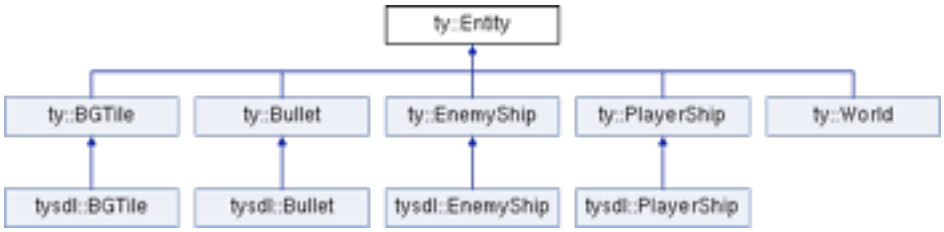
C++ Project 2014-2015

Goal of this project:

Write a Tyrian (2000) - like game in C++ using SFML and (try to) follow these guidelines:

Game logic library:

- Provide a clear **separation between game logic and game presentation**. Do this by encapsulating all game objects and logic, except their presentation and user-interaction (i.e. keyboard), in a self-contained (static or shared) **library**. (Look at C++ “gobelijn” code for examples of creating libraries. CMake can easily create static & shared libraries) The goal of this separation is to provide a very simple way of writing a completely new visual presentation based on the same game logic and structure. This way you might have two completely different looking games, based on the same game library. Use a separate namespace (f.i.: **ty**) for this library.
- Design a hierarchy of game entities (ships, bullets, decoration, ...) and their interactions (collision control, position control, ...). Be as creative & fancy as you like. A basic entity hierarchy might be, for instance, implemented as in this figure:



  Some suggestions:

  - Provide an entity that represents the game world (**ty::World**). The world contains a list of child entities (cfr: **composition** design pattern) and delegates visualization requests to its children. The world might also do other game related things such as collision control.
  - It “might” be convenient for entities to have a pointer to their parent.
  - Use one class to represent **the game**: an object that interacts with the world to keep the game running.
  - Classes in the **ty** namespace **must not** contain any calls to visualization libraries, interactive keyboard control, etc...
  - Use the **abstract factory** pattern to create entities (such as bullets or enemies) throughout the game. Pass the creating factory to every entity it creates. As such, entities will know how to create children.
  - Create two singleton classes: a **Keyboard**, for recording user input and a basic **Stopwatch** class for keeping time (in milliseconds) between two “ticks”. (Because not every computer has the same speed and yet your entities should move with the same speed on all computers)

- The game field is a 2D space: [-4,4] x [-3,3]. This is the visible game world (= the screen). The game simulates a space ship flying through some space as seen from the top. While the background (f.i.: a collection of BGTiles in my case) slides from the top to the bottom of the screen at a constant rate the player can control its ship to move left/right and accelerate/slow down. Opportunities to fire bullets present themselves as soon as enemy space ships appear on screen.
- Implement **multiple levels** and **load those levels from a file** (for instance, use a library which you need to include in your final project, or use the BOOST library).
- Be creative but implement at least the basic entities found in the figure above. Make it work first; then make it fancy. **Not** the other way around.

- Don't focus too much on enemy AI, simple movements & random shooting are easy to implement.
- Use rudimentary collision control; you can assume all objects are spherical and check for collision based on their position and radius.

Visual & interactive implementation:

- Use the SFML library (http://www.sfml-dev.org/) in your **visual** and **interactive** implementation of game's objects. Also in this case, use a separate namespace to encapsulate SFML implementations of your game objects (f.i.: **tysfml**).
- In your visual implementation, you can use your own images for the entities or you can find some online. See for instance: http://www.lostgarden.com/2007/04/free-game-graphics-tyrian-ships-and.html (They're really good!)
- If you use graphics from the remastered Tyrian I suggest using a rather low screen resolution such as 320 x 240.
- Everything put together might look similar to:

Practical:

- A basic **working** implementation of the library with basic game logic (and more importantly: following clear design considerations) & visual representation in SFML are sufficient to get a passing grade.
- Apply proper **code commenting & documentation** of your API. This is obviously implied to get a passing grade.
- Apply a l**ogical structure of the code & code base** (source files in logically structured directories, do not throw everything in one huge folder), proper class construction that proves you understand C++. This is obviously implied to get a passing grade.
- Use **CMAKE** as your build system.
- Implement features **incrementally**! Don't write hundreds of lines of code hoping it will magically compile and work at the end.
- Extensions are **up to you**:
    - Different types of enemy ships.

    - Different types of cannons with varying power.
    - …

- **Attention**: projects that do not compile or work (e.g., compiling error or segmentation faults when starting the application) automatically imply “student **failed** this part”. The computers in G026 are used as the reference platform: your code should compile and work on those computers.
- The project has to be made and handed in **individually**. Of course, you can discuss design/ problems/solutions with other students as much as you like.
- Describe your design and some of the choices you made in **a report** (ca. 2 A4 pages). This report has to reflect that you really thought about the choices you made for your design. If necessary, you can attach some UML diagrams to illustrate your design.
- Good luck! If you have any questions, remarks or comments: glenn.daneels@uantwerpen.be przemyslaw.klosiewicz@uantwerpen.be or drop by @ G212/G207.
- Deadline: To be announced. (Somewhere in **January 2015**)
- Submit the final project and report **on Blackboard and by mail**.