes
    }
}
```

```cpp
void Game::render()
{
    SDL_RenderCopy(renderer, texture, NULL, NULL); //print the background

    //draw player, asteroids and bullets
    player->draw();
    for (auto& a : asteroids)
    {
        a->draw();
    }
    for (auto& b : bullets)
    {
        b->draw();
    }

    //make texture for the text showing the player's score

    std::stringstream ss;

    ss << "Score: " << score;

    TTF_Font* arial = TTF_OpenFont("arial.ttf", 20);
    SDL_Color white = { 255, 255, 255 };

    SDL_Surface* scoreSurface = TTF_RenderText_Solid(arial, ss.str().c_str(), white);

    // now you can convert it into a texture
    SDL_Texture* scoreText = SDL_CreateTextureFromSurface(renderer, scoreSurface);

    SDL_Rect scoreDestination = { 950, 20, 80, 25 };
    SDL_RenderCopy(renderer, scoreText, NULL, &scoreDestination);

    SDL_DestroyTexture(scoreText);
    SDL_FreeSurface(scoreSurface);

    //render everything
    SDL_RenderPresent(renderer);
}

void Game::clean()
{
    delete player;
    TTF_Quit();
    SDL_DestroyTexture(Asteroid::texture);
    SDL_DestroyTexture(Bullet::texture);
    SDL_DestroyWindow(window);
    SDL_DestroyRenderer(renderer);
    SDL_Quit();
}

void Game::generate_asteroid()
{
    asteroids.emplace_back(std::make_unique<Asteroid>(renderer, player));
}

void Game::generate_bullet(int inputX, int inputY)
{
    bullets.emplace_back(std::make_unique<Bullet>(renderer, player, inputX, inputY));
}
```

Player.hpp & Player.cpp

La clase Player hace referencia a la nave que el jugador controla. Como se puede observar, la mayoría de funciones miembros que tiene no son más que simples *getters* y *setters*. De todas formas, sí que hay un par de cosas que detallar en otras funciones. En

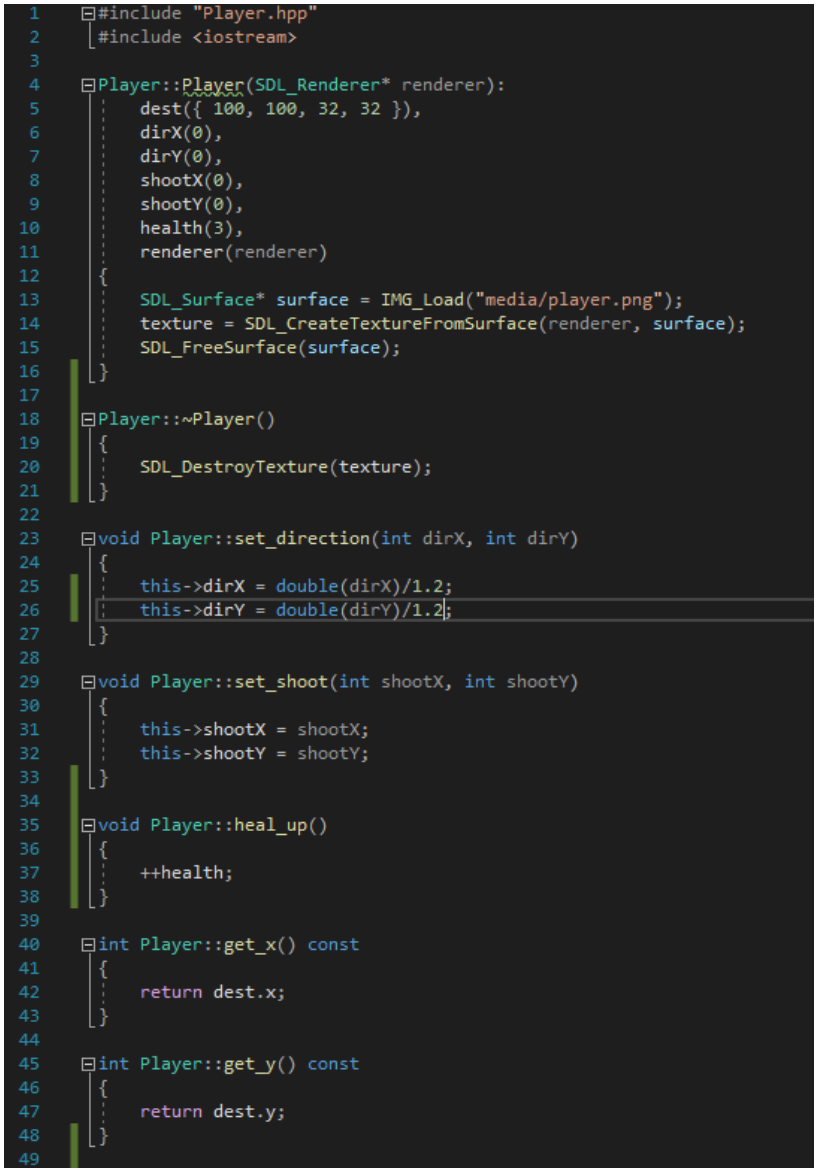
update, por ejemplo, se calcula en cada *frame* el ángulo de giro que hay que aplicar al *Sprite* a la hora de dibujarlo en pantalla. Esto es porque, dependiendo de la dirección a la que se mueva la nave, tus *Sprite* irá rotando. Se obtiene de la fórmula

$$tan\theta = \frac{v_y}{v_x}$$

Obviamente, el caso donde *vx = 0* es distinto (estudié el límite, y simplemente lo puse como un *if* aparte en el código.

La función *collide* comprueba si la nave está colisionando con algún otro asteroide. Esto lo comprueba mediante el algoritmo AABB, que resulta muy práctico para casos como este, donde las *hitbox* son rectangulares.

```
1   #pragma once
2
3   #include "SDL.h"
4   #include "SDL_image.h"
5
6   class Player
7   {
8   public:
9       Player(SDL_Renderer* renderer);
10      ~Player();
11      void set_direction(int dirX, int dirY);
12      void set_shoot(int shootX, int shootY);
13
14      void heal_up();
15
16      bool collide(const SDL_Rect& asteroid);
17
18      int get_x() const;
19      int get_y() const;
20      int get_health() const;
21
22      void update();
23      void draw();
24  private:
25      SDL_Texture* texture;
26      SDL_Renderer* renderer;
27
28      const SDL_Rect src{ 0, 0, 16, 16 };
29      SDL_Rect dest;
30      int dirX;
31      int dirY;
32      int shootX;
33      int shootY;
34      double angle;
35      int health;
36  };
37
```



```cpp
#include "Player.hpp"
#include <iostream>

Player::Player(SDL_Renderer* renderer):
    dest({ 100, 100, 32, 32 }),
    dirX(0),
    dirY(0),
    shootX(0),
    shootY(0),
    health(3),
    renderer(renderer)
{
    SDL_Surface* surface = IMG_Load("media/player.png");
    texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_FreeSurface(surface);
}

Player::~Player()
{
    SDL_DestroyTexture(texture);
}

void Player::set_direction(int dirX, int dirY)
{
    this->dirX = double(dirX)/1.2;
    this->dirY = double(dirY)/1.2;
}

void Player::set_shoot(int shootX, int shootY)
{
    this->shootX = shootX;
    this->shootY = shootY;
}

void Player::heal_up()
{
    ++health;
}

int Player::get_x() const
{
    return dest.x;
}

int Player::get_y() const
{
    return dest.y;
}

```

```cpp
int Player::get_health() const
{
    return health;
}

void Player::update()
{
    dest.x += dirX / 10;
    dest.y += dirY / 10;

    if (dirX == 0)
    {
        if (dirY > 0)
        {
            angle = 90;
        }
        else
        {
            angle = 270;
        }
    }
    else if (dirX > 0)
    {
        angle = atan(double(dirY) / double(dirX)) * 180 / M_PI;
    }
    else
    {
        angle = 180 + atan(double(dirY) / double(dirX)) * 180 / M_PI;
    }
}

bool Player::collide(const SDL_Rect& asteroid)
{
    if (dest.x < asteroid.x + asteroid.w &&
        dest.x + dest.w > asteroid.x &&
        dest.y < asteroid.y + asteroid.h &&
        dest.y + dest.h > asteroid.y)
    {
        --health;
        return true;
    }

    return false;
}

void Player::draw()
{
    SDL_RenderCopyEx(renderer, texture, &src, &dest, angle, NULL, SDL_FLIP_NONE);
}
```

Asteroid.hpp & Asteroid.cpp

Esta es la clase que define los objetos asteroides.

La idea era que los asteroides pudieran aparecer en cualquier parte de borde de la ventana de forma aleatoria. Lo que hice fue separar los bordes en cuatro partes: la zona de arriba, la de abajo, y la de cada lateral. Con un generador de números pseudoaleatorios, que después explicaré con mayor detalle, se decide en cuál zona va a aparecer el asteroide. Después, según la zona que haya salido, se generan de forma pseudoaleatoria las coordenadas del asteroide.

Los asteroides se mueven de forma lineal. La dirección será hacia la posición de la nave del jugador en el momento en que el asteroide aparece.

Además, para hacerlo un poco más visualmente dinámico, los asteroides también giran: al aparecer, se escoge de forma aleatoria el ángulo en que empiezan; y, en cada *frame*, girarán dos grados.

```hpp
#pragma once
#include "SDL.h"
#include "Player.hpp"

class Asteroid
{
public:
    Asteroid(SDL_Renderer* renderer, Player* player);
    void update();
    void draw();

    SDL_Rect get_dest();

    bool active = true;
    int health;

    static inline SDL_Texture* texture;
private:
    SDL_Renderer* renderer;

    const SDL_Rect src{ 0, 0, 32, 32 };
    SDL_Rect dest;

    double angle;
    int dirX;
    int dirY;
};
```

```cpp
#include "Asteroid.hpp"
#include "RNG.hpp"
#include <cmath>

Asteroid::Asteroid(SDL_Renderer* renderer, Player* player):
    health(2),
    renderer(renderer)
{
    dest.w = dest.h = 64;

    //zones 0 and 1 are, respectively, top and bottom borders of the window.
    //zones 2 and 3 are, respectively, left and right borders of the window.

    int zone = RNG::generate<0, 3>(); //randomly choose a zone to spawn in
    switch (zone)
    {
    case 0:
        dest.x = RNG::generate<0, 1080>();
        dest.y = RNG::generate<0, 20>();
        break;
    case 1:
        dest.x = RNG::generate<0, 1080>();
        dest.y = RNG::generate<700, 720>();
        break;
    case 2:
        dest.x = RNG::generate<0, 20>();
        dest.y = RNG::generate<0, 720>();
        break;
    case 3:
        dest.x = RNG::generate<1060, 1080>();
        dest.y = RNG::generate<0, 720>();
        break;
    }

    //the direction of the asteroid will be towards the player's position at the time the asteroid spawns.
    //it's necessary to normalise the vector, so that all asteroids have the same speed.
    int tempX = player->get_x() - dest.x;
    int tempY = player->get_y() - dest.y;

    int modulus = sqrt(tempX * tempX + tempY * tempY);

    dirX = tempX * 75 / modulus;
    dirY = tempY * 75 / modulus;

    angle = RNG::generate<0, 359>();

    SDL_Surface* surface = IMG_Load("media/asteroid.png");
    texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_FreeSurface(surface);
}
```

```cpp
void Asteroid::update()
{
    dest.x += dirX / 10;
    dest.y += dirY / 10;

    if (dest.x + 64 < 0 || dest.x - 64 > 1080 ||
        dest.y + 64 < 0 || dest.y - 64 > 720 ||
        health <= 0)
    {
        active = false; //disable asteroid if it gets out of bounds or has no health left
    }
    else
    {
        active = true;
    }

    angle += 2; //slightly rotate it
}

void Asteroid::draw()
{
    SDL_RenderCopyEx(renderer, texture, &src, &dest, angle, NULL, SDL_FLIP_NONE);
}

SDL_Rect Asteroid::get_dest()
{
    return dest;
}
```

Bullet.hpp & Bullet.cpp

Esta es la clase que define los objetos bala/rayo/disparo.

Al igual que en el caso de los asteroides, queremos que todas las balas se muevan a la misma velocidad; por tanto, debemos normalizarlo primero (y multiplicarlo por un escalar para darles cierta velocidad).

Además, la forma en que se inicializa el ángulo es parecido a como pasa con la nave del jugador. Aun así, hay una clara diferencia: cuando giras una bala 180 grados, se ve exactamente igual que sin haber hecho el giro. Esto nos permite simplificar un poco esa parte del código.

Como pasaba con los asteroides, las balas también desaparecen al salir de la ventana; si no, estaremos cometiendo un grave error de *memory leaking*.

Por último, también comprueba, como hacía la nave del jugador, la colisión con los asteroides con el algoritmo AABB.

```hpp
#pragma once
#include "SDL.h"
#include "Player.hpp"
#include "Asteroid.hpp"

class Bullet
{
    friend Asteroid;
public:
    Bullet(SDL_Renderer* renderer, Player* player, int dirX, int dirY);
    void update();
    void draw();

    bool collide(const SDL_Rect& asteroid);

    bool active = true;

    static inline SDL_Texture* texture;
private:
    SDL_Renderer* renderer;

    const SDL_Rect src{ 0, 0, 16, 16 };
    SDL_Rect dest;
    int dirX;
    int dirY;

    double angle;
};
```

```cpp
#include "Bullet.hpp"
#include <cmath>

Bullet::Bullet(SDL_Renderer* renderer, Player* player, int inputX, int inputY):
    renderer(renderer)
{
    dest.w = 64;
    dest.h = 4;

    dest.x = player->get_x();
    dest.y = player->get_y();

    //normalise the vector first, as we want all bullets to be shot at the same speed
    int modulus = sqrt(inputX * inputX + inputY * inputY);

    dirX = inputX * 600 / modulus;
    dirY = inputY * 600 / modulus;

    if (dirX == 0)
    {
        angle = 90;
    }
    else
    {
        angle = atan(double(dirY) / double(dirX)) * 180 / M_PI;
    }

    SDL_Surface* surface = IMG_Load("media/bullet.png");
    texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_FreeSurface(surface);
}

void Bullet::update()
{
    dest.x += dirX / 10;
    dest.y += dirY / 10;

    if (dest.x + 64 < 0 || dest.x - 64 > 1080 || //if the bullet is out of bounds, inactivate it
        dest.y + 64 < 0 || dest.y - 64 > 720)
    {
        active = false;
    }
    else
    {
        active = true;
    }
}
```

```cpp
void Bullet::draw()
{
    SDL_RenderCopyEx(renderer, texture, &src, &dest, angle, NULL, SDL_FLIP_NONE);
}

bool Bullet::collide(const SDL_Rect& asteroid)
{
    if (dest.x < asteroid.x + asteroid.w && //check AABB collision with an asteroid
        dest.x + dest.w > asteroid.x &&
        dest.y < asteroid.y + asteroid.h &&
        dest.y + dest.h > asteroid.y)
    {
        active = false; //if a collision is found, then the bullet gets deactivated
        return true;
    }

    return false;
}

```

RNG.hpp & RNG.tpp

Creé un *namespace* como interfaz para la generación de números pseudoaleatorios. Este *namespace* sería equivalente a lo que en otros lenguajes de programación serían las clases estáticas.

Las funciones *template* no pueden ser implementadas en ficheros *.cpp*. Sin embargo, como suelo reservar los archivos de cabecera *.hpp* para los prototipos de funciones, y no, para sus implementaciones, creé un archivo *.tpp*, que, por convención, se ha solido utilizar muchas veces en estos casos.

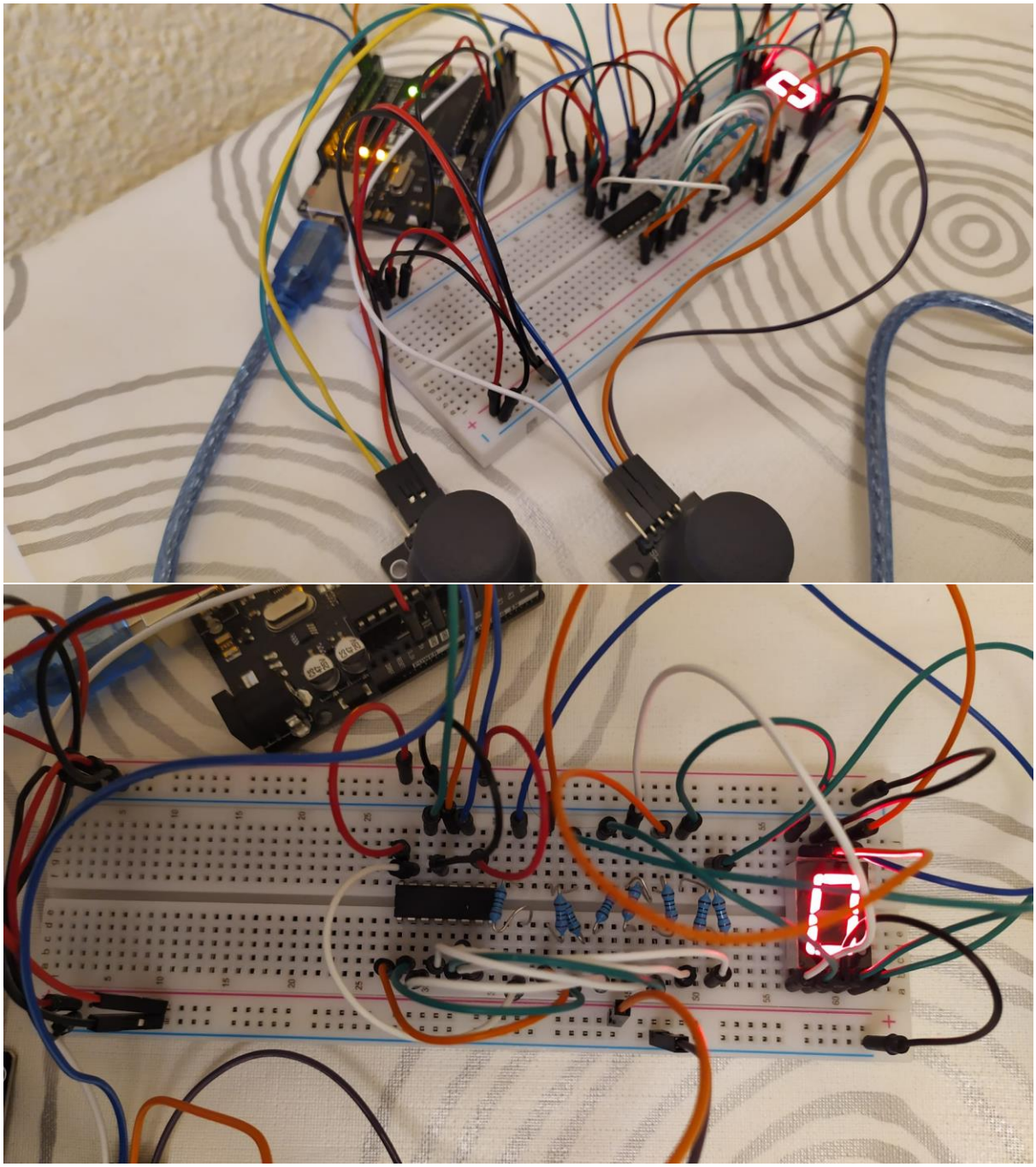
Que la función sea *template* permite que todas las llamadas a *generate* con el mismo intervalo hagan uso del mismo *uniform_int_distribution*, gracias a que esta está declarada como estática.

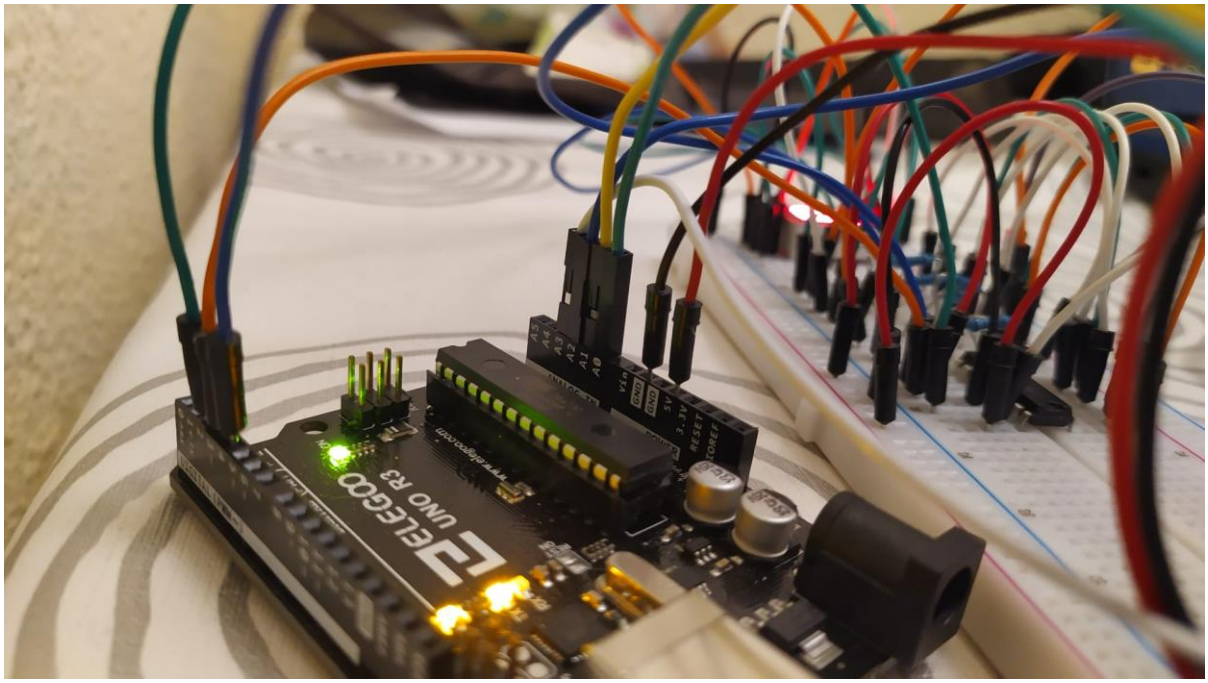
Si no fuera *template*, todas las llamadas a la función harían uso del mismo *uniform_int_distribution*, ignorando si el intervalo es realmente el mismo. Y, si esta variable no fuera estática, entonces tendría que crearse una cantidad innecesariamente alta de veces.

```cpp
#pragma once
#include <random>

namespace RNG
{
    template<int A, int B>
    int generate();

    namespace //anonymous namespace to simulate private membership
    {
        static inline std::random_device rd;
        static inline std::mt19937 rng(rd());
    }
}

#include "RNG.tpp"
```

```cpp
#pragma once
#include <random>

namespace RNG
{
    template<int A, int B>
    int generate()
    {
        static std::uniform_int_distribution<int> uid(A, B); //create distribution for a given interval
        return uid(rng);
    }
}
```

Montaje completo





Conclusión

En general, no ha sido un proyecto complicado, y lo he podido realizar en un corto lapso de tiempo. Me habría gustado haberme organizado mejor para poder haberlo podido refinar un poco, pues hay bastantes cosas que mejorar.

Por ejemplo, con polimorfismo se podrían haber creado distintos tipos de asteroides, cada uno con funcionalidades distintas (algunos que dieran vida, otros que rebotasen en los bordes, etc.); o también podría haber aprovechado para hacer más flexible el programa mediante el uso de lo que se conoce como *delta-time*; algo para lo que tengo ya hecha una interfaz bastante completa por otro proyecto mío, pero por las prisas ni si quiera consideré meterlo.

Sobre todo, tuve problemas a la hora de realizar la conexión serie. No sabía muy bien cuánto *delay* tenía que usar por ninguna de las dos partes, y eso me robó un buen rato. Además, todavía no sé si habrá alguna mejor manera de hacerlo, ya que, como lo tengo hecho, el programa reacciona como medio segundo tarde a todas las acciones realizadas en los *joysticks*, lo cual es bastante molesto para un videojuego, donde la inmediatez suele ser casi imprescindible.

De todas formas, me ha sido de ayuda para cuestiones de proyectos más personales. Por ejemplo, desde hacía tiempo estaba pensando en realizar la interfaz para el generador de números pseudoaleatorios, pero, seguramente por pura pereza, nunca lo acabé haciendo…, hasta este momento que lo he comenzado.

Además, gracias a este proyecto, he podido comprender, al menos a un nivel básico, cómo funciona la conexión serie por los puertos USB del ordenador. Es algo sobre lo que no tenía idea en el pasado, y, aunque no sea el tema que más me llama la atención, el conocimiento nunca viene mal, y me ha resultado bastante interesante.