B5 - Advanced C++ Programming

B-PAV-532

R-Type

A game engine that roars!

42.0

R-Type

binary name:	r-type
group size:	4-6
repository name:	cpp_rtype
repository rights:	ramassage-tek

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

For this project of the **Advanced C++** knowledge unit, **R-Type** will introduce you to networked video game development.

You will have to implement a multi-threaded server and a graphical client, using a re-usable game engine of your own design. Your game must also be fun to play!

Introduction

For the sad ignorants among you who may not know this best-selling video game, which accounts for countless lost hours of our childhood, here is a little introduction.

As you now understand, you have to make your own version of **R-Type**.

The purpose of this project is to create a one-to-four player game, using a client/server architecture. This is important. It **MUST** be a client/server architecture. Peer-to-peer communication is not allowed.

Server

The server **MUST** be multi-threaded.

The server **MUST** be able to handle more than one game at a time, it **MUST** be able to handle multiple games in a row, and it **MUST** be the referee of all games it manages.

Your abstractions' quality will be strongly evaluated during the final defense, so pay **close** attention to them.

Client

The client is the display terminal of the game.

It **MUST** contain anything necessary to display the game and handle player input.

You **MUST** use the **SFML** for this.

Here is a description of the official **R-Type** screen:

- 1: Player
- 2: Monster
- 3: Monster (that spawns a powerup upon death)
- 4: Enemy missile
- 5: Player missile
- 6: Stage obstacles
- 7: Destroyable tile
- 8: Background (starfield)

Requirements

+ Platforms

The project MUST be OS independent. It has to compile and run in a similar manner on at least one **Unix** system and one **Windows** system.

+ Protocol

You **MUST** design a binary protocol for client/sever communications.

You **MUST** use UDP for communications between the server and the clients. A second connection using TCP can be tolerated but you **MUST** provide a quite strong justification. In any event, ALL in-game communications **MUST** use UDP.

You **MUST** document your protocol. The documentation **MUST** be an RFC. RFC format is described in RFC 2223.

Your RFC **MUST** be formatted like an official RFC.

You **MUST** write the RFC in ASCII format. Postscript format is optional.

You **MUST** respect standard RFC keywords as described in RFC 2119.

You **MAY** choose to write the documentation in English or in French. You are strongly invited to write the documentation in English.

If you choose the French version, the keywords of the RFC 2119 **MUST** be replaced by "DOIT", "NE DOIT PAS", "OBLIGATOIRE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS", "RECOMMANDE", "PEUT", "OPTIONNEL".

+ Language

Only C++ is allowed. Neither C nor "C+" will be tolerated: you will lose a **LOT** of points.

Pay close attention to your use of `const`, references, etc…

+ Libraries

You **MUST** use the SFML on the client side.

You **MAY** use Boost and/or Qt libraries for the client side ONLY. Please note that any rendering **MUST** be done using the SFML. As a consequence, any graphical resource from Qt is **FORBIDDEN**.

You ARE allowed to use Boost::ASIO for your server. Nevertheless, a home-made server will be considered a huge bonus.

+ GAME ENGINE

You've now been experimenting with C++ and Object-Oriented Design for a year. That experience means it should now be obvious for you to create **abstractions** and write **re-usable code**.

Therefore, before you begin work on your game, it is important that you start by creating a **game engine**!

The game engine is the core foundation of any video game: it determines how you represent an object in-game, how the coordinate system works, and how the various systems of your game (graphics, physics, network…) communicate.

Having an effective game engine is almost more important than having a fun game: without a decent engine, it will quickly be impossible to extend or improve your game. We recommend you take **AT LEAST** a day or two **ONLY DESIGNING** your engine. Once you've settled on a given design, implementing the engine is generally straightforward and shouldn't take more than a day.

When designing your engine, always question how extensible it is: after your **R-Type** is done, how easy would it be to write a **Starcraft** clone using your engine? A **Mario** clone? A **Battlefield** clone? Anything should be possible.

Your engine will be reviewed during the follow-ups and final defense, and we will focus on the following points:

- Runtime extensibility: the ability to add systems to a game through the form of dynamic libraries, scripts…
- Compile-time safety: if your engine provides no type-safety at compile-time, using it may lead to code that's unclear
- Ease of use: when encountering your engine for the first time, how easy is it to create a simple game like **Snake**?

+ GENERAL

The client **MUST** display a slow horizontal scrolling background representing space with stars, planets… This is the "starfield".

The starfield scrolling must NOT be tied to the CPU speed. Instead, you **MUST** use timers.

Players **MUST** be able to move using the arrow keys.

The server **MUST** be multi-threaded.

If a client crashes for any reason, the server **MUST** continue to work and **MUST** notify other clients in the

same game that a client crashed.

R-Type sprites are freely available on the Internet, but a set of sprites is available with this subject for lazy students.

The four players in a game **MUST** be blue, red, yellow and green, respectively.

There **MUST** be Bydos slaves in your game.

- Each kind of monster **MUST** be a dynamic library that can be loaded by your server **without having to restart it**.
- You **MUST** write your own API to interact with those libraries.
- Monsters **MUST** be able to spawn randomly on the right of the screen.
- The server **MUST** notify each client when a monster spawns, is destroyed, fires, kills a player, and so on…

This is the minimum, you **MUST** add anything you feel will get your game closer to the original.

Timeline

+ Design follow-up

You **MUST** provide an RFC for your protocol.

You **MUST** provide a UML class diagram for both client AND server. This diagram **MAY** be split into two sub-diagrams. It **MUST** be printed and readable.

You **MUST** provide a UML sequence diagram for both client AND server. This diagram **MAY** be split into two sub-diagrams. It **MUST** be printed and readable.

+ Implementation follow-up

You **MUST** have a fully functional server.

It **MUST** be possible to connect any number of graphic terminals (up to 4 per game instance).

It **MUST** be possible to move players using a graphic terminal. The movement **MUST** be dispatched to all other game clients.

Collisions between players **MUST** be detected. This feature is only mandatory for this follow-up, as a proof of concept. An **R-Type** without friendly collisions will be accepted during the final defense.

+ Final defense

You **MUST** have both a client and a server, fully functional, in accordance with the above **Requirements**.

Your implementation **MUST** match your initial design, or you WILL lose points.

You **MUST** bring all your documentation, including UML diagrams and RFC protocol.

The teachers **MUST** enjoy playing your game!

Some advice

As I promised, **R-Type** is a very fun but difficult project. Give the project everything you have, and work hard! It will be impossible otherwise.

As R-Type is a video game, I strongly recommend you to take contact with local experts (you're in a computer science school, one of your teachers must specialize in video games) to ask any technical question about video games.
When designing your game engine, **decoupling** is the most important thing you should focus on. The graphics system of your game only needs an entity's appearance and position to render it. It doesn't need to know about how much damage it can deal or the speed at which it can move! Similarly, a physics system doesn't need to know what an entity looks like to update its position. Think of the best ways to decouple the various systems in your engine.

To do so, we recommend taking a look at the Entity-Component-System **architectural pattern**, as well as the **Mediator** design pattern. But there are many other ways to implement a game engine! Be creative!

General setpoints

You are (more or less) free to implement the client and server any way you please. However, here are a few restrictions:

- The only authorized functions from the **libc** are the ones that wrap system calls (and don't have C++ equivalents!)
- Any solution to a problem **MUST** be object-oriented.
- Any not explicitly authorized library is explicitly forbidden.
- Any value passed by copy instead of reference or pointer MUST be justified, or you'll lose points.
- Any member function or method that does not modify the current instance not **const** MUST be justified, or you'll lose points.
- Koalas don't use any **C++** norm. However, any code that is deemed unreadable, unmaintainable or with unnecessary performance costs **WILL** be arbitrarily sanctioned. Be rigorous! Write code you'll be proud of!
- Any conditional branching longer than **if … else if … else …** is FORBIDDEN. Factorize! Use the STL's **associative containers**.
- Keep an eye on this subject regularly, it could be modified.
- We pay great attention to our subjects. If you run into typos, spelling mistakes or inconsistencies, please contact us so we can correct it.
- You can contact the authors by mail. Their adresses can be found on the module's page, under "Module Designers"
- The **C++** Yammer group will contain information and answers to your questions. Please make sure the answer to your question can't be found there before contacting the authors.