Final Project

The final project of GAS course consists in implementing the Fortune algorithm to compute the Voronoi Diagram of a set of points.

The maximum grade is 36/30.

The algorithm must be implemented in C++. We give a Base Project in which you can find some basic features that are useful to successfully develop your project.

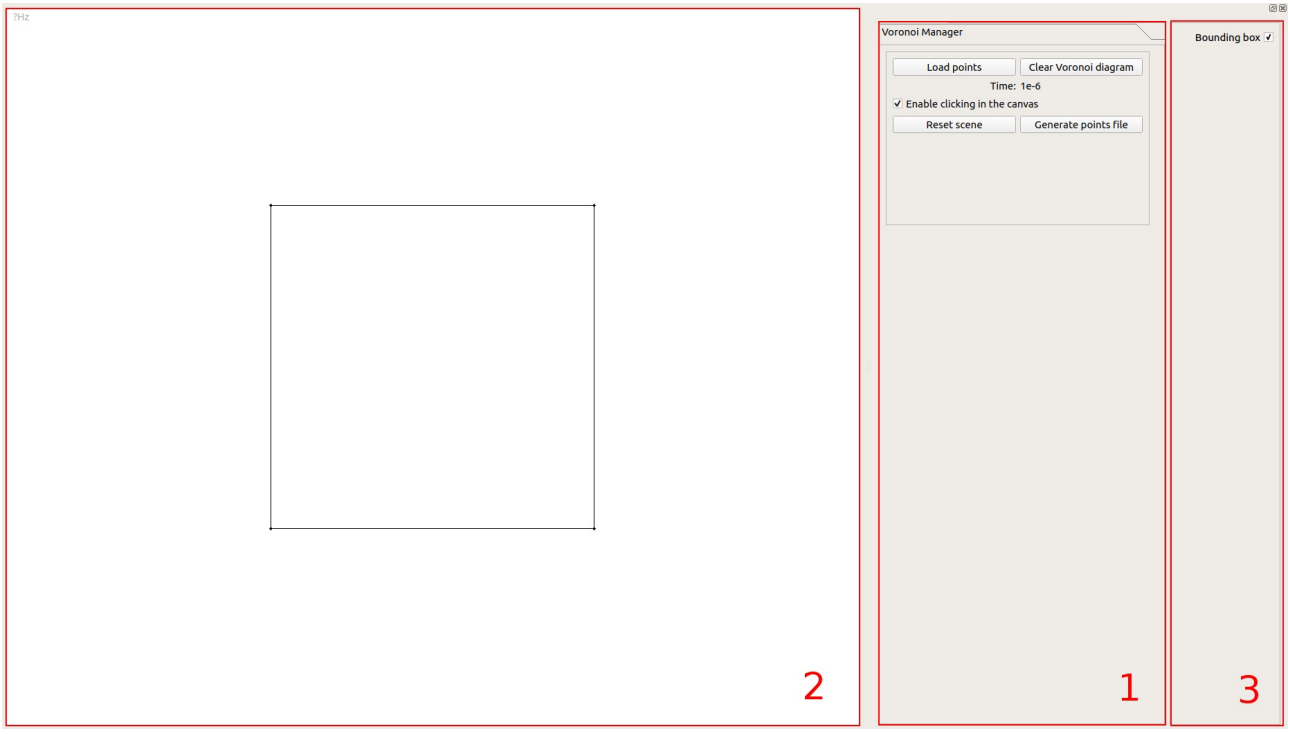
1. Base Project

1.1. Overview

The Base Project is composed by some files (in the main folder) and some modules of the cg3lib: the “core module” and the “viewer module”. The first module contains basic data structures and utilities that are useful for our purposes and the latter contains utilities for building the user interface and draw geometric primitives. Each module is organized with a .pri file which is included by the main .pro file of the project. **You must not modify the folders or the files in the folders that are associated to the cg3lib library**. All your files and folders (you should hyhe must be organized in the main folder of the project.

In the cg3lib folder of the project you can find the two folders of the modules “core” and “viewer”, and, in the main folder there are two folders “gui” and “utils”, and the main.cpp. The “gui” folder contains the VoronoiManager (the manager which has to implement all the requested features) and the “utils” folder contains some utility functions.

When you compile and run the project, the following window, the so-called “MainWindow” appears.



The MainWindow is composed of three parts:

1. Here you can find all the managers, GUI classes (QFrame) which contains all the tools for handling a set of operations that allows to manage some features and

objects. In our Base Project there is just the Voronoi Manager, which is the manager that must handle all functions and features of your Voronoi diagram generation. A bunch of buttons and checkboxes are already in the manager, and they are enough to give the interface to all the requested features. You can see how to add a manager to the MainWindow in the main.cpp file.

2. This is the GL Canvas. It is a Canvas that allows to draw objects with opengl calls. In this project, the MainWindow takes care to draw in the canvas all the "DrawableObjects" passed to it. In the picture we can see a 2D BoundingBox that has been drawn in the canvas. Clicking in the canvas will fire an event in which you can do something with the coordinates of the point that has been clicked.
3. Here you can find a set of checkboxes: every checkbox is linked to a DrawableObject that is inside the MainWindow. In the example picture, when you run the project there is an already implemented "Bounding Box" DrawableObject that is drawn in the GL Canvas.

1.2. *Drawable Objects*

How does the viewer manage to draw objects? The "MainWindow" gives you an interface to draw in the GLCanvas every object which implements the interface "DrawableObject" (viewer/drawable_objects/drawable_object.h). Every class that inherits the interface "DrawableObject" must implement the three pure abstract methods "draw()", "sceneCenter()", "sceneRadius()".

In "draw()," you have to implement the OpenGL code that draws the object. In "sceneCenter()" you have to return the point (a 3D point, cg3::Pointd) in the center of the object. In "sceneRadius()" you have to calculate the radius of the circle which entirely contains the object. These last 2 functions could be needed by the viewer to fit the scene to make all the objects in the canvas visible to the user. In the picture above you can see an example of a drawn 2D bounding box. You can find the code which implements the DrawableObject "DrawableBoundingBox2D" in the class file "cg3lib/cg3/viewer/drawable_objects/2d/drawable_bounding_box2d.h".

Note that you do not actually have to study OpenGL: indeed, you need only to draw points and lines to show a Voronoi diagram in the canvas.

In the viewer module there are the functions "drawPoint2D" and "drawLine2D": they allow you to draw in the canvas those primitives (the file is "cg3lib/cg3 /viewer/renderable_objects/2d/renderable_objects2d.h").

1.3. *Managers*

Let's take a deeper look at the Voronoi Manager. First of all it inherits the QFrame class, which represent an object composed of an header file, a source file and an ui file. If you open "Forms/gui/voronoimanager.ui", you can see the GUI of the manager. When the application is running, whenever a component (button, checkbox, …) is clicked, a special member function (Qt calls them "slots") is called. For example, if you want to modify the member function associated to the "Load points" button, you can right click on the button, select "Go to slot", select "clicked" and click OK. You should now be inside the member function called "VoronoiManager::on_loadPointsPushButton_clicked()".

In this project, the main purpose of the VoronoiManager is to give an interface to handle the execution of the fortune algorithm and to show the output. Therefore, you have to put your drawable Voronoi diagram (it must be saved in a proper data structure) as an attribute of the VoronoiManager (take the boundingBox attribute as example). If you want to draw your Voronoi diagram, your data structure (in which the diagram is saved) needs to implement the "DrawableObject" interface. Note that you should use inheritance to keep the data structure and the drawable object

independent. For example DrawableDataStructure should inherit DataStructure (so we can execute the operations we need) and DrawableObject (so we can draw it in the canvas). Another option is to define a Drawer, which takes the object as input and draws it in the canvas (use OOP composition).

To draw your drawable object in the canvas, you can call the method mainWindow.pushObj(), which takes a const pointer to a DrawableObject. We suggest you to take as model the classes BoundingBox2D, DrawableBoundingBox2D and the way we used them in the VoronoiManager.

2. SPECIFICATIONS

The project consists in implementing the **Fortune algorithm** for generating the Voronoi diagram of a given set of points.

It is asked to draw the Voronoi Diagram for a given set of points and to allow the user to insert new points to the set by just clicking in the GUI. Note that it is not an incremental algorithm, so your implementation is launched again on the entire set of points for each point clicked.

2.1. DETAILS

All the input points have to stay inside the bounding box declared in the VoronoiManager. Note that the test input files contains only points inside the given bounding box.

Do not change the values of the bounding box.

In the VoronoiManager, you have some (already implemented) slot member functions associated with buttons. They call some methods that you have to fill with your code:

- **computeVoronoiDiagram**: here you have to execute your algorithm for the input vector of points (argument “inputPoints”). The algorithm should fill your Voronoi Diagram data structure. *Hint: this data structure (that you need to fill executing your algorithm) could be called **VoronoiDiagram**. You could define an **attribute** of the VoronoiManager, called **DrawableVoronoiDiagram**, that is a DrawableObject and a VoronoiDiagram (take the attribute DrawableBoundingBox2D as example). The canvas will automatically draw your changes to the data structures, whenever the MainWindow “updatecanvas” method is called.*
- **clearVoronoiDiagram**: here you have to clear your Voronoi diagram.
- **drawVoronoiDiagram**: the goal of this method is to draw/show the drawable object in the canvas. Read the comments in the code carefully, probably you do not have to write anything on this function. Indeed you could choose to keep your drawable object always rendered (also when it is empty): the canvas will automatically draw your updated object.
- **eraseDrawnVoronoiDiagram**: the goal of this method is to erase/hide the drawable object from the canvas. Read the comments in the code carefully, probably you do not have to write anything on this function. Indeed you could choose to keep your drawable object always rendered (even when it is empty): the canvas will keep drawing an empty object.
- You will need to add some code in the **constructor** and in the **destructor** of the manager.
- Do not write code (avoid it when it is possible, and it is actually possible) in the member functions with the comment “Do not write code here”.

Data structures and algorithms can be **used** by the manager, but they must be

implemented outside of it and **organized in proper folders**. Use the Object Oriented Paradigm and try to keep data structures and algorithms as general as possible.

Note that there could be duplicates in the vector of points, since you can click multiple times in the same point. You should remove duplicates in your algorithm: you can easily do it while sorting the points (it is a step you have to do, to properly implement the Fortune algorithm).

2.2. NOTES

In order to get the maximum grade, you **necessarily** have to implement **on your own**:

- The data structure to save a Voronoi Diagram: a 2D DCEL (keep it general, complete and navigable);
- The Fortune algorithm to get a Voronoi Diagram (fill the 2D DCEL data structure).

Whoever uses (or, even worse, copies some code from) an external library or tool for these data structures/algorithms, **will be penalized.** You can obviously use all the data structures/algorithms in the standard library.

Keep in mind that these data structures and algorithms should be independent from each other (make them as general as possible).

If you find a bug on the code, or if you have questions, please write an email at stefano.nuvoli@gmail.com or open a discussion on the forum on moodle.

3. HOW TO SEND THE PROJECT

This year, you are asked to send your final project through github or bitbucket, and you are asked to **commit often during the development** of the project.

Access here to this link to have an assigned private repository:

https://classroom.github.com/assignment-invitations/ade7bae671e014052787740fd1e141a1

1. Access to your github account;
2. Accept the assignment in order to have a private repository;
3. Push the base project on your repository (**it must be your first commit**);
4. Commit your changes often and do not forget to push at the end;
5. Once the project is developed, upload on moodle a pdf file which show how you organized the project (how data structures and algorithms are linked, how are organized your file, etc.). Write in the pdf file the name of your github repository.

Projects with a too small number of commits (or no commits at all) will be rejected. A project should have at least 15-20 commits.

Remember that this is an *individual* project: you can collaborate in order to solve high level problems, but projects with similar pieces of code will be not tolerated (*both* projects will be rejected and further actions will be taken).

4. Guidelines for a Good Final Project

1. Before submitting, rename the .pro file with a name of this format: <matr>_<surname>_<name>.pro.
2. The base project compiles with zero warnings. Make sure to submit a project with zero warnings. Warnings are indicative of bad and error-prone coding. A zero-warnings code does not mean that is good, but good code always produces zero warnings.
3. Separate the definition of a class (files .h) from its implementation (files .cpp), and do not use “using namespace ...” on headers files (why? http://stackoverflow.com/questions/5849457/using-namespace-in-c-headers).
4. Try to avoid the usage of global variables and other shortcuts. Try to follow the Object Oriented paradigm as best as you can (why? http://c2.com/cgi/wiki?GlobalVariablesAreBad).
5. Please organize your code following the Object Oriented Paradigm. **Keep separated algorithms from data structures**. Write short methods which solve standalone problems when it is possible.
6. Use const keyword. If you don’t know how to use const keyword, take a look at this tutorial: (http://www.cprogramming.com/tutorial/const_correctness.html).
7. Asserting is a powerful debugging tool to test correctness of your algorithms at runtime. (how and why? https://stackoverflow.com/questions/1571340/what-is-the-assert-function).
8. Comment your code, documentation is important. It is not asked to comment each operation and line of your code, but it is required to write high-level comments which explain what a function or a block of code does.
9. Follow the guidelines given during the short C++ course!

5. Tips

5.1. Debugging

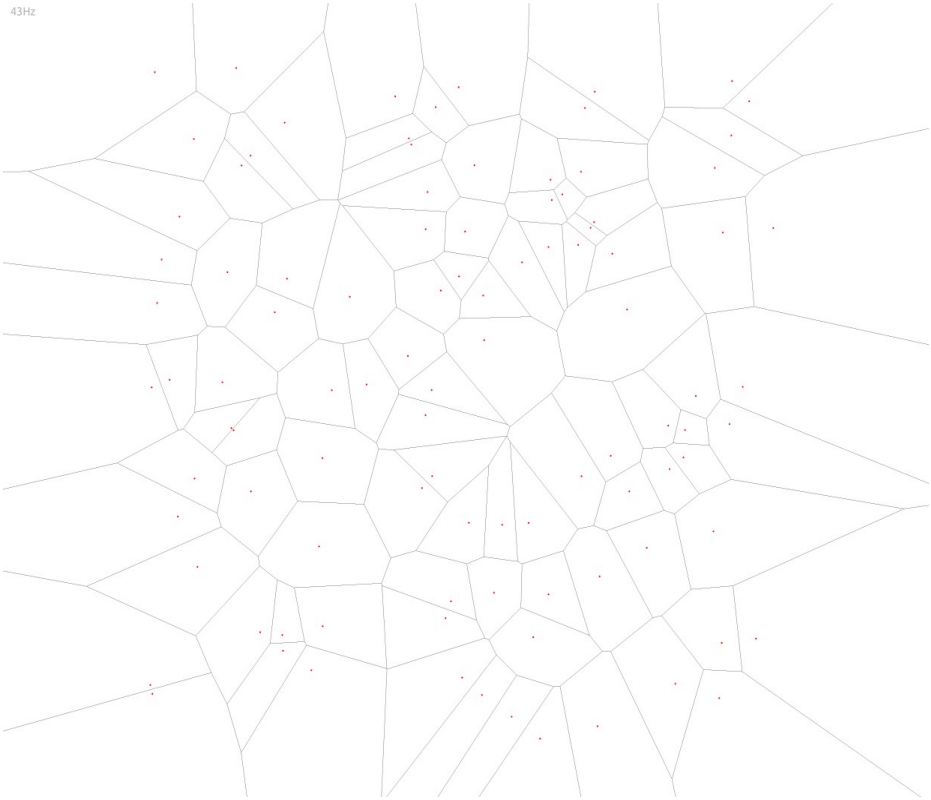
Debugger is your friend and it is a very powerful tool. In the 90% of the situations, using a breakpoint is faster (and a smarter choice) than using std::cout (or std::cerr).

The debugger allows you to find the exact point where your application is crashing (and why) and to see the state of all the variables in every scope during the execution of your application. If you have never used a debugger, this should be the best time to start.

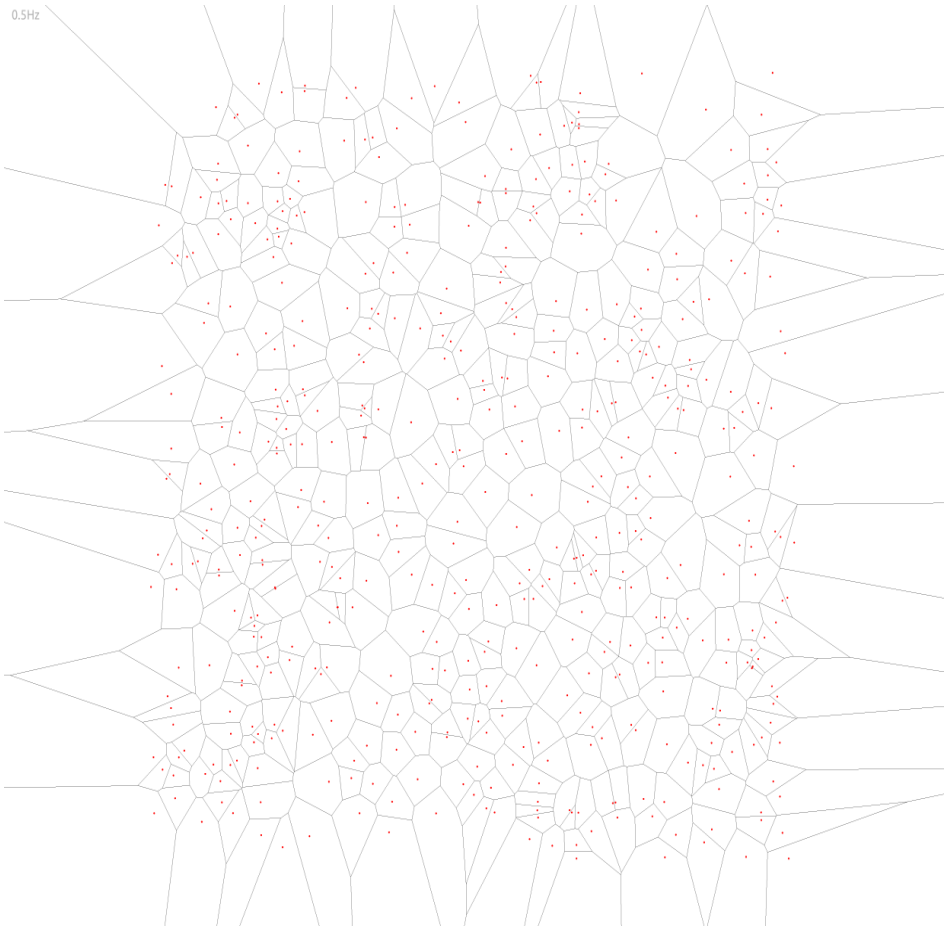
5.2. Results

Here you can find some screenshots of the results you have to expect on the given input files:

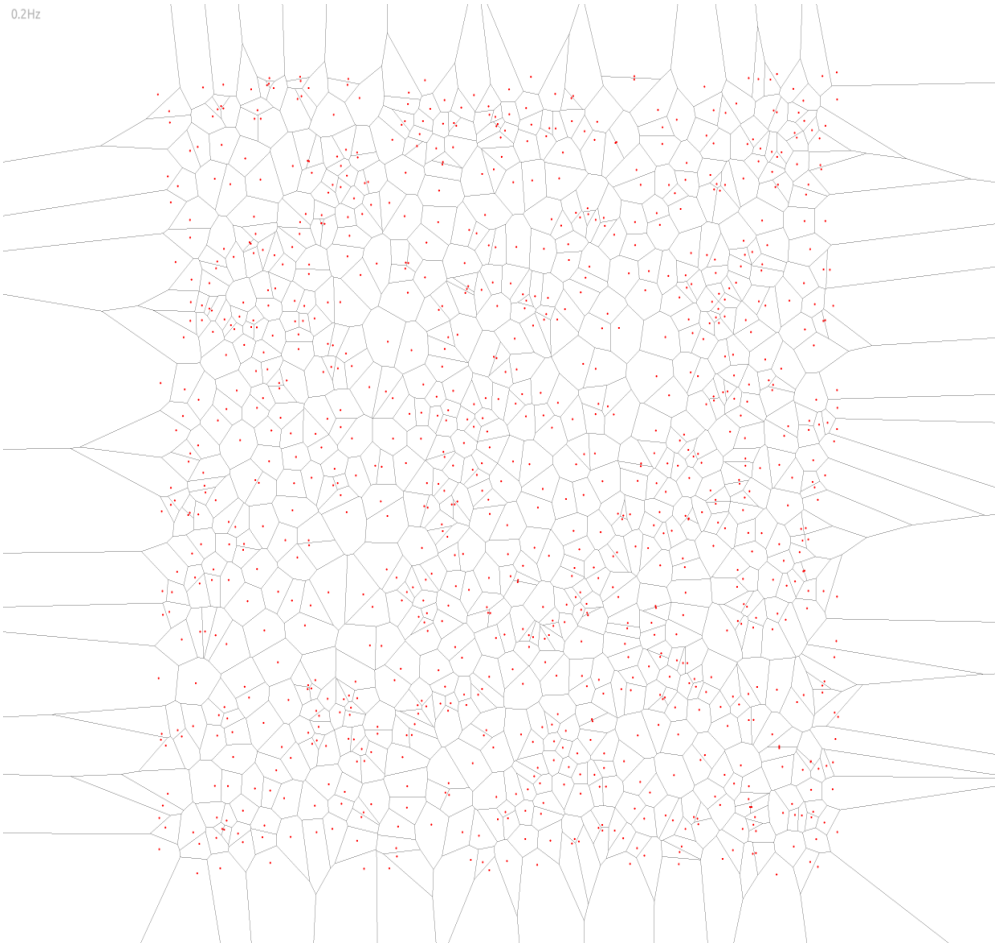
100:



500:



1000:



6. Grade and Deadlines

The grade will be composed by:

- 8/30: Correctness of the project;
- 6/30: Documentation;
- 12/30: Structure of the project, code modularity and style;
- 8/30: Efficiency;

The deadline for the project is May 31st. If you submit your project after May 31st 23:59, your grade will have a **malus** every month. This is a summary table:

Submitted before	Maximum Grade
23:59 May 31st	36/30
23:59 June 30th	34/30 (-2)
23:59 July 31st	32/30 (-4)
23:59 August 31st	30/30 (-6)
23:59 September 30th	28/30 (-8)

After September 30th, you will not be able to submit a solution for this project, but you will have to wait for the end of the winter semester 2019/2020 to get your new project.

We remind (again) that:

- **Projects with a too small number of commits (or no commits at all) will be rejected.**
- **Remember that this is an *individual* project: you can collaborate in order to solve high level problems, but projects with similar pieces of code will be not tolerated (*both* projects will be rejected and further actions will be taken).**