TIE-20106 Data structures and algorithms, spring 2019

Programming assignment 1: Beacons of RGB

Last modified on 04/05/2019

Topic of the assignment

In the mystical world of Cmyk the Ancients have built beacons, which for an unknown reason do not rotate, but send their light to another beacon, which combines its own light to the received light. Each beacon has a light color, (x,y) position and name. The first programming assignment is to write a class, which handles beacons and their lightbeams.

The idea of this assignment is to practice the use of ready-made data structures and algorithms (STL), but it also practicing writing own algorithms and analyzing their performance.

The beacons are now identified with a unique ID instead of pointers. The end result is a program which records information about beacons (name, coordinates, and color), and beacons can be printed out in different orders, and minimum and maximum queries can be made. As a non-compulsory part, beacons can also be removed. Operations exist for asking the lightbeam relationships of the beacons, as well as the total color they emit.

Since this is a Data structures and algorithms assignment, performance of the program is an important grading criteria. The goal is to code an as efficient as possible implementation, under the assumption that all operations are executed equally often (unless specified otherwise on the command table). Getting better performance helps to increase the grade. Especially note the following:

- In this assignment efficient use of ready-made data structures and algorithms (STL) is one goal, so it's a good idea to prefer STL over self-written algorithms/data structures (when that's reasonable from performance point of view).
- The main program given by the course can be run either with a graphical user interface (when compiled with QtCreator/qmake), or as a textual command line program (when compiled just with g++ or some other C++ compiler). In both cases the basic functionality and students' implementation is exactly the same.
- **Hint** about (not) suitable performance: If the performance of any of your operations is worse than $\Theta(n \log n)$ on average, the performance is definitely *not* ok. Most operations can be implemented much faster.
- **As part of the assignment, file datastructures.hh contains a comment next to each operation. Into that comment you should write your estimate of the asymptotic performance of the operation, which a short rationale for you estimate.**

- **As part of the assignment submission, a document should be added to git (in the same directory/folder as the code). This document should contain reasons for choosing the data structures used in the assignment (especially performance reasons). Acceptable formast are plain test (readme.txt), markdown (readme.md), and Pdf (readme.pdf).**
- Implementing operations `remove_beacon(), path_inbeam_longest(),` and `total_color()` is not compulsory to pass the assignment. **If you only implement the compulsory parts, the maximum grade for the assignment is 3.**
- In performance the essential thing is how the execution time changes with more data, not just the measured seconds.
- The performance of an operation includes all work done for the operation, also work possibly done during addition of elements.
- More points are given if operations are implemented with better performance.
- Likewise more points will be given for better real performance in seconds (if the required minimum asymptotic performance is met). **But** points are only given for performance that comes from algorithmic choices and design of the program (for example setting compiler optimization flags, using concurrency or hacker optimizing individual C++ lines doesn't give extra points).
- If the implementation is bad enough, the assignment can be rejected.
- Examples of questions, which may help in improving the performance: Is somewhere the same thing re-done several times? Can you sometimes avoid doing something completely? Does some part of the code do more work than is absolutely necessary? Can some things be done "almost free" while doing something else?

On calculating colors

In this assignment beacons send lightbeams to other beacons, which combine received light with their own light to an outgoing lightbeam. Combining the lightbeams is done simply by calculating the averages of each color component (r, g, and b) for the received lightbeams and beacon's own color. The average is calculated by first summing the color components, and then dividing by the number of components, and rounding down (i.e., the default rounding in C++ integer arithmetic).

Sorting involves calculating the brightness of a color. The brightness is calculated with formula $3*r+6*g+b$ (this is an approximation of how human eye perceives the brightness of a color in sRGB color space).

On sorting

While sorting beacons in alphabetical order it's possible that several beacons have the same name (or the same brightness when sorting according to brightness). The mutual ordering of such beacons doesn't matter (they are allowed to be in any order with respect to each other).

The main program only accepts names consisting of letters A-Z and a-z. The alphabetical sorting can be done either with the regular < comparison of C++ string class (in which case upper case letters come first, followed by lower case letters) or the "correct way", where upper and lower case letters are equal.

When sorting beacon IDs, C++ string comparison "<" should be used.

About implementing the program and using C++

The programming language in this assignment is C++17. The purpose of this programming assignment is to learn how to use ready-made data structures and algorithms, so using C++ STL is very recommended and part of the grading. There are no restrictions in using C++ standard library. Naturally using libraries not part of the language is not allowed (for example libraries provided by Windows, etc.). *Please note however, that it's very likely you'll have to implement some algorithms completely by yourself.*

Structure and functionality of the program.

Part of the program code is provided by the course, part has to be implemented by students.

Parts provided by the course

Files *mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui* (you are **NOT ALLOWED TO MAKE ANY CHANGES TO THESE FILES)**

- Main routine, which takes care of reading input, interpreting commands and printing out results. The main routine contains also commands for testing.
- If you compile the program with QtCreator or qmake, you'll get a graphical user interface, with a embedded command interpreter and buttons for pasting commands, file names, etc in addition to keyboard input. The graphical user interface also shows a visual representation of beacons, their lightbeams, results of operations, etc.

File *datastructures.hh*

- `class Datastructures`: The implementation of this class is where students write their code. The public interface of the class is provided by the course. You are **NOT ALLOWED TO CHANGE THE PUBLIC INTERFACE** (i.e. change names, return type or parameters of the given public member functions).
- Type definition `BeaconID`, which used as a unique identifier for each beacon (and which is used as a return type for many operations). There can be several beacons with the same name (and even same coordinates), but every beacon has a different id.
- Type definition `Coord`, which is used in the public interface to represent (x,y) coordinates. As an example, some comparison operations (==, !=, <) and a hash function has been implemented for this type.
- Type definition `Color`, which is used in the public interface to represent (r,g,b) colors. As an example, some comparison operations (==, !=) has been implemented for this type (you can implement more, if you need).

- Constants `NO_ID`, `NO_NAME`, `NO_COORD`, and `NO_VALUE`, which are used as return values, if information is requested for a beacon that doesn't exist.

File *datastructures.cc*

- Here you write the code for the your operations.
- Function `random_in_range`: Like in the first assignment, returns a random value in given range (start and end of the range are included in the range). You can use this function if your implementation requires random numbers.

On using the graphical user interface

When compiled with QtCreator, a graphical user interface is provided. It allows running operations and test scripts, and visualize the data.

The UI has a command line interface, which accepts commands described later in this document. In addition to this, the UI shows graphically created beacons and their lightbeams. The graphical view can be scrolled and zoomed. Clicking on a beacon prints out its information, and also inserts the beacon's ID on the command line (a handy way to give commands ID parameters). The user interface has selections for whether beacon names, lightbeam, and their colors are shown graphically.

Note! The graphical representation gets all its information from student code! It's not a depiction of what the "right" result is, but what information students' code gives out. The UI uses operation `all_beacons`() to get a list of beacons, and asks their information with `get_...`() operations. If drawing lightbeams is on, they are get with operation `get_lightsources`(), and if lightbeam coloring is on, the colors are get with operation `total_color`() (non-compulsory).

At this point the UI also has a couple of disabled controls, they will become relevant in the second assignment.

Parts of the program to be implemented as the assignment

Files *datastructure.hpp* and *datastructure.cpp*

- `class Datastructure`: The given public member functions of the class have to be implemented. You can add your own stuff into the class (new data members, new member functions, etc.)
- **In file *datastructures.hh*, for each member function you implement, write an estimation of the asymptotic performance of the operation (with a short rationale for your estimate) as a comment above the member function declaration.**

Additionally the readme.pdf mentioned before is written as a part of the assignment.

Note! The code implemented by students should not do any output related to the expected functionality, the main program does that. If you want to do debug-output while testing, use the `cerr` stream instead of `cout` (or `qDebug`, if you you use Qt), so that debug output does not interfere with the tests.

Commands recognized by the program and the public interface of the Datastructures class

When the program is run, it waits for commands explained below. The commands, whose explanation mentions a member function, call the respective member function of the Datastructure class (implemented by students). Some commands are completely implemented by the code provided by the course.

If the program is given a file as a command line parameter, the program executes commands from that file and then quits.

The operations below are listed in the order in which we recommend they to be implemented (of course you should first design the class taking into account all operations).

Command Public member function	Explanation

`beacon_count`
`int beacon_count()`
	Returns the number of beacons currently in the data structure.

`clear_beacons`
`void clear_beacons()`
	Clears out the data structure, i.e. removes all beacons and lightbeams (after this size returns 0).

`all_beacons`
`std::vector<BeaconID> all_beacons()`
	Returns a list (vector) of the beacons in any (arbitrary) order. *This operation is not included in the default performance tests.*

`add_beacon ID Name (x,y) (r,g,b)`
`bool add_beacon(BeaconID id, std::string const& name, Coord xy, Color color)`
	Adds a beacon to the data structure with given unique id, name, coordinates, and color. New added beacons are initially not emitting a lightbeam anywhere. If there already is a beacon with the given id, nothing is done and `false` is returned, otherwise `true` is returned.

`(no command)`
`std::string get_name(BeaconID id)`
	Returns the name of the beacon with given ID, or NO_NAME if such beacon doesn't exist. (Main program calls this in various places.) *This operation is called more often than others.*

`(no command)`
`Coord get_coordinates(BeaconID id)`
	Returns the name of the beacon with given ID, or value NO_COORD, if such beacon doesn't exist. (Main program calls this in various places.) *This operation is called more often than others.*

`(no command)`
`Color get_color(BeaconID id)`
	Returns the own color of the beacon with given ID, or NO_COLOR if such beacon doesn't exist. (Main program calls this in various places.) *This operation is called more often than others.*

(The operations below should probably be implemented only after the ones above have been implemented.)	

`sort_alpha`
`std::vector<BeaconID> beacons_alphabetically()`
	Returns beacon IDs sorted according to alphabetical order of beacon names.

Command Public member function	Explanation

`sort_brightness`
`std::vector<BeaconID> beacons_brightness_increasing()`
	Returns beacon IDs sorted according to increasing brightness of becons' own colors. *Note the definition of "brightness" earlier in this document.*

`min_brightness`
`BeaconID min_brightness()`
	Returns the beacon with dimmest brightness based on beacon's own color. If there are several such beacons with the same brightness, returns one of them. If no beacons exist, NO_ID is returned. *Note the definition of "brightness" earlier in this document.*

`max_brightness`
`BeaconID max_brightness()`
	Returns the beacon with brightest brightness based on beacon's own color. If there are several such beacons with the same brightness, returns one of them. If no beacons exist, NO_ID is returned. *Note the definition of "brightness" earlier in this document.*

`find_beacons name`
`std::vector<BeaconID> find_beacons(std::string const& name)`
	Returns beacons with the given name, or an empty vector, if no such beacons exist. The return value must be sorted in increasing order of IDs. *The performance of this operation is not critical (it is not expected to be called often), so it's not part of default performance tests.*

`change_name ID newname`
`bool change_beacon_name(BeaconID id, std::string const& newname)`
	Changes the name of the beacon with given ID. If such beacon doesn't exist, returns `false`, otherwise `true`.

`change_color ID (r,g,b)`
`bool change_beacon_color(BeaconID id, Color newcolor)`
	Changes the color of the beacon with given ID. If such beacon doesn't exist, returns `false`, otherwise `true`.

(The operations below should probably be implemented only after the ones above have been implemented.)	

`add_lightbeam SourceID TargetID`
`bool add_lightbeam(BeaconID sourceid, BeaconID targetid)`
	Adds a lightbeam from one beacon to another. A beacon can only send a lightbeam to one beacon. You can assume that lightbeams do not form cycles (i.e., a beacon cannot directly or indirectly receive its own light). If either of the beacons do not exist, or the source beacon already send a lightbeam, nothing is done and `false` is returned. Otherwise `true` is returned.

`lightsources BeaconID`
`std::vector<BeaconID> get_lightsources(BeaconID id)`
	Returns the IDs of beacons that directly send a ligthbeam to a given beacon, or a vector with a single element NO_ID, if no beacon with given ID exists. The return value must be sorted in increasing order of IDs. (Main program calls this in various places.)

Command Public member function	Explanation

`path_outbeam`
`std::vector<BeaconID> path_outbeam(BeaconID id)`
	Returns a list of beacons to which the given beacon's light end up either directly or indirectly. The return vector first contains the beacon itself, then the beacon it sends its lightbeam to, the lightbeam target of that beacon, etc. as long as there are lightbeams. If no beacon with given ID exists, a vector with a single element NO_ID is returned.

(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	

`remove_beacon ID`
`bool remove_beacon(BeaconID id)`
	Removes a beacon with the given id. If a beacon with given id does not exist, does nothing and returns `false`, otherwise returns `true`. If the beacon to be removed sends or receives lightbeams, those lightbeams are also removed. *The performance of this operation is not critical (it is not expected to be called often), so it's not part of default performance tests. Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

`path_inbeam_longest ID`
`std::vector<BeaconID> path_inbeam_longest(BeaconID id)`
	Returns the longest possible chain of lightbeams, that end up in the given beacon. The return contains beacons so that the previous beacon sends its lightbeam to the next, and the last beacon is the beacon given as the parameter (if there are several possible equally long lightbeam chains, anyone of them is returned). If the ID does not correspond to any beacon, a vector with a single element NO_ID is returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment).*

`total_color ID`
`Color total_color(BeaconID id)`
	Returns the total color of the beacon's outgoing lightbeam, i.e. the average of its own color and received lightbeams (more detailed explanation earlier in this document). If the ID does not correspond to any beacon, NO_COLOR is returned. *Implementing this command is not compulsory (but is taken into account in the grading of the assignment). This operation is called more often than others.*

(The following operations are already implemented by the main program.)	
random_add n (implemented by main program)	Add *n* new beacons with random id, name, coordinates, and color (for testing). With a 80 % probability a lightbeam is also added to another beacon. Note! The values really are random, so they can be different for each run.

`Command`
`Public member function`
	Explanation

random_seed n (implemented by main program)	Sets a new seed to the main program's random number generator. By default the generator is initialized to a different value each time the program is run, i.e. random data is different from one run to another. By setting the seed you can get the random data to stay same between runs (can be useful in debugging).
read 'filename' (implemented by main program)	Reads more commands from the given file (This can be used to read a list of beacons from a file, run tests, etc.)
stopwatch on / off / next (implemented by main program)	Switch time measurement on or off. When program starts, measurement is "off". When it is turned "on", the time it takes to execute each command is printed after the command. Option "next" switches the measurement on only for the next command (handy with command "read" to measure the total time of a command file).
perftest all/compulsory/cmd1;cmd2... timeout n n1;n2;n3... (implemented by main program)	Run performance tests. Clears out the data structure and add *n1* random beacons (see random_add). Then a random command is performed *n* times. The time for adding elements and running commands is measured and printed out. Then the same is repeated for *n2* employees, etc. If any test round takes more than *timeout* seconds, the test is interrupted (this is not necessarily a failure, just arbitrary time limit). If the first parameter of the command is *all*, commands are selected from all commands. If it is *compulsory*, random commands are selected only from operations that have to be implemented. If the parameter is a list of commands, commands are selected from that list (in this case it's a good idea to include also random_add so that elements are also added during the test loop). If the program is run with a graphical user interface, "stop test" button can be used to interrupt the performance test (it may take a while for the program to react to the button).
testread 'infilename' 'outfilename' (implemented by main program)	Runs a correctness test and compares results. Reads command from file infilename and shows the output of the commands next to the expected output in file outfilename. Each line with differences is marked with a question mark. Finally the last line tells whether there are any differences.
help (implemented by main program)	Prints out a list of known commands.
quit (implemented by main program)	Quit the program. (If this is read from a file, stops processing that file.)

"Data files"

The easiest way to test the program is to create "data files", which can add a bunch of beacons and lightbeams. Those files can then be read in using the "read" command, after which other commands can be tested without having to enter those beacons every time by hand.

Below are examples of a data files, one of which adds beacons, the other lightbeams:

- *example-beacons.txt*

```
# Add beacons
add_beacon G1 Lime (0,0) (0,255,0)
add_beacon M1 Fuchsia (5,0) (255,0,255)
add_beacon R1 Crimson (0,5) (220,20,60)
add_beacon B2 Teal (10, 10) (0,128,128)
add_beacon M2 Indigo (10, 5) (75,0,130)
```

- *example-lightbeams.txt*

```
# Add light sources
add_lightbeam G1 M1
add_lightbeam R1 M2
add_lightbeam M1 M2
add_lightbeam M2 B2
```

Example run

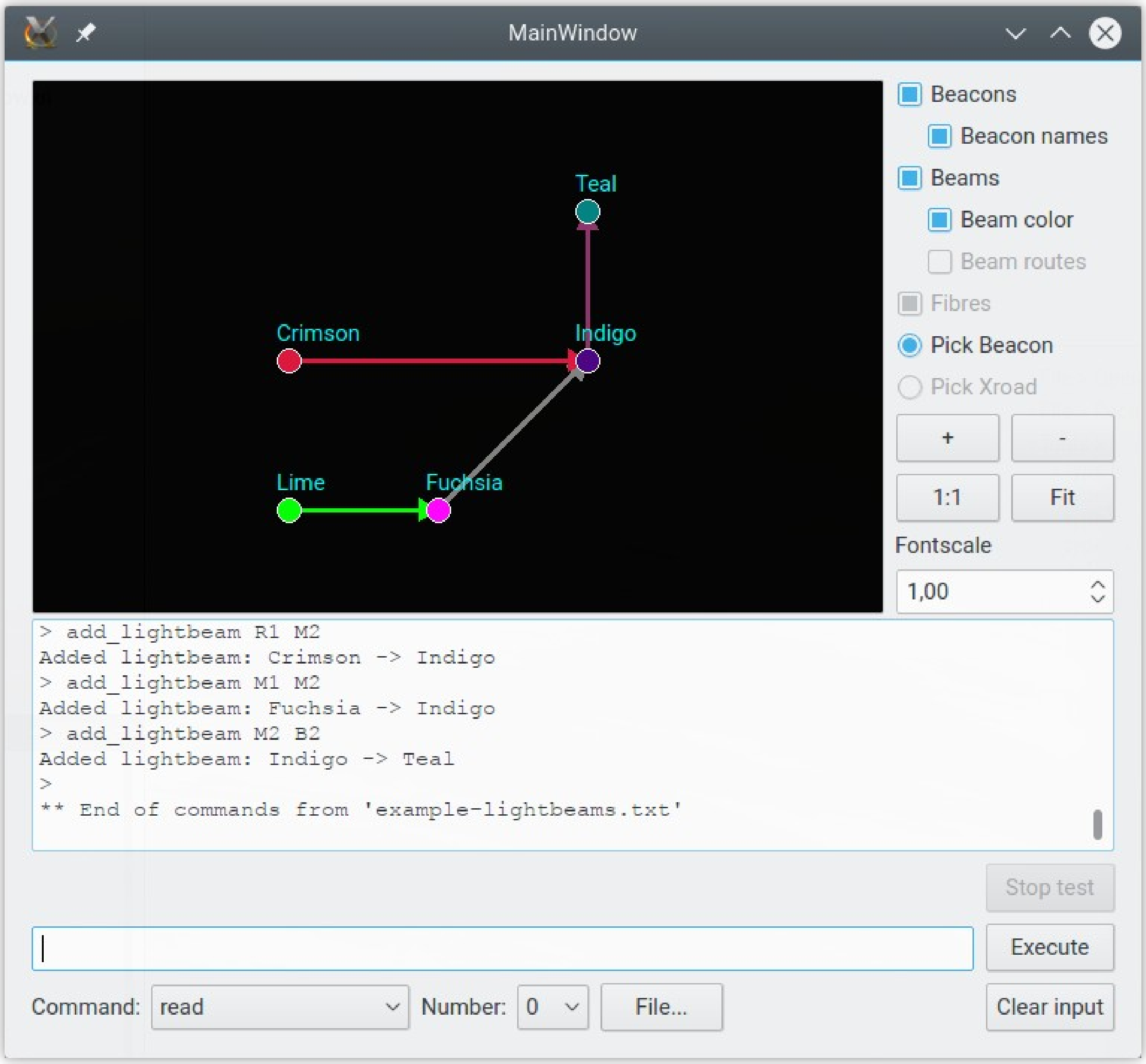
Below are example outputs from the program. The example's commands can be found in files *example-compulsory-in.txt* and *example-all-in.txt*, and the outputs in files *example-compulsory-out.txt* and *example-all-out.txt.* I.e., you can use the example as a small test of compulsory behaviour by running command
`testread "example-compulsory-in.txt" "example-compulsory-out.txt"`

```
> read "example-compulsory-in.txt"
** Commands from 'example-compulsory-in.txt'
> clear_beacons
Cleared all beacons
> beacon_count
Number of beacons: 0
> read "example-beacons.txt"
** Commands from 'example-beacons.txt'
> # Add beacons
> add_beacon G1 Lime (0,0) (0,255,0)
Lime: pos=(0,0), color=(0,255,0):1530, id=G1
> add_beacon M1 Fuchsia (5,0) (255,0,255)
Fuchsia: pos=(5,0), color=(255,0,255):1020, id=M1
> add_beacon R1 Crimson (0,5) (220,20,60)
Crimson: pos=(0,5), color=(220,20,60):840, id=R1
> add_beacon B2 Teal (10, 10) (0,128,128)
Teal: pos=(10,10), color=(0,128,128):896, id=B2
> add_beacon M2 Indigo (10, 5) (75,0,130)
Indigo: pos=(10,5), color=(75,0,130):355, id=M2
>
** End of commands from 'example-beacons.txt'
> beacon_count
Number of beacons: 5
> sort_alpha
1. Crimson: pos=(0,5), color=(220,20,60):840, id=R1
```

```
2. Fuchsia: pos=(5,0), color=(255,0,255):1020, id=M1
3. Indigo: pos=(10,5), color=(75,0,130):355, id=M2
4. Lime: pos=(0,0), color=(0,255,0):1530, id=G1
5. Teal: pos=(10,10), color=(0,128,128):896, id=B2
> min_brightness
Indigo: pos=(10,5), color=(75,0,130):355, id=M2
> max_brightness
Lime: pos=(0,0), color=(0,255,0):1530, id=G1
> sort_brightness
1. Indigo: pos=(10,5), color=(75,0,130):355, id=M2
2. Crimson: pos=(0,5), color=(220,20,60):840, id=R1
3. Teal: pos=(10,10), color=(0,128,128):896, id=B2
4. Fuchsia: pos=(5,0), color=(255,0,255):1020, id=M1
5. Lime: pos=(0,0), color=(0,255,0):1530, id=G1
> change_name M1 Indigo
Indigo: pos=(5,0), color=(255,0,255):1020, id=M1
> find_beacons Indigo
1. Indigo: pos=(5,0), color=(255,0,255):1020, id=M1
2. Indigo: pos=(10,5), color=(75,0,130):355, id=M2
> read "example-lightbeams.txt"
** Commands from 'example-lightbeams.txt'
> # Add light sources
> add_lightbeam G1 M1
Added lightbeam: Lime -> Indigo
> add_lightbeam R1 M2
Added lightbeam: Crimson -> Indigo
> add_lightbeam M1 M2
Added lightbeam: Indigo -> Indigo
> add_lightbeam M2 B2
Added lightbeam: Indigo -> Teal
>
** End of commands from 'example-lightbeams.txt'
> lightsources M2
1. Indigo: pos=(5,0), color=(255,0,255):1020, id=M1
2. Crimson: pos=(0,5), color=(220,20,60):840, id=R1
> change_color M1 (83,0,143)
Indigo: pos=(5,0), color=(83,0,143):392, id=M1
> path_outbeam R1
0.    Crimson: pos=(0,5), color=(220,20,60):840, id=R1
1. -> Indigo: pos=(10,5), color=(75,0,130):355, id=M2
2. -> Teal: pos=(10,10), color=(0,128,128):896, id=B2
>
** End of commands from 'example-compulsory-in.txt'
> path_inbeam_longest B2
0.    Lime: pos=(0,0), color=(0,255,0):1530, id=G1
1. -> Indigo: pos=(5,0), color=(83,0,143):392, id=M1
2. -> Indigo: pos=(10,5), color=(75,0,130):355, id=M2
3. -> Teal: pos=(10,10), color=(0,128,128):896, id=B2
> total_color B2
Total color of Teal: (56,88,107)
> remove_beacon M2
Indigo removed.
> lightsources B2
No lightsources!
>
```

Screenshot of user interface

Below is a screenshot of the graphical user interface after *example-beacons.txt* and *example-lightbeams.txt* have been read in.



Fontscale: 1,00

```
> add_lightbeam R1 M2
Added lightbeam: Crimson -> Indigo
> add_lightbeam M1 M2
Added lightbeam: Fuchsia -> Indigo
> add_lightbeam M2 B2
Added lightbeam: Indigo -> Teal
>
** End of commands from 'example-lightbeams.txt'
```

Command: read Number: 0