Using orxhub

Summary

In this tutorial, we're going to start a fresh project using OrxPM, the Orx Package Management tool. We're going to install packages from orxhub(the official OrxPM repository), handle merge conflicts, then again install new packages later on in the project to experience how OrxPM works during the lifetime of a project. The tutorial gives a detailed tour of using OrxPM, but all it takes to start using OrxPM(and the packages on it) is the following:

- Copy orxpm.sh into an empty folder
- Create a file named `orx_packages` containing names of desired packages on each line. (For an empty project, just write `minimal` in `orx_packages`.
- Run: (**Note for Windows Users:** Run these commands in git bash shell.)

```
git init
./orxpm.sh
```

- Create a folder named `.build`
- Inside that folder, run:

```
cmake ../src
```

And the project files for your platform are generated in the `.build` folder.

Starting a New Project

All it takes to start an OrxPM powered project is a working git installation and the orxpm.sh script. You can use the following steps to start a new project **in any git repository**, but to set you up for a merge conflict (which we'll learn how to deal with) we're going to start by cloning the official orxhub repository (I keep referring to it as the “official” repository since anyone can easily fork their own.). So, in any folder of your liking run the following command: (**Note for Windows users:** The CLI commands given in this tutorial depend on the BASH shell, which is provided by your git installation. So, please run them under bash provided by git.)

```
git clone https://github.com/orx/orxhub.git tutorial_project
```

This creates a new folder with the name `tutorial_project`. Now let's go into that folder and see what there is inside:

```
cd tutorial_project
ls
```

This will show `README.md` and `orxpm.sh`. Normally, all you need to start using OrxPM in **an existing git repository** is to copy `orxpm.sh` into its root.

In order to install packages from OrxPM, one lists the names of those packages in a file named `orx_packages` and runs `orxpm.sh`:

In the repo root: (Note that this will result in an intentional merge conflict, so don't panic)

```
echo minimal > orx_packages
./orxpm.sh # merge conflict expected here
```

These commands will install the `minimal` package from orxhub. This package contains the template for an empty Orx project that can compile against multiple platforms and it's the base of all packages.

As expected, a merge conflict occurred on the `README.md` file, as the `minimal` package is also trying to bring a `README.md` file. OrxPM works by merging the requested packages into your repository and those packages and the `minimal` template is designed to minimize any unnecessary merge conflicts. The entire project structure is designed so that the packages can work together without stepping on each others' toes, but still, an occasional merge conflict is possible and there's no need to panic, we can resolve the conflict and use the packages.

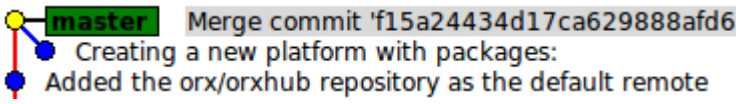
If we now inspect `README.md` we'll see that git has marked the conflict. Now let's delete everything in that file except for the line:

```
Please refer to the [build instructions](doc/build_instructions.md) for
building this project.
```

Now let's finish the merge by issuing the following commands:

```
git add README.md
git commit --no-edit # This keeps the merge commit message
```

At this point, this is how the commit tree looks:



See how OrxPM created a new “platform” commit with all the packages you requested and merged it into your repository. Now, if you look around your repository, you'll notice that many files and folders have appeared. The file tree looks something like this:

```
REPOROOT
├── android
├── data
│   ├── android
│   │   ├── android_debug.ini
│   │   └── android.ini
│   ├── common
│   │   └── common.ini
│   ├── desktop
│   │   └── desktop.ini
│   └── game.ini
├── doc
```

```
│   └── build_instructions.md
├── orxpm.sh
├── README.md
└── src
    ├── android
    ├── cmake
    ├── CMakeLists.txt
    ├── common
    │   └── main.cpp
    ├── desktop
    ├── lib
    └── orxpm
```

At this point, the interesting files are:

- **src/common/main.cpp:** This is where the game's `main()` lives. In the minimal package, it has the bare minimum code to keep the future packages together.
- **src/CMakeLists.txt:** This is the game's build configuration. Most users will probably never have to modify it.
- **src/common, src/desktop, src/android:** These are platform aware source folders. Every `.c` and `.cpp` file under `desktop` and `common` gets compiled and linked to desktop builds (Linux, Win, Mac) and everything under `android` and `common` goes into the Android build. 99-100% of a game's code should typically live in the `common` folder.
- **data/common, data/desktop, data/android:** These folders are similar to their counterparts under the `src` folder.
- **data/desktop/desktop.ini:** This is the orxConfig entry point for the desktop builds.
- **data/android/android.ini, data/android/android_debug.ini:** These are the Android and Android debug build orxConfig entry points respectively.
- **data/common/common.ini:** This empty .ini file gets included by all the builds, so it's probably best place to start building your game, unless you want to do something specific to the platform.
- **doc:** This is the documentation folder. When you install packages, they put their documentation in this folder. The `minimal` package currently only has documentation on how to build this project, more documentation will likely be added to that package in the future though.

Working with the Project

Now, Let's start hacking :)

The first thing we do is to compile the project as it is now. For that, we need to have a C/C++ compiler installed on our system, as well as the CMake cross-platform build tool. Having these, go the project root folder and run the following commands:

```
mkdir .build # You can have multiple build folders
cd .build
cmake ../src -DCMAKE_BUILD_TYPE=Debug # Other options are Release and
RelWithDebInfo
```

CMake has many many options on how to configure your project. The commands above will create the

most basic project files for your platform and build tools. You can make it generate project files for IDEs like Eclipse or CodeLite, but those things are out of scope for this tutorial. Since this tutorial is meant to be platform-independent, I'll leave it to you to figure out how to perform the compilation. On Windows you'll typically have solution files that you can open in MSVC, or a Makefile if you have MinGW. On Linux, you'll typically get a Makefile, and on Mac, you'll probably get XCode project files. From this point on, I'll assume that you've figured out how to compile the project.

I won't go into the details, but imagine that we create the familiar running soldier object from the official Orx Anim Tutorial. We then assign it a timeline track so that it endlessly patrols to the left and right. Here's what we change in the project files:

data/common/common.ini

```
[MainViewport]
Camera = Camera

[Camera]
... ; Please see the "what we change in the project files" link above for
details

[Input]
...

@soldier.ini@
```

data/common/soldier.ini

```
[Soldier]
AnimationSet = AnimSet
TrackList = PatrolTrack
...

[PatrolTrack]
Loop = true
0 = Object.SetAnim ^ WalkRight # Object.AddFX ^ MoveRightFX
4 = Object.SetAnim ^ IdleRight
6 = Object.SetAnim ^ WalkLeft # Object.AddFX ^ MoveLeftFX
10 = Object.SetAnim ^ IdleLeft
12 = Object.SetAnim ^ IdleLeft

[AnimSet]
AnimationList = IdleRight#WalkRight#IdleLeft#WalkLeft
LinkList      = IdleRightLoop # IdleRight2Left # IdleRight2WalkRight #
                WalkRightLoop # WalkRight2IdleRight # IdleLeftLoop #
                IdleLeft2Right # IdleLeft2WalkLeft # WalkLeftLoop #
                WalkLeft2IdleLeft
```

```
... ; Frame, Animation and LinkList sections follow
```

data/common/soldier.png



src/common/main.cpp:

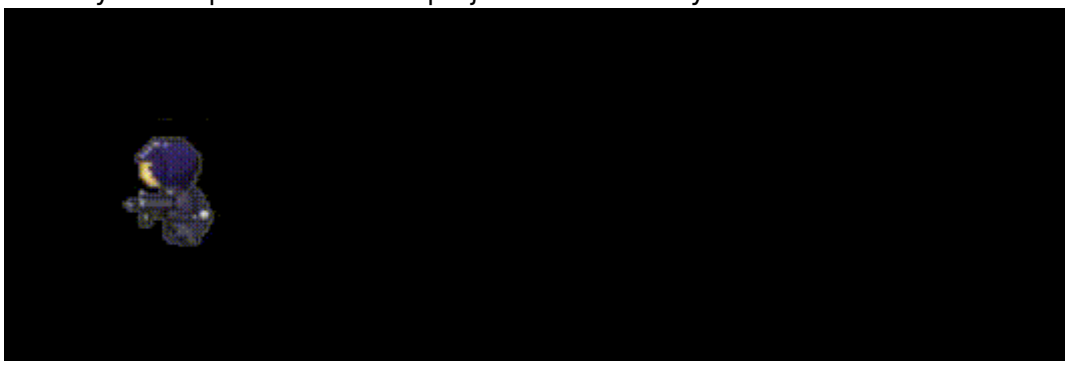
In the `Init()` function

```
orxViewport_CreateFromConfig("MainViewport");
orxObject_CreateFromConfig("Soldier");
```

In the `Run()` function

```
if(orxInput_IsActive("Quit")) return orxSTATUS_FAILURE;
```

Now if you compile and run the project here's what you'll see:



Let's commit the project now:

```
cd <repo root dir>
git add data/common/* src/common/main.cpp
git commit -m "Added a patrolling soldier"
```

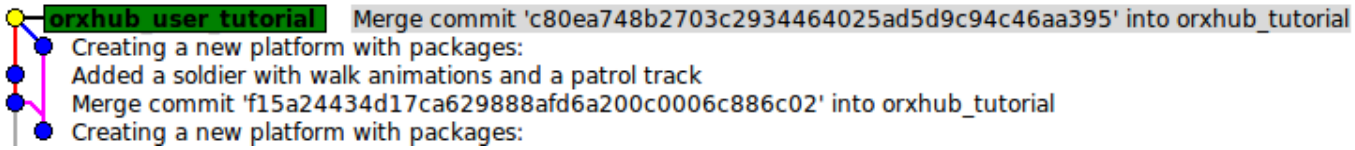
Installing New Packages

At this point, we learn that there's an orxhub package called `animgraph` that simplifies the definition of `AnimationSets` and decide to take advantage of it. Here's what we do; we first make sure that there are no uncommitted changes in the repository by issuing `git status`. It should say `nothing to commit, working directory clean`. Then we perform the following:

```
cd REPOROOT
```

```
echo animgraph >> orx_packages
./orxpm.sh
```

OrxPM will then merge the changes caused by added and removed(we haven't removed any but we could have) packages into our repository. Here's what our commit graph now looks like:



Note how OrxPM created a new commit with the required changes and merged it in. At this point, we have all the new packages we've requested downloaded, along with their dependencies and integrated into our game. For instance, the `animgraph` package registers a few orxCommands, and that's already handled for us, and we can start using them in Config right away!

Using the animgraph Package

Notice how the `animgraph` package brought a few files under the `doc/` folder. Some of those files have come from its dependencies, but the `animgraph` package itself is documented in the animgraph.md file. To understand what we're doing next, please quickly skim that file. We're now going to shorten our soldier.ini:

```
[AnimSet]
IR = IdleRight
IL = IdleLeft
WR = WalkRight
WL = WalkLeft
AnimationGraph = IR # IL # WR # WL # IR! : IL,WR : !IR # IL : !WL! : IL
%ConfigX.ProcessAnimGraph AnimSet

... ; all the link sections removed

[PatrolTrack] ; Notive how we've changed the SetAnim calls
0 = ObjectX.SetAnimByTag ^ WR # Object.AddFX ^ MoveRightFX
4 = ObjectX.SetAnimByTag ^ IR
6 = ObjectX.SetAnimByTag ^ WL # Object.AddFX ^ MoveLeftFX
10 = ObjectX.SetAnimByTag ^ IL
12 = ObjectX.SetAnimByTag ^ IL
```

Now, if you compile and run the game, you'll see that nothing is changed, but we've shortened `soldier.ini` by 41 lines! **Important Note:** On some platforms, when you try to compile the project now, the C/C++ linker will complain about some missing function definitions. That's because CMake hardcodes the name of the `.c/.cpp` files to be compiled. So, whenever you add/remove `.c/.cpp` files (which is often what happens when you fetch packages) either run CMake again manually, or do the following so that the project runs CMake automatically at the next build:

```
touch REPOROOT/CMakeLists.txt
```

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://www.orx-project.org/wiki/en/tutorials/projects/orxhub_user?rev=1598878881

Last update: **2025/09/30 17:26 (14 months ago)**

