CPL: Rust

Project Assignment: Developing a Window Manager

Version 1.0

Contents

1 What is a Window Manager?

In this project your task is to build a **window manager**. A window manager is the piece of software that controls the placement of windows, and the interaction of the user with these windows, e.g., clicking on a window to focus it, dragging a window around, resizing it, selecting another window when the users presses `Alt-Tab`, etc.

The window manager you will be developing is for the X Window system (also known as X11), the most used windowing system[1] on Linux and UNIX-like operating systems that has been around since 1984. X11 window managers are typically responsible for the following tasks:

- When a new window is created, the window manager displays it at a certain location on the screen. This can be determined by the application, the user preferences, and/or the window manager.
- Focusing a window when the users hovers onto it or clicks on it.

[1] Some newer Linux distributions are using the more modern Wayland display server.

- Resizing and moving windows using the mouse.
- Bringing a window to the front when the user clicks on it.
- Drawing the title bar with the minimise, maximise, and close buttons. The window manager you will be developing will not have to do this.
- (Un)minimising, (un)maximising windows.
- Switching between **workspaces**, which involves hiding and showing different sets of windows, according to the current workspace. This is known as Virtual Desktops in Windows 10 and Spaces in Mac OS X.
- Make a window fullscreen.
- **Tile** windows according to some layout. Tiling window managers, like Awesome[2] or XMonad[3], do things differently. The first window that appears will be maximised on the screen. When a second window appears, the screen is split (horizontally or vertically) in two tiles. Each window will be displayed in a different tile. A third window will cause one of the existing tiles to split again, and so on. Closing a window will make the corresponding tile disappear. The idea is that the screen space is maximally used at all times, there is never an unused gap. With a tiling window manager you will never see your desktop background unless you close all windows. These window manager are often very configurable as well as their tiling layout. Such window managers are mainly controlled using the keyboard. Look up some videos of tiling window managers on YouTube to better understand what they do.

When you complete this project including all its optional assignments, you will have built a window manager that is capable of doing all these things and more.

Developing a window manager for X11 is not so hard, as demonstrated by the long list[4] of window managers written over the years. Unlike for other display servers, an X11 window manager is not responsible for the actual rendering of windows or anything complex involving graphics. An X11 window manager connects to the display server, sets up some stuff, and then starts listening for events. Some of these events include:

- `XMapRequestEvent`: a client (an application like Firefox) wants to **map** (to display) a window. The window manager can just call the appropriate library function (in this case `XMapWindow`) to map the window.
- `XKeyEvent`: the user has pressed a key that the window manager has registered as a hotkey. For example, when the user pressed `Alt-Tab`, the window manager should focus the next window. This is done by determining which window is next, focusing it and unfocusing the previously focused window using the appropriate library function(s).
- `XEnterWindowEvent`: the user has moved the mouse onto another window. The window manager must then focus the window the mouse has 'entered' and unfocus the window the mouse has 'left'.

[2] `https://awesomewm.org/`
[3] `http://xmonad.org/`
[4] `https://wiki.archlinux.org/index.php/Window_manager`.

2 Approach

The hardest parts of developing a window manager for X11 are: understanding the protocol of X11, described in the Inter-Client Communication Conventions Manual[5], and figuring out how to use the Xlib library, which provides the functions that manipulate windows. As this library is written in C, it is very easy to get things wrong.

To make things easier for you we take the following approach. We split the window manager in two parts:

1. The **Backend** does the binding with the Xlib library, handles the event loop, translates the X11 protocol into simpler actions, and takes care of all the low-level dirty unsafe C stuff.[6]
   We provide the backend.
2. The **Window Manager** provides the window layout logic: it determines the size and location of every window, the stacking order of the windows (which window is on top, which window is below, etc.), and which window is focused. The window manager doesn't know anything about the backend.[7]
   You will develop the window manager.

The window manager works passively. The backend responds to events by calling the appropriate methods of the window manager. After each event, the backend queries the window manager for the layout of the windows on the screen and then calls the appropriate library functions to make sure the window manager actually displays the desired window layout. For example, when a new window is created, the following happens:

- The backend's event loop receives a `XMapRequestEvent` for the window.
- The backend calls the `add_window` method of the window manager with the new window and its size and location.
- The window manager adds the window to its data structures.
- The backend queries the window manager for its updated window layout using the `get_window_layout` method.
- The window manager calculates the window layout, e.g., the new window is displayed fullscreen.
- The backend compares this layout with the layout of the window manager before the event was handled (the backend remembers the previous layout).
- The backend calls the appropriate Xlib functions to make sure the current window layout is displayed by the display server, i.e. the window will be shown fullscreen, it will be stacked on top of the other window, and it will be focused.

The window manager can be written in pure Rust, without any need to call external C library functions. The only thing that the window manager must do is update its state so

[5] `https://tronche.com/gui/x/icccm/`

[6] If you are interested in writing your own backend, have a look at `https://seasonofcode.com/posts/how-x-window-managers-work-and-how-to-write-one-part-i.html`

[7] This means that you can use your window manager on another display system if you write the appropriate backend.

that the next time `get_window_layout` is called, it can return an updated window layout.

The `WindowManager` trait

We use Rust's traits to decouple the two: a backend must implement the `Backend` trait and a window manager must implement the `WindowManager` trait. We will now walk through **some** of the important methods of the `WindowManager` trait to give you an idea of what a window manager must do. Note that we gloss over some of the details and present **a simplified version** of the trait that differs from the real implementation. The documentation[8] of the trait is very extensive and should explain all the details.

Rust

```rust
trait WindowManager {
    fn windows(&self) -> Vec<Window>;
    fn get_focused_window(&self) -> Option<Window>;
    fn add_window(&mut self, window: Window, geometry: Geometry) -> ..;
    fn remove_window(&mut self, window: Window) -> ..;
    fn focus_window(&mut self, window: Option<Window>) -> ..;
    fn get_window_layout(&self) -> WindowLayout;
    ..
}
```

- `windows`: the window manager must be able to return a `Vec` of all the windows managed by the window manager. The order of the windows in this `Vec` does not matter. A `Window` is only an identifier, it is represented by an unsigned long.
- `get_focused_window`: return the window that is currently focused according to the window manager. If no window should be focused, return `None`.
- `add_window`: add a new window along with its **geometry**. This is called whenever a new window is created. The `Geometry` defines the location and size of the window, and is defined as follows:

Rust

```rust
struct Geometry {
    x: c_int,
    y: c_int,
    width: c_uint,
    height: c_uint,
}
```

- `remove_window`: remove the given window from the window manager. This is called whenever a window is destroyed/killed.
- `focus_window`: focus the given window, or when passed `None`, focus nothing. This is called when the user clicks on or hovers onto a window, or changes the focus

[8] `https://people.cs.kuleuven.be/~thomas.winant/cpl/doc/cplwm_api/wm/trait.WindowManager.html`

using the keyboard.

- `get_window_layout`: return the current window layout according to the window manager. The `WindowLayout` is defined as follows:

```rust
struct WindowLayout {
    focused_window: Option<Window>,
    windows: Vec<(Window, Geometry)>,
}
```

This struct contains everything needed to determine the window layout. The `windows` field contains a `Vec` of all the visible windows, along with their location and size. The order of this `Vec` determines the stacking order of the windows: the first window is at the bottom, the last window is at the top (because it's easy for a `Vec` to add at the end using `push`). Finally, the `focused_window` field determines which window should be focused, or whether no window should be focused.

By decoupling the window manager from the backend, it becomes much easier to develop a window manager. In fact, you can safely ignore the backend altogether during this project. Also, it becomes much easier to test a window manager: we can just create one without even having an X server running, call some of its methods to simulate adding and moving around some windows, and can then simply inspect the window layout returned by the window manager. Note that this also means you can develop this project on any platform (Linux, Windows, Mac OS, ...) because you don't need an X server to compile and test your code.

3 Assignment

We expect you to spend at most **40 hours** on this assignment. The project consists of a number of mandatory assignments and a number of optional assignments. You are not required to complete the optional assignments, that's what **optional** means.

When you complete all mandatory assignments correctly, and write readable, documented, and well-structured code, you will get at least a **passing grade**. The more optional assignments you complete, the higher your grade will be. Keep in mind that when you only hand in the mandatory assignments, and your code doesn't work correctly or is a complete (undocumented) mess, you will not get a passing grade. Therefore, we recommend that you implement at least one optional assignment. You will still lose points for handing in a complete (undocumented) mess, and we strongly advise against it.

In each assignment you will write a different window manager, often building further on the previous one. We provide a **template file** for every assignment in the `assignment/src` folder, **please use the right file for the right assignment**. At the top of each module, there will be some TODOs that you have to complete: indicate whether you completed the assignment, etc. You are free to add additional functions, traits, data type,

modules, etc. You will often need them. However, you are not allowed to modify or rename the given ones.

The documentation of the assignment can be consulted online at:
`https://people.cs.kuleuven.be/~thomas.winant/cpl/doc/cplwm_assignment/`
It is also possible to build the documentation yourself, see the Hints & Tricks section.

3.1 Mandatory Assignments

A: Fullscreen Window Manager

Implement the `WindowManager` trait by writing a simple window manager that displays every window fullscreen. When a new window is added, the last window that was visible will become invisible.

B: Tiling Window Manager

Write a more complex window manager that will **tile** its windows. Tiling is described in 1. Your window manager must implement both the `WindowManager` trait and the `TilingSupport` trait. See the documentation of the `TilingSupport` trait for the precise requirements and an explanation of the tiling layout algorithm.

C: Floating Windows

Extend your window manager with support for floating windows, i.e. windows that do not tile but that you move around and resize with the mouse. These windows will **float** above the tiles, e.g. dialogs, popups, video players, etc. See the documentation of the `FloatSupport` trait for the precise requirements.

Either make a copy of the tiling window manager you developed in the previous assignment and let it implement the `FloatSupport` trait as well, or implement the `FloatSupport` trait by building a wrapper around your tiling window manager. This way you won't have to copy paste code. Note that this window manager must still implement the `TilingSupport` trait.

D: Minimising Windows

Extend your window manager with support for (un)minimising windows. i.e. the ability to temporarily hide windows and to reveal them again later. See the documentation of the `MinimiseSupport` trait for the precise requirements.

Either make a copy of the tiling window manager with support for floating windows you developed in the previous assignment and let it implement the `MinimiseSupport` trait as well, or implement this trait by building a wrapper around the previous window manager. Note that this window manager must still implement all the traits from previous assignments.

3.2 Optional Assignments

You can choose which optional assignments you complete and in which order you do so. Remember that we advise to at least complete one optional assignment.

E: Fullscreen Windows

Extend your window manager with support for fullscreen windows, i.e. the ability to temporarily make a window take up the whole screen, thereby obscuring all other windows. See the documentation of the `FullscreenSupport` trait for the precise requirements. Don't confuse this with the first assignment, in which you built a window manager that displayed all windows fullscreen.

Like in the previous assignments, either make a copy of, or define a wrapper around your previous window manager to implement the `FullscreenSupport` trait as well. Note that this window manager must still implement all the traits from previous assignments.

F: Gaps

Extend your window manager with support for gaps, i.e. the ability to add some space between the different tiles. See the documentation of the `GapSupport` trait for the precise requirements.

Make a copy of your tiling window manager from assignment B and let it implement the `GapSupport` trait. You are not required to let this window manager implement all the previous traits.

G: Multiple Workspaces

Extend your window manager with support for multiple workspaces. The concept of workspaces is described in Section 1. See the documentation of the `MultiWorkspaceSupport` trait for the precise requirements.

Unlike the previous assignments, you are not allowed to make a copy of your previous window manager. You **have** to define a wrapper implementing the `MultiWorkspaceSupport` trait. This wrapper can take any existing window manager and uses it to create the different workspaces. This wrapper must also implement all the traits you have implemented in the other assignments, you can forward them to the window manager of the current workspace.

H: Different Tiling Layout

Come up with a different tiling layout algorithm than the one you have already implemented. If you are uninspired, feel free to look for one on the internet,[9] but **don't forget to mention where you found it**. The layout algorithm **may not be trivial**, e.g., not just adding tiles by splitting the screen horizontally, and must be at least as complex as, but different enough from the original layout algorithm you already have had to implement.

[9] XMonad has a whole bunch of layouts (look under the Layout heading): `http://xmonad.org/xmonad-docs/xmonad-contrib/XMonad-Doc-Extending.html`

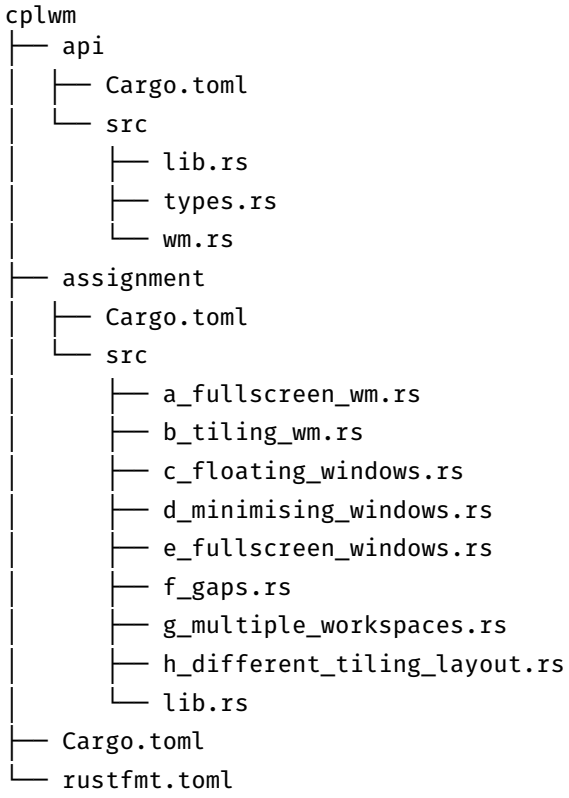

Figure 1: Directory structure of the project

Make a copy of your tiling window manager that implements the tiling layout algorithm. This window manager has to implement the `WindowManager` trait, but **not necessarily** the `TilingSupport` trait, as not every layout has a master tile. Feel free to add additional methods to your window manager that can be used to manipulate its layout. You are not required to let this window manager implement all the previous traits.

3.3 Tests

During grading we will be using a number of tests to verify whether your window manager implementations correctly implement the interfaces and don't violate the invariants stated in the trait documentation.

We **strongly advise you to write your own tests**. To help you get started, the tutorial mentioned in the next section also includes a description of how to write a simple test.

3.4 How to Get Started

1. As this project consists of multiple modules, you won't be able to complete it using the online Rust Playground. You must **install Rust** on your computer by following the instructions at `https://www.rust-lang.org/downloads.html`.
2. **Unzip the project** and you will see the directory structure of Figure 1.

   We organised this crate (the name for a Rust project) in two subcrates: `api` and

`assignment`. The `api` crate contains the definitions of the various traits you will implement and some basic types. You are not allowed to edit this folder. The `assignment` crate is where you will develop your own window managers. We have provided a template for each assignment.

The `lib.rs` files are the main modules of each crate. The `Cargo.toml` files are the crate config files that define each project and their dependencies. As we will discuss in Hints & Tricks, you may depend on additional crates. You can add these dependencies by modifying the `assignment/Cargo.toml` file. The `rustfmt.toml` configures the `rustfmt` formatting tool, which is also mentioned in Hints & Tricks.

3. **Compile the project** using Rust's package manager, Cargo. By running `cargo build` in the `assignment` folder. Cargo is installed together with Rust. For more information about Cargo, see its tutorial: `http://doc.crates.io/guide.html`.
4. **Read the documentation** of the `assignment` crate[10] and/or **look at the source code** of `assignment/src/lib.rs`.
5. **Read the documentation** of the `a_fullscreen_wm` module[11] and **look at the source code** of `assignment/src/a_fullscreen_wm.rs`. This file contains a tutorial on how to write your first window manager and how to write a test.

4 Practical Matters

4.1 Deadline

The deadline for this project is **2016-12-23 23h59**. There will be a Toledo assignment to submit your project.

Submit your project as a zip file of the `assignment` folder, as this is the only folder you are allowed to edit. Don't forget to **complete the TODOs** at the top of each module.

Make sure your project **compiles without errors**, otherwise we will subtract points. Also, try to **fix all warnings** (and not by asking the compiler to stop generating them), they often hint at problems in your code. We also advise you to format your code as discussed in Hints & Tricks.

4.2 Plagiarism

The code that you turn in must have been written by you, and by you only. You are welcome to discuss your work with other students and with teaching assistants, but you are not allowed to copy or share pieces of code with others. You are not allowed to make your code available to others, and you are not allowed to use code made available by others.

[10] `https://people.cs.kuleuven.be/~thomas.winant/cpl/doc/cplwm_assignment/`
[11] `https://people.cs.kuleuven.be/~thomas.winant/cpl/doc/cplwm_assignment/a_fullscreen_wm/`

4.3 Forum

If you have a question about the project or found a bug in the code we gave you, post it on the **discussion forum on Toledo**. This forum will be actively monitored by the assistant.

This forum is not a help desk, questions like: "What does `&self` mean?" will not be answered, you should know how to use Google by now. Keep the rules about plagiarism in mind, so don't share code, use abstract examples.

4.4 How can I run my Window Manager?

The X11 backend of the window manager and instructions on how to actually use your window manager in practice will be made available at a later date. Remember that you can complete the whole project without any backend.

4.5 Rust Version

We expect you to use a stable version of Rust (1.12.1 is the latest stable version at the time of writing). When a new stable version of Rust is released, 1.13 or even 1.14, you are allowed, but not required to switch to it. You typically don't have to worry about things no longer working with newer versions of Rust.

While browsing the Rust APIs, you will probably encounter certain methods and **features** that are **unstable**. These things are not available in a stable version of Rust, and require a **nightly** build. If you have a good reason to use one of those unstable things, you may use a nightly build, as long as you state which version and your reasons for it in `assignment/src/lib.rs`.

5 Hints & Tricks

- **Documentation**: there are three important sources of information about Rust:
  - **The Rust Book**: `http://doc.rust-lang.org/book/`
    For when you can't remember what Ownership is or if you don't know how Rust does Error Handling.
  - **Rust by Example**: `http://rustbyexample.com/`
    For when you want to know what the syntax of a **struct** is or how to define **methods**.
  - **The Rust API documentation**: `http://doc.rust-lang.org/std/`
    For when you want to know what methods are available for a `Vec` or what the interface is of `Iterator`.
- **Extra dependencies**: you are free to add additional dependencies from `https://crates.io` to your project. It is certainly possible to complete all assignments without any additional dependencies, but some crates might provide useful utilities (for example the `itertools` crate) or additional data structures. The Cargo

(Rust's package manager) guide[12] clearly explains how to add dependencies to your project.

- **Extra modules**: you are free to add additional modules. Moreover, you are encouraged to do so whenever a file becomes too large. The chapter on Crates and Modules in the Rust Book explains how to add modules.
- **Formatting**: use the `rustfmt`[13] tool to automatically format your code. You can simply invoke `cargo fmt` in the `assignment` folder and all your code will be formatted. One less thing to worry about.
- **Generate documentation**: you can generate the documentation of your project using `rustdoc`. By running `cargo doc --open` in the `assignment` folder the documentation will be generated and opened in the web browser. Documentation is written in the Markdown format.

[12] `http://doc.crates.io/guide.html#adding-dependencies-from-cratesio`
[13] `https://github.com/rust-lang-nursery/rustfmt`