Sprites and Concurrency Lab
AI Programming for Games
COMP10068

Paul Keir

Date Issued: March 25, 2025
Instructor: Dr. Paul Keir

In this week's lab we will look both at how to animate sprites in the raylib videogame programming library; and the handling of asynchronous function calls: which return immediately, while performing work in another thread: concurrency.

CMake

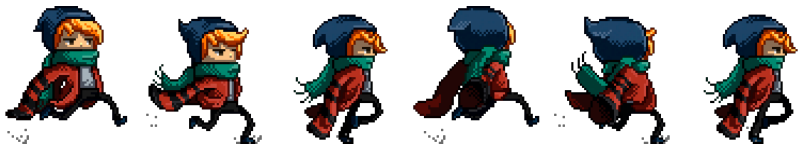
CMakeLists.txt is within the **code** subdirectory. Open **cmake-gui**, then copy and paste the directory path containing CMakeLists.txt in to the top field of CMake GUI: "Where is the source code"; then paste the same directory in the lower CMake GUI field: "Where to build the binaries", but here append **\build** to the path. Then click the following in sequence:

1. Configure
2. Generate
3. Open Project

Raylib Sprites & Animation

We start by exploring the use of PNG sprite sheets in raylib; aiming for a 2D top-down C++ RPG-style game environment, where characters can move, pick up objects, and interact. This leads towards the goal of interacting with "llama.cpp" within a game environment.

Animated Sprites (01-scarfy.cpp)



Build and run the 01-scarfy project in Visual Studio. You should see an animated character with a scarf "Scarfy" running through a 6-frame animation sequence. Look at the C++ code, and try to answer the following questions:

1. Where in the code does the animated frame change?
2. Where is the position of Scarfy defined?

Now make the following modifications to the code within 01-scarfy.cpp. At each step, check by eye that the program is running as you expect.

1. Change the animation to run at double the current rate.
2. Change the position that the sprite is drawn on the screen.
3. Add a new `float` variable before the `while` loop called `angle`, and set it to zero.
4. Comment our the call to `DrawTextureRec`, and replace it with the following:

```
angle += 2.0f;
const Rectangle dest{ position.x, position.y,
                      (float)scarfy.width/6, (float)scarfy.height};
Vector2 origin{ 0.0f, 0.0f };
DrawTexturePro(scarfy, frameRec, dest, origin, angle, WHITE);
```

5. Can you change the rotation so that it revolves around the sprite's centre?
6. Add a second call to `DrawTexturePro`, and use it to draw a second sprite, at a different position, which is not rotating.
7. Slow the animation of the second sprite to 3 fps. To do this:
    1. Declare and initialise a second set of variables like `position`, `frameRec`, `currentFrame`, `framesCounter` and `framesSpeed`. Name them `position2`, `frameRec2`, `currentFrame2`, `framesCounter2` and `framesSpeed2`.
    2. Then add a second `if` statement, and use it to reduce the animation rate of the second sprite. `frameSpeed2` is the crucial value here.
8. Use `currentFrame2--` to change the second animation to run backwards.

Two Dimensional Sprite Sheets (02-sprite-class.cpp)

The 02-sprite-class project uses a `Sprite` class which can accommodate varied 2D sprite sheets, including those from the Time Fantasy pixel art website. Change the startup project in Visual Studio to **02-sprite-class** and look at sprite.hpp header file containing the `Sprite` class definition. The `Sprite` constructor is shown below.

```
Sprite(const raylib::Texture& tex,
       const int ncols = 1, const int nrows = 1, const Vector2 posn = {},
       const std::map<std::string, std::vector<int>>& all_frame_ids = {{ "", { 0 } }},
       const int sprite_fps = 0);
```

Make the following modifications to the program:

1. The audio file `coin.wav` has already been loaded. Add a call to the `Play` member function of the `coin_sound` object when the grey knight (i.e. the player) comes within 30 units of the grim reaper.

Concurrency

We now turn to the C++ standard library function template `std::async`. The `std::async` function template will immediately return an `std::future` object. We can then call member functions of `std::future` such as `wait`, `wait_for`; and then conclude with a call to `get`.

Time and Sleep (03-chrono-sleep.cpp)

We want the call to `take_a_break` to pause for 5 seconds. Add a call to `std::this_thread::sleep_for` inside `take_a_break`. `sleep_for` is looking for an argument such as `std::chrono::seconds(5)`. Note that we have included the standard header files: `chrono` and `thread`.

1. Change the animation to run at double the current rate.

Without Function Parameters (04-async-wait.cpp)

Use `std::async` to call `internet_request` to remove the (now unwelcome) pause between "Hello" and "World". Use `std::future::wait` to wait until `internet_request` has been evaluated (and then also call `std::future::get` to ensure the future is reset to its default invalid state). After that, "Goodbye" should be printed. (In

fact, `std::future::wait` is not required, as `std::future::get` calls `std::future::wait` anyway.)

With Function Parameters (05-async-wait-param.cpp)

Now `internet_request` takes an argument; and returns a value (both are integers). Again use `std::async` to call `internet_request` to remove the pause between "Hello" and "World". Change the template type of `std::future` from `void` to `int`. Use `std::future::get` again, but note that it will now return the `int` returned by `internet_request`. Use that value to update `x` and display it.

Function Reference Parameters (06-async-wait-param-ref.cpp)

Here we will use a different approach. `internet_request` now returns nothing, but its argument is passed *by reference* (see the ampersand in the `int&` parameter type). This is more efficient: `internet_request` now works with the same `x` which is declared at the top of the `main` function. Again use `std::async` and `std::future::get`, but you must wrap the `x` argument you pass to `std::async` with `std::ref`; as in `std::ref(x)`.

This time the call to `std::future::get` does not return anything (as `internet::request` doesn't return anything. But after `std::future::get`, we can be sure that variable `x` has been updated.

Member Function Reference Parameters (07-async-wait-param-ref-member.cpp)

In this project, the function we want to manage via `std::async`, is a class member function. This requires an extra 3rd parameter to `std::async`: the address of the object which contains the member function. In this case the object has been named `inet`, and so its address is `&inet`.

Polling with a While Loop (08-async-wait-param-ref-member-while.cpp)

This time we will use a (game-like) while loop, and check on the status returned by a call to `std::future::wait_for` with a zero time argument; such as `std::chrono::seconds(0)`. The `if` statement should be set `true` when `std::future_status::ready` is returned by `std::future::wait_for`.

Graphics Context (09-raylib-concurrency.cpp)

The last project is close to the situation in the assignment; though much simpler; and with less code. When the Enter key is pressed, there will be an unwelcome pause in the game loop; halting both the movement of the sprite, and the music. The solution will be similar to the last project with the `while` loop, but here, each time you check the value returned by `std::future::wait_for`, you must first check that your `std::future` is in a valid state, using the member function: `std::future::valid`. The future will become valid after being assigned to from a call to `std::async`; and invalid after a call to its member function `std::future::get`.