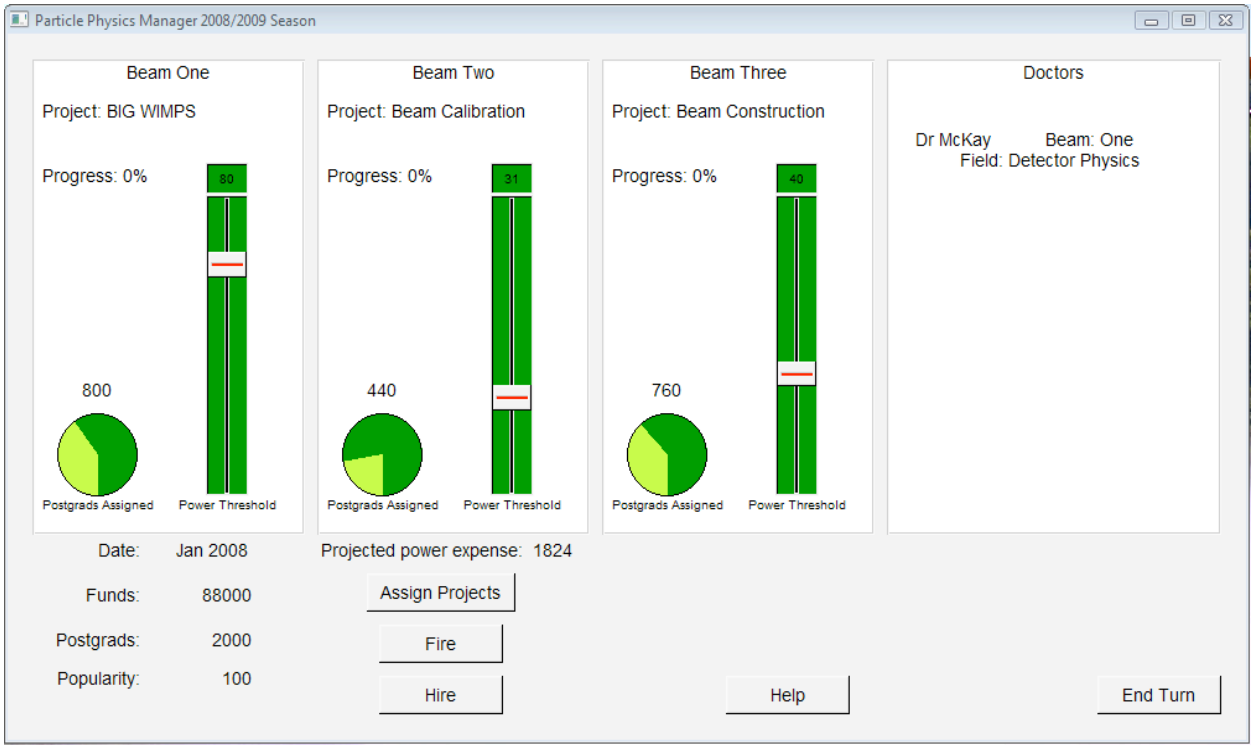
C++ Laboratory Report
'Particle Physics Manager 08/09 Season'

Aim

In this project I attempted to create a user interface driven game using the C++ language and the Fast Light Toolkit (FLTK) of my own choosing. The program was based on the idea of turn-based resource management strategy simulations as applied to particle physics research. As such, it casts the player as Director General of a new large-scale experiment and charges them with selecting their research projects and the scientists who will undertaken them. However, the game only took it's theme from research science and I did not try to replicate it or teach the user any solid physics.

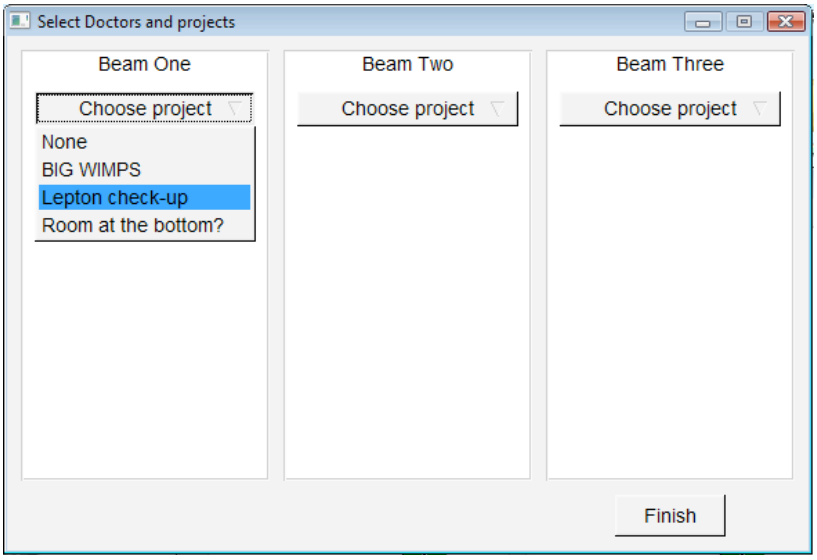
I chose this project as the idea of crafting both a functioning interface and interesting game design appealed to me. Being a turn-based program, it's operation is similar to a calculator in that a user can set a range of variables before pressing an 'equals' button that computes their actions and displays the results, a task to which FLTK is suited.

Brief Operation Guide

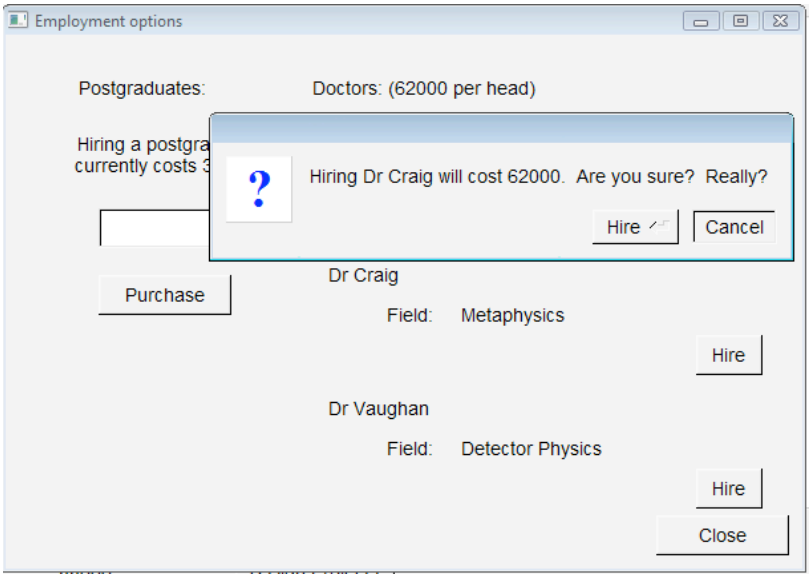


A reasonably thorough explanation of core game concepts can be called from the 'Help' button from the main display. This acts as an ingame manual and is accessible at all points. An average turn consists of a player choosing whether to hire new postgraduates or doctors from their limited budget then assigning those employees to each of their three 'beams' which hold and a particular project that may itself be selected. The available projects are determined by the projects already completed as well as the specialist areas of the doctors assigned to a specific beam and the progress on a project is calculated every turn based on the number of postgraduate workers and units of power assigned to a project. The cost per unit power is proportional to the square of units allocated, so a small amount of power may easily be obtained whilst rushing several projects on has a large cost associated with it, especially as the effect per unit power is

linear.



This is the most obvious mechanic for introducing a strategy element to the player's choices, as it is stacked against the monthly budget. This budget decays every turn as public interest wanes but is recovered upon the completion of a project. As such, the player must decide whether to invest a small amount of power and hope that reducing public interest will sufficiently fund the project or else 'buy' the completion with increased



power usage, keeping public interest high.

In a full play through, a sufficient amount of research will eventually be reached to uncover the Higgs boson.

An early version of the code was shown to a few unprepared players. From there observations several usability issues were uncovered. In particular, the tooltips initially used to teach the player were insufficient, so an automatic pop-up was added to the beginning of the program with an overview of the game's concepts as well as pointing the user to the 'Help' button for more guidance.

I also noted that players didn't like exploring their options on beams 2 or 3, so I set those to automatically begin with their 'building project' to guide the player (beam 3 must be 'built' and 'calibrated' before it can undertake standard projects, whilst beam 2 only

needs to be 'calibrated').

Architecture

Most of the player's data in a game is stored in a pair of vectors of custom classes. Vector<Doc> doctors stores all the doctors that a player may have hired, including their specialised field and the beam to which they are assigned. Vector<tech> projects stores every project available to the player (having read it in from the local file 'projects.dat') as well as progress and completion status. Both of these classes were designed to make handling of these game concepts simpler and included more items than were eventually used, Doc having a set of potential skills available (now removed) and tech including a canRepeat flag that I decided not to use when determining a group of projects for the first iteration of the game. Furthermore, these classes are largely 'dull', in that they simply input and output their private values, with the exception of the "string getXyz" functions, which calls an appropriate string related to the a single integer, and getDrName, which conveniently returns "Dr *stringName*".

A separate header file was briefly considered, but as that achieves nothing that prior declaration can (that is, there is no interdependency of the objects and they are only used by one program), they were left in the single source code file.

In addition to the two vectors, an array of 3 ints tracks which project, as a position in vector<tech> projects, is being research at any given moment. As a fringe benefit of this design, a project may be begun and halted without losing any earned 'progress' on it, as it remains stored in the vector for the next call of that item.

The code itself is of two largely distinct parts; callback functions that open a new window for a player to alter some value and buttons that effect the changes selected. Where possible, repeated chunks of code were compacted into a separate function and similar functions were condensed into a general purpose function who, with the correct inputs on call could recreate the behaviour of each prior function.

The stringinate() function is a good example of the former type, where I was using a trick I had seen on www.cplusplus.com to put integer values onto a stringstream and reading them back off it into a string value, effectively casting from int to string. As this was frequently occurring, it made sense to migrate it from within the getDate(int month) function and use it on it's own.

hide_win_cb(Fl_Widget* w, void *) is a good example of the latter case, where approximately four similar window hide callbacks were replaced by one general one that took the void pointer in as an Fl_Window* via an implicit cast and hide that. This also shows my improving grip on coding concepts. Over the course of designing my final code, by overcoming challenges and finding solutions I learnt several things that, had I known earlier, would have dramatically changed my code structure. For example, I could have conceivably created a struct to pass more data through the void* of a FLTK callback and reduced the number of items globally declared.

It is worth noting that my largest function, aside from the main() which defines several objects and defines their state, is project_selectpt2_cb(Fl_Widget* w, void* button), the second stage of selecting a project. This is require to establish the research levels in five areas based on projects completed and the locations of the Docs as well as determine which projects are valid for inclusion in the drop down menu, discarding any finished projects, checking that research level requirements are met or exceeded and ensure that

beams 2 & 3 are 'calibrated' and beam 3 is 'built'. Were it not for the last condition, a reasonable chunk of the code – project to drop menu assignment – could be extracted as a function corresponding to each beam. As it is, there is no space gain in extracting any significant part of it and it successfully encodes a good number of game mechanics (IE, variable number of doctors who add to a research are on a specific beam, beams that don't start 'ready', projects available determined by previous progress).

Evaluation

The program is works functionally – that is, it acts as a structure within which a game takes place. However, in it's present form the game fails to be fun, partly because it is frustrating to newcomers and partly because there is no balanced challenge. In order to make this an enjoyable single player experience, the values of each project would require tuning with respect to the weight and value of the main resources. This would require repeated play testing to achieve and the complete success of this is beyond the scope of my project. The code is submitted with my testing projects.dat file, which although smaller than the length of game I envision is much more conducive to shorter runs through the game to identify balance.

As an example, the number of postgraduates has been increased in a standard game and the effectiveness and value greatly decreased. This contrasts well with the strictly limited number of doctors, making the postgrads (who thanklessly do the actual work in the game) appear worthless numbers to be applied to a problem and making the Docs feel more elite.

Additionally, the cost of power was previously linear before testing showed that there was no reason not to buy as much power as possible. This was rectified as described in the brief operations guide.

An alternative to balancing the game for one player would be to make it into a race to discover Higgs between two players. This could potentially be achieved with little structural change by making some class composed of all of one player's data (vector<doc> doctors, etc) and serialising it, then loading/saving it between turns.

As such, I am very pleased with the technical aspects of my code, but feel that the mechanics are too obtuse to be immediately entertaining.

References

For the most part, new techniques and understanding were taken from the documentation included in Quincy (including it's FLTK manual), discussion with fellow students and demonstrators and careful reading of cplusplus.com for it's comprehensive explanation of the standard C++ libraries. Where code is based on sources outside of these or else barely modified, it is noted in the code itself.