Computational Geometry. Lab Documentation

Vera Sacristán, Marc Sunet

Departament de Matemàtiques
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya
Year 2016-2017 Q2

1 Introduction

Throughout the lab sessions in this course we will be solving several geometric problems such as the intersection of line segments, the classification of a point's position with respect to a triangle and a circle, and the triangulation of a set of points, which in the last exercise will be lifted in 3D to obtain a three-dimensional terrain model.

To ease your task, we have provided a framework that will deal with the non-geometric aspects of the programs you will be developing, such as reading input and displaying geometry on the screen. This allows you, as a student, to focus on the geometric problem solving, at least for the first part of the laboratory. Be aware, though, that you are not obliged to use our framework; it is only intended to help you, not to constrain your work.

Laboratory work

The laboratory work is divided into five exercises. Exercises 1 to 3 are somehow independent, while exercise 4 builds on the previous one and exercise 5 build on exercise 4.

How to read this document

The sections in this document are meant to be read in order, at least until Section 4 included. At that point, you might consider skipping to Section 5 and reading further about the framework.

2 Lab Session 0: The Viewer

The viewer is not an application, but a library to build applications to view both 2D and 3D geometry. To ease the building of such applications, the viewer provides some functionalities such as reading basic geometric objects from a file and from the screen, displaying them and manipulating the camera to view the scene. As a consequence, applications built with the viewer share some common behavior. The purpose of this lab session is to introduce you to the workings of a typical viewer application by exploring one such application.

Obtaining the viewer

You can download the viewer from `/assig/grau-geoc/viewer.tar.bz2` in the faculty's server, which can be accessed remotely via the `Racó` or directly from a PC classroom.

Requirements

If you are installing the viewer on a computer of your own you will need the following libraries installed on your system:

- Qt4 libraries. Precisely: QtCore, QtGUI, QtOpenGL
- Boost >= 1.47.0

Installation on Linux

Extract the archive file, switch to the `viewer/` directory and run the setup script as shown below:

```
$ chmod +x setup.sh
$ ./setup.sh
```

The setup script will build the viewer library and a set of applications provided with it. Once the setup is complete, the library and applications will be found in the `viewer/bin` directory.

Attention! When working on the FIB computers, make sure that you download everything and do the the setup in your HOME directory! In addition, you may need to edit the file `setup.sh` and replace `pwd` by `/bin/pwd`.

Compiling the library

The setup script performs an initial build of the viewer library, but recompiling the library will be necessary for any changes made to it to take effect in the applications linked against it. It is therefore necessary to discuss how to build the viewer.

In the `viewer/` directory you will find a `Makefile`. The default option is `release`, which compiles the library optimised for speed. A `debug` option also exists, which builds the library with debug symbols and enables certain precondition assertions, making applications easier to debug. You are encouraged to work with a debug build of the library while developing your algorithms.

To build the library in debug mode, switch to the `viewer/` directory and run:

```
$ make debug
```

To build it in release mode, simply run:

```
$ make
```

Note that recompiling the applications built with the viewer library is `NOT` necessary, as the viewer is built as a shared library after all.

Running the viewer

Amongst all of the applications shipped with the library there is an example, which can be found in the `/viewer/bin/demos` directory. Please run the it to start getting acquainted with the viewer. The example viewer displays points, segments, triangles, and circles read from a file or from the screen.

```
$ ./run example
```

Since the library is compiled as a shared object, we must tell the system where it is located when running any application linked to it. The `run` script takes care of this by overriding the `LD_LIBRARY_PATH` variable.

You should now be seeing the example application. At the very top there is a menu bar with a few options to manipulate the scene. Right at the bottom we can see an empty scroll list and a status bar displaying certain information: the camera's projection mode, the camera's control mode, which can be either 2D or 3D, and the state of the screen input subsystem, to be discussed later.

Adding geometric objects: Screen input

Objects are introduced by switching the screen input subsystem into a specific state and then inserting geometry using the mouse:

- KEY_SPACE Idle state.
- KEY_P Point mode.
- KEY_S Segment mode
- KEY_T Triangle mode.
- KEY_C Circle mode.

Select an input mode from the above by pressing its assigned key and then use `LMB` to enter geometric objects.

You may as well delete the objects you have entered, undo the last set of operations, and redo them:

- KEY_BACKSPACE Deletes the last object.
- KEY_CONTROL + KEY_Z Undoes the last operation.
- KEY_CONTROL + KEY_R Redoes the last undone operation.

Manipulating the view

The view can be manipulated in several different ways:

- Pan: `Drag LMB`.
- Zoom: `KEY_SHIFT + drag LMB` or using the scroll wheel.
- Zoom + pan: Because zoom and pan are not mutually exclusive, you can zoom+pan by combining both operations.
- Restore the initial view: `KEY_I`.
- Switch to 2D mode: `KEY_2`.
- Switch to 3D mode: `KEY_3`.
- Change the thickness of objects (increase or decrease): `KEY_+` and `KEY_-`.
- Toggle wireframe: `KEY_F`.
- Toggle projections (orthogonal or perspective): `KEY_V`.
- Toggle labels: `KEY_L`.

In 3D mode, you can also do the following:

- Rotation (angle ψ): While pressing `KEY_CONTROL+LMB`, drag the mouse horizontally.
- Rotation (angle θ): While pressing `KEY_CONTROL+LMB`, drag the mouse vertically.
- Rotation (angle ϕ): While pressing `KEY_Z`, drag the mouse horizontally.

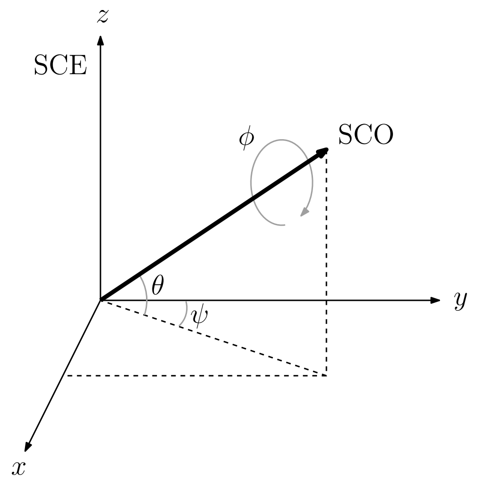


Figure 1: Role of the angles ψ, θ and ϕ in the relationship between the scene coordinate system (SCE) and the observer coordinate system (SCO).

Adding geometric objects: File Input

Viewer applications will generally take an input file as an optional parameter. The application will parse the file passed to it during initialisation and load the geometric objects described in it.

To illustrate this, we have provided the file `example_input.txt`, which can be found in the `/assig/grau-geoc/viewer/bin/input/` directory. To pass this file as an argument, run the following command:

```
$ ./run example input/example_input.txt
```

Close the example application and do this now. Notice how the geometry described in the file now appears on the screen. One can also load a scene file or reload the current one via the `Scene` menu in the application's menu bar.

Saving the scene

The contents of the scene can be saved either by pressing `KEY_G` or by choosing the menu option `Scene -> Save`. The default output file is `output.txt`. You may choose another file name by choosing the menu option `Scene -> Save As`, which will ask you for the file to output the scene contents to. Further saves (`KEY_G` or `Scene -> Save`) will output to the file just chosen.

Saving the scene can be useful, for example, in a situation where you come across a set of input data for which the algorithm you are working on does not behave properly and you wish to save the contents of the scene to later analyse it or show it to your instructor.

3 The Viewer Framework Explained

We shall now explain what is going on behind the scenes. We discuss the tools the framework makes available and how an application can use them to perform its tasks.

General view

At a very high level, we can break up a viewer application into three main areas:

The world represents input files and the user itself, who is ready to input more geometry into the system via the screen.

Observers watch geometry as it moves into the system. Observers have total control over what goes into the application; they can transform geometry in any way, discard it or move it into the scene.

The scene represents the geometry being currently displayed on screen.

The cool part of it all is that the framework provides no default observer; the bridge between the world and the scene does not exist. Instead, it is the application's role to provide at least one observer to get anything done.

What the framework does provide, however, is a set of classes to read points, segments, triangles, and circles from a file and from the screen. We are not limited to this restricted set of geometric types, however; we can add our own types to the system without modifying existing code, as shown in Section 7.

The World

The world represents input geometry. The framework provides us with two components for handling input: an `object loader`, which handles file input, and a `screen input component`, which reads geometry from the screen.

The Object Loader

The object loader does not know how to load any kind of geometry by default. An application adds functionality to the object loader by attaching loaders to it at runtime. For instance, the example application attaches four loaders to the object loader: a point loader, a segment loader, a triangle loader, and a circle loader. It is through the aggregation of components that the object loader learns how to load specific geometric types. The design of the object loader is similar to that of a chain of responsibility.

Once again, we are not limited to the four loaders just mentioned. We could, for example, make a loader which is able to read `obj` files and attach it to the object loader, in which case the object loader would then be able to load such files.

Every concrete loader may have attached any number of concrete observers of its own kind. For example, a triangle loader could have triangle observers attached to it. When the triangle loader loads a triangle, it notifies all of its observers about the new triangle.

The Screen Input Component

The screen input component is a state machine for handling user input. An application adds functionality to the input machine by configuring its states.

The example application, for instance, adds four states to the screen input: a state for reading points, one for reading segments, one for triangles, and one for circles. In the process, the example application sets up the key bindings used to switch to each of these states; for this reason, the keys used to switch to each of these states, `KEY_P`, `KEY_S`, `KEY_T`, and `KEY_C`, are application-specific.

The screen input component provides a default state: the `idle` state. Its key binding is `KEY_SPACE`; this means every viewer application will have a default input state, which is the `idle` state, the key binding for which is `KEY_SPACE`.

As with loaders, a screen input state may have any number of observers of its own kind attached to it. For instance, the state for reading points could have point observers attached to it; when a point is read from a screen, the point state notifies all of its observers about the new point.

The Observers

An application must provide at least one observer to manipulate data and to get anything viewed on the screen. An observer acts as a bridge between the world and the scene, and it can manipulate data as it flows in. The relationship between the object loader, screen input and observers is that of an observer pattern.

The example application provides an observer which is able to observe points, segments, triangles, and circles. This observer is attached to all loaders and screen input states; by these means, whenever a piece of geometry is read, the observer is notified about it.

The observer in the example application simply attaches the geometry it receives to the `scene manager` so that it is rendered on the screen.

The Scene

The scene represents those objects being displayed on the screen. The framework provides a `scene manager` that handles the rendering for us, allowing us to attach and detach objects from the scene as well as undoing and redoing insertion and deletion operations. For this reason, all viewer applications will have the undo/redo functionality baked into them. The keys to carry out these operations are also common to all viewer applications.

The Camera Control Component

The last component the framework provides us with which we should be aware of is the `camera control component`. The camera control is an implementation of the state pattern with two possible states: one for 2D control and one for 3D. It is technically possible to customise the controls, for example by adding a free-fly first person view. All viewer applications share the same camera controls.

Summary

The viewer framework:

- Provides an `object loader` for handling file input.
- Provides a `screen input component` to handle input from the screen.
- Provides a `scene manager`, with undo and redo functionalities baked into the framework.
- Provides a `camera control component`; the controls to manipulate the camera are also hard-coded into the framework.

A viewer application:

- Customises the `object loader` by attaching loaders to it.
- Configures the `screen input component` by configuring its states, setting up key bindings in the process; the key bindings used to switch between states are application-specific.
- Provides an observer to bridge the input components and the scene manager.

4 Lab Exercises

Along the lab exercises you will be making changes to the viewer library. It is advised that you complete the exercises in the order they are provided, as the guidelines for one exercise build on those provided for the preceding ones.

Note that you are free to make any changes to the library apart from the ones you are asked to do. These include adding auxiliary methods or additional properties to a class. You should not, however, remove or rename any of the methods you are asked to modify since the applications provided depend on them.

During each lab exercise, only the ideas which are strictly necessary to complete the exercise at hand will be discussed. Often, you will be pointed out to *the docs* for additional information. These refer to the full reference documentation of the framework, which can be found in `docs/docs/html/index.html` inside the `viewer/` directory.

Exercise 1

Goal To test and classify the intersection of two line segments.

Classes/modules involved LineSegment, Math, Vector

Application directory `viewer/exercises/exercise1`

Description

Write a program to decide whether or not two line segments intersect. Recall that if two segments share an endpoint, they do intersect. Your program must be able to deal with aligned points. If two segments intersect, your program must indicate whether the intersection is a point or a segment, whether the intersection point is an interior point or an endpoint, whether the intersection coincides with one of the original segments, etc.

Guidelines

An application is provided to help you complete this exercise. The application takes care of input and displaying line segments on the screen. The code for this exercise's application resides in `viewer/exercises/exercise1`.

The application links against the viewer library, but the library is missing two key points: it does not know how to perform an orientation test, and it does not know how to intersect line segments. Your job is to add the code to get these two pieces working. No changes need to be made to the application itself; you only need to modify the viewer library.

Compile the viewer library if you have not already done so. Switch to `viewer/bin` and run the application by running the following command:

```
$ ./run ex1 tests/TestLab1.txt
```

The application should launch and display a set of white line segments and a list with many entries reading "To-do". You must fix the viewer library to get this application working. Line segments should be drawn in a different colour for every kind of intersection. A description about the kind of intersection should also be provided.

Note that if you toggle labels on (`KEY_L`) you will see the line segments numbered in the same order as they appear in the input file. This allows you to quickly identify any pair of segments, which should be helpful when debugging your application.

Exercise 1 has been broken up into two parts, each of which deals with one particular sub-problem.

Part One: Orientation test in 2D

Implement a 2-dimensional orientation test. The function body you must complete resides in `src/geoc/math/Orientation.cc`:

```
num Math::orientation2D(const Vector3& p, const Vector3& q,
                        const Vector3& r)
{
    //Exercise 1.
    return 0;
}
```

This function takes three points, p, q and r, and returns the sign of the oriented area of the triangle Δpqr.

The `Math` namespace is a namespace grouping up several helper functions, one of which is the `orientation2D` function.

The return type for `Math::orientation2D` is `num`. `num` is a type defined as `double` in `src/geoc/geoc.h`. You will see `num` used all across the library; it allows us to easily switch between single and double point precision by changing its type definition to either `float` or `double`.

The arguments passed to the orientation test function are three vectors, p, q, r, each of type `Vector3`, defined in `src/geoc/math/Vector.h`.

The `Vector` class provides operators to deal with addition/subtraction/multiplication/division of both vectors and vectors and scalars, plus an indexing operator, the `[]` operator, which returns a reference to the desired vector's component.

To index a vector, we must pass a value within the range $[0, N)$, where N is the size of the vector. For convenience, we have also defined the following enumeration in `geoc/math/Vector.h`:

```
enum COORD { X, Y, Z, W };
```

Since $X = 0$, $Y = 1 \ldots W = 3$, we can index vectors using these constants instead of numbers $0 \ldots 3$ for the sake of readability.

Finally, we can print a vector using the $<<$ operator.

A few examples shall illustrate these concepts:

```
Vector3 v1, v2, v3;
num a;

v1 = Vector3(1, 2, 3);
v2 = Vector3(4, 5, 6);

v3 = v1 + v2;        //v3 = Vector3(5, 7, 9)

cout << v3;          //v3 gets printed to the standard output stream.

v3 = v1 * v2;        //v3 = Vector3(4, 10, 18)

v3 = v1 + 2;         //v3 = Vector3(3, 4, 5)
```

```
a = v1[0] + v2[0];  //a = 5

a = v1[X] + v2[X];  //same as the above.

v1[Y] = 3;          //v1 = Vector3(1, 3, 3)
```

This should be enough to get you started. You should now be ready to complete part one of this exercise. For more information about class `Vector`, check out *the docs*.

Part Two: Line segment intersection

For the last part of this exercise, you will need to implement the function that classifies the intersection between two line segments, the empty definition of which can be found in `src/geoc/geometry/LineSegment_utils.cc`:

```
void geoc::classifyIntersection(const LineSegment& s,
                                const LineSegment& t,
                                Colour3& colour, std::string& desc)
{
    //Exercise 1.
    colour = Colour3(1, 1, 1);
    desc = "To-do";
}
```

This function takes as an input two `LineSegment`s and outputs two values: a `colour` to draw the two segments in, and a `desc`ription of the kind of intersection between the line segments. You should choose a unique colour and description for every kind of intersection.

Class `LineSegment` is defined as shown below. The complete definition can be found in `src/geoc/geometry/LineSegment.h`.

```
class LineSegment
{
    Vector3         points[2];

public:
    ...

    //! Gets the ith vertex.
    const Vector3& operator[](int index) const;

    ...
}
```

You can use the `[]` operator to access a segment endpoint, as we did with `Vector3`. `LineSegment::operator[]` takes an index in the range $[0, 1]$ and returns a reference to the vertex (of type `Vector3`) in that position.

This is all you need. For more information on class `LineSegment`, do not forget to check out *the docs*.

Tests and Delivery

Test file. The file `TestLab1.txt` contains some sample input comprised of a series of line segments. The application will process them in order and test for the intersection of each pair of consecutive segments.

You are encouraged to test the correctness of your code with this file, but also with your own tests. You can pass the input file to the application by running the following command within the `viewer/bin` directory:

```
$ ./run ex1 tests/TestLab1.txt
```

Delivery. Deliver at the *Racó* an archive `lastname_name_1` containing the files that you have modified. In addition, include in the archive a brief pdf document, including any necessary command line options and buttons needed to make your program run, as well as a brief description of your algorithm's special features.

Exercise 2

Goal To classify the relative position of a point with respect to a triangle.

Classes involved Triangle, Vector

Application directory `viewer/exercises/exercise2`

Description

Write a program to decide whether a point is located in the interior, the boundary or the exterior of a triangle. When the point lies in the boundary, the program must also detect whether the point coincides with a vertex or not. Make sure that your program works properly, no matter whether the vertices of the triangle are given in clockwise or counter-clockwise order.

Input. The three vertices of the triangle and a set of points.

Output. The test points are drawn in green if they lie in the interior of the triangle, in red if they lie in the exterior, yellow if they lie on the interior of an edge of the triangle, and in another color of your choice if they lie on a vertex of the triangle. A description shall accompany each test case describing the results.

Guidelines

Yet another application has been provided to you to complete this exercise. The code for this application can be found in `viewer/exercises/exercise2`. The application handles input and the display of triangles and points on the screen, but the library it is linked against, the viewer library, is missing one key part: it does not know how to classify a point's position with respect to a triangle. Your job is to fill in the missing functionality.

Switch to the `viewer/bin` directory and run the following command:

```
$ ./run ex2 tests/TestLab2_1.txt
```

This time we get white points and a "To-do" description for every test.

Note that if you toggle labels on (`KEY_L`) you will see the input points numbered in the same order as they appear in the input file. This can be useful to identify points should any of them not be classified properly.

The function you must implement is a `Triangle` helper function, the definition of which lies in `src/geoc/geometry/Triangle_utils.cc`:

```
void geoc::Triangle::classify(const Triangle& t, const Vector3& p,
                              Colour3& colour, std::string& desc)
{
    //Exercise 2.
    colour = Colour3(1, 1, 1);
    desc = "To-do";
}
```

An incomplete definition of class `Triangle` now follows. The full definition can be found in `src/geoc/geometry/Triangle.h`.

```
class Triangle
{
    Vector3     points[3];

public:
    ...

    //! Gets the ith vertex.
    const Vector3& operator[](int index) const;

    ...
}
```

As usual, you can access a triangle's vertices using the `[]` operator. This operator takes an index in the $[0, 2]$ range and returns a reference to the specified `Vector3` component.

This is all you need to complete this exercise. Remember to check *the docs* for more information on any of the classes discussed.

Tests and Delivery

Test files. Please, use the three test files, namely `TestLab2_i.txt` ($i = 1, 2, 3$) to test the proper working of your program.

Delivery. Deliver at the *Racó* an archive `lastname_name_2` containing the files that you have modified. In addition, include in the archive a brief pdf document, including any necessary command line options and buttons needed to make your program run, as well as a brief description of your algorithm's special features.

Exercise 3

Goal To classify the relative position of a point with respect to a circle.

Classes involved Math, Circle

Application directory `viewer/exercises/exercise3`

Description

Write a program to decide whether a point lies in the interior, the boundary or the exterior of a circle defined by three points. Make sure that your program works properly, no matter whether the three points are given in clockwise or counter-clockwise order.

Input. Three points defining a circle and a set of test points.

Output. The test points are drawn in green if they lie in the interior of the circle, in red if they lie in the exterior, yellow if they lie on the boundary. A description shall also accompany each test case.

Guidelines

As usual, an application is provided to help you complete this exercise. The application takes care of input and display points and circles, but the viewer library it links against is missing some functionality: the 2.5-dimensional orientation test function and the `Circle` helper function for classifying points are missing their implementation. Your job is to implement these functions.

The code for this exercise's application can be found in `viewer/exercises/exercise3`. To run the application, switch to the `viewer/bin` directory and run the following command:

```
$ ./run LabTest3_1.txt
```

Once again we just get white points and a dummy description for each test case.

Once again, toggling labels on (`KEY_L`) will show the input points numbered in the same order as they appear in the input file, allowing you to identify them quickly.

Because you will need an orientation test function to complete this exercise, we have broken this section into two parts like we did for Exercise 1.

Part One: Orientation Test in 2.5D

The 2.5-dimensional orientation test function is defined in `src/geoc/math/Orientation.cc`:

```
num Math::orientation25D(const Vector2& p, const Vector2& q,
                         const Vector2& r, const Vector2& t)
{
    //Exercise 3.
    return 0;
}
```

The function takes three 2-dimensional points p, q and r, and a test point t, projects them vertically onto the unit paraboloid and returns the sign of the oriented volume of the tetrahedron defined by the projected points p^*, q^*, r^* and t^*.

Part Two: Circle-Point classification

The circle-point classification function is defined in `src/geoc/geometry/Circle_utils.cc`:

```
void geoc::classify(const Circle & c, const Vector3& p,
                    Colour3& colour, std::string& desc)
{
    //Exercise 3.
    colour = Colour3(1, 1, 1);
    desc = "To-do";
}
```

The function takes a `Circle` and a `Vector3` as inputs and outputs a `colour` and a `desc`ription describing the relative position of the given point with respect to the given circle.

An incomplete definition of class `Circle` follows. The complete definition can be found in `src/geoc/geometry/Circle.h`.

```
class Circle
{
    Vector3 points[3];

public:
    ...

    //! Gets the ith vertex.
    const Vector3& operator[](int index) const;

    ...
}
```

This should look pretty familiar already. The indexing operator takes a value in the range [0..2] and returns the `Vector3` component in that position.

Tests and Delivery

Test files. Please use the two test files, namely `TestLab3_i.txt` ($i = 1, 2$) to test your code.

Delivery. Deliver at the *Racó* an archive `lastname_name_3` containing the files that you have modified. In addition, include in the archive a brief pdf document, including any necessary command line options and buttons needed to make your program run, as well as a brief description of your algorithm's special features.

Exercise 4

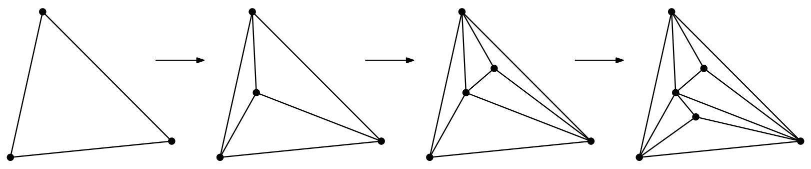
Goal To triangulate a set of points.

Classes involved Triangulation

Application directory `viewer/exercises/exercises4-5`

Description

Write a program to incrementally construct a triangulation of a set of points in the plane. Starting from an initial enclosing triangle, points will be added one by one. At each step, the new point is inserted in the current triangulation by locating the triangle it falls in and replacing it by three new triangles connecting the newly inserted point with the vertices of the disappeared triangle, as shown below:



Use an efficient data structure, such as a DCEL, to store the triangulation. This will be essential for the algorithm.

In order to achieve a reasonably quick location of the triangle that contains the following point in the insertion order, use an auxiliary fixed point (namely, one of the given points).

Important: this exercise comes with two goals: (i) to construct a triangulation for points in general position (i.e. assuming the input points do not include any three collinear points, or any 4 concyclic points), and (ii) to construct a triangulation for points in any position (that is, handling degeneracies such as collinear and concyclic points).

Solutions achieving only goal (i) will be subject to a maximum grade of 8. To obtain the maximum grade of 10 it is necessary to address successfully goal (ii).

Input. A set of points read from a file.

Output. The triangulation is shown in the screen.

Guidelines

You are provided with an application to help you complete exercises four and five. Exercise 5 builds on Exercise 4, which is why we provide a single application for both. The application takes care of input and display. You can find the code for this exercise's application in `viewer/exercises/exercises4-5`.

The application reads a set of points from a file and passes them to a `Triangulation` class instance to triangulate them. The `Triangulation` class is missing all of its functionality, however. Your task it to implement its empty methods for the triangulation to be done.

The definition of class Triangulation can be found in src/geoc/geometry/Triangulation.h. An incomplete definition now follows:

```
class Triangulation
{
public:
    ...

    void triangulate(const std::vector<Vector3>& ps
                     const std::vector<int>& idxs,
                     std::vector<LineSegmentEnt>& segments,
                     std::vector<TriangleEnt>& triangles,
                     std::vector<TriangleEnt>& triangles_pruned);

    ...
}
```

Class Triangulation exposes a single method, triangulate, but its definition is empty (see src/geoc/geometry/Triangulation.cc). This method is called to triangulate a given set of points.

The ps vector contains the set of points to be triangulated.

The idxs vector is of no importance right now; we will come back to it later in the section devoted to Exercise 5.

Once the triangulation has been built (i.e., the DCEL has been completed), the segments, triangles and triangles_pruned vectors are output parameters where the triangulation data should be dropped for visualisation. The triangles_pruned vector will be relevant in Exercise 5.

In this exercise, you can choose whether to display a triangulation as a set of segments or as a set of triangles. Should you choose the former, segments is where you will want to store the segments describing the triangulation, otherwise the triangles vector is where you should place your triangles.

Notice that these vectors, segments and triangles, are vectors of LineSegmentEnt and TriangleEnt, respectively. These are beefed up line segments and triangles. Classes LineSegmentEnt and TriangleEnt subclass LineSegment and Triangle, respectively, and enhance their base class by adding extra functionality to it. The definitions for these classes can be found in src/geoc/scene/LineSegmentEnt.h and src/geoc/scene/TriangleEnt.h.

Incomplete definitions for these classes follow:

```
class LineSegmentEnt : public LineSegment, public Entity
{
public:
    ...

    Colour3 colour;

    LineSegmentEnt(const LineSegment& s);

    void setLabel(const std::string& label);

    ...
};


class TriangleEnt : public Triangle, public Entity
{
public:
    ...

    Colour3 colour;

    TriangleEnt(const Triangle& t);

    void setLabel(const std::string& label);

    ...
};
```

In particular, we may build a `LineSegmentEnt` from a `LineSegment` and a `TriangleEnt` from a `Triangle` by calling the shown constructors. Notice that we need not do this explicitly; C++ will handle implicit conversions when a conversion exists and there is no ambiguity, allowing us to do the following:

```
void Triangulation::compile(std::vector<LineSegmentEnt>& segments ...)
{
    LineSegment s; // LineSegment, not LineSegmentEnt.
    segments.push_back(s); // Implicit conversion from LineSegment to
                           // LineSegmentEnt.
}
```

The public `colour` attributes these classes expose, of type `Colour3`, which is a `typedef` for `Vector3`, can be assigned any colour of your choice. The triangle or segment will then be drawn in that particular colour. The field initialises to black $(0.0, 0.0, 0.0)$ by default. The colour components are floating point values ranging from 0 to 1, not integer values.

Finally, the `setLabel` method exposed by these classes can be called to assign a label to the triangle or segment. When labels are toggled on in the viewer application, the triangles and segments will display the labels assigned to them. This can be useful to show debug information.

Feel free to modify the `Triangulation` class in any way. You may add any attributes and helper methods you might feel necessary.

Also note that recompiling the applications for exercises 4 and 5 is not necessary after making changes to the header file `Triangulation.h`. Recompiling the library is sufficient for the changes to take effect.

Further Hint

Class `Triangle` exposes a constructor taking three points (`Vector3`), as shown below:

```
class Triangle
{
    ...
    Triangle(const Vector3& p1, const Vector3& p2, const Vector3& p3);
    ...
}
```

A triangle will be built so that it is facing the camera if `p1`, `p2`, and `p3` are given in counter-clockwise order.

Tests and Delivery

In the folder `viewer/bin/tests/TestLab4-5` you will find point sets corresponding to several different real terrains.

Each test file consists of one point set: a list of points with their coordinates (x, y, z), one per line, followed by some more lines in some cases, with information that you will need in the next exercise. In this exercise you will only use the x and y coordinates of the points.

Start testing your application with the file `Lanzarote.txt`, which can be found in the `viewer/bin/tests/TestLab4-5/` directory. The distribution of the points over the actual terrain is randomized, so you can initially use a smaller amount of points to start testing your program (by simply copying as many points as you want to a different input file).

The point sets in filenames with the word "Degenerate" are not in general position (i.e. contain sets of 3 collinear points and/or sets of 4 concyclic points).

Even if you are aiming at solving goal (ii) (i.e. handling degeneracies), it is recommended to first implement your algorithms assuming that no degeneracies occur, and test your program thoroughly with Lanzarote. Only after you are sure that the output produced by your algorithm is correct, extend your code to handle degeneracies and test your program with the remaining test files.

Delivery. Deliver at the *Racó* an archive `lastname_name_4` containing the files that you have modified. In addition, include in the archive a brief pdf document, including any necessary command line options and buttons needed to make your program run, as well as a brief description of the elements which you consider to be specific to your implementation, such as:

- How your program computes the enclosing triangle.
- Which is the precise data structure used to represent the triangulation (i.e. DCEL or similar).
- Which auxiliary point you use to speed up the location phase (or how you do the location, if you have chosen to implement a different method).
- Any additional peculiarity of your implementation that you wish to emphasize.

Exercise 5

Goal To compute a Delaunay triangulation of a set of points in general or in any position.

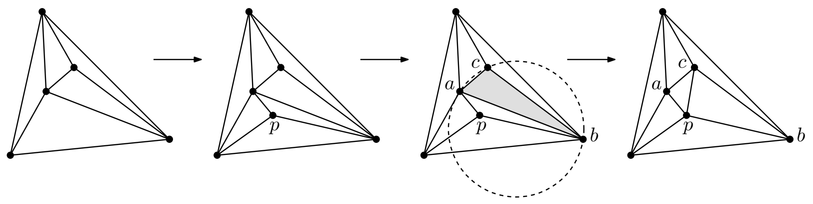
Classes involved Triangulation

Application directory exercise_4-5/

Description

Write a program to incrementally construct a Delaunay triangulation of a set of points in the plane.

This algorithm is a modification of the previous one in Exercise 4. The difference lies in the fact that the Delaunay property must hold at each step of the algorithm. In other words, each time that one point p is inserted in the triangulation, the algorithm recursively checks, for each triangle t incident to p whether or not p lies in the circumcircle of the triangle adjacent to t through the edge opposite to p. If it does, the common edge of the two triangles is replaced by the edge connecting p with the opposite vertex in the other triangle. In the figure, p is interior to the circumcircle of the triangle abc: in this case, the edge ab is replaced by the edge pc. This test is repeated for all the triangles incident to p which appear after the edge flips. A new point can be inserted in the triangulation only when p does not lie in the interior of any of the circumcircles of the triangles that are adjacent to the triangles incident to p.



Important: as in the previous exercise, this exercise comes with two goals: (i) to construct a Delaunay triangulation for points in general position (i.e. assuming the input points do not include any three collinear points, or any 4 concyclic points), and (ii) to construct a Delaunay triangulation for points in any position (that is, handling degeneracies such as collinear and concyclic points).

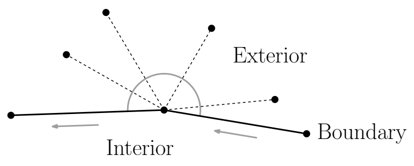
Solutions achieving only goal (i) will be subject to a maximum grade of 8. To obtain the maximum grade of 10 it is necessary to address successfully goal (ii).

Lanzarote input file: dealing with the boundary

The test file corresponding to Lanzarote comes with additional boundary information: in addition to the terrain points, you are told which of them form the boundary of the island. This allows, once the Delaunay triangulation has been obtained, to eliminate the triangles lying in the ocean, which do not correspond to the triangulation of the terrain itself. The file contains a list of vertex indices, corresponding to those vertices belonging to the boundary of each connected component, and they are given in counter-clockwise order along the boundary of each island.

In order to eliminate the undesired edges, two cases can appear:

- The connected component is one single point: Eliminate all the incident edges (as well as all the corresponding triangles).
- The connected component has more than one point. Each point has two incident boundary edges, and the location of the interior and the exterior of the island is known. Eliminate all the external incident edges (dotted lines in the figure), as well as all the corresponding triangles.



The next step is to reconstruct the terrain in 3D. From the resulting 2D pruned Delaunay triangulation, produce the 3D triangles in the appropriate format and visualize them on your screen. Compare the resulting terrain with the one that can be obtained using the triangulation of Exercise 4.

Input. A set of points read from a file.

Output. The Delaunay triangulation is shown on the screen.

Guidelines

You shall modify the code you wrote for Exercise 4 so that it produces a Delaunay triangulation. The basic data flow remains the same.

We must now discuss class `Triangulation` a little further. Recall the definition we presented in Exercise 4:

```
class Triangulation
{
public:
    ...

    void triangulate(const std::vector<Vector3>& ps
                     const std::vector<int>& idxs,
                     std::vector<LineSegmentEnt>& segments,
                     std::vector<TriangleEnt>& triangles,
                     std::vector<TriangleEnt>& triangles_pruned);

    ...
}
```

The `segments` and `triangles` vectors are the ones where you should drop your segments or triangles, just as you did in Exercise 4.

Exercise 5 requires a further step, however, which is leaving the triangles of the pruned triangulation in the `triangles_pruned` vector. A pruned triangulation consists of only those triangles belonging to the actual terrain; triangles along the boundary not pertaining to the terrain must be discarded.

The `idxs` vector holds indices into the `ps` vector, pointing out the vertices along the boundary. You should use these indices to discard the triangles that do not belong to the terrain. An example index vector would be the following:

```
3 51 0 1 -1 8 37 84 27 114 87 35 -1 267 51 978 264
```

Indices in the `idxs` vector fall in the range $[-1 \ldots N)$, where N is the number of points in the `ps` vector. Negative indices (-1) separate index groups. The index groups in this example are:

1. $[3, 51, 0, 1]$
2. $[8, 37, 84, 27, 114, 87, 35]$
3. $[267, 51, 978, 264]$

Each index group denotes the boundary of a connected component. In this example we have 3 connected components.

The `idxs` vector may be empty, however, in which case no pruning is required. This will be the case for some test files other than the Lanzarote one. Your code should behave properly with an empty indices vector.

To prune the triangulation you will have to modify the definition of the `triangulate` method and leave the pruned triangulation in the `triangles_pruned` vector.

Remember that you can colour segments and triangles in any way and add labels to them.

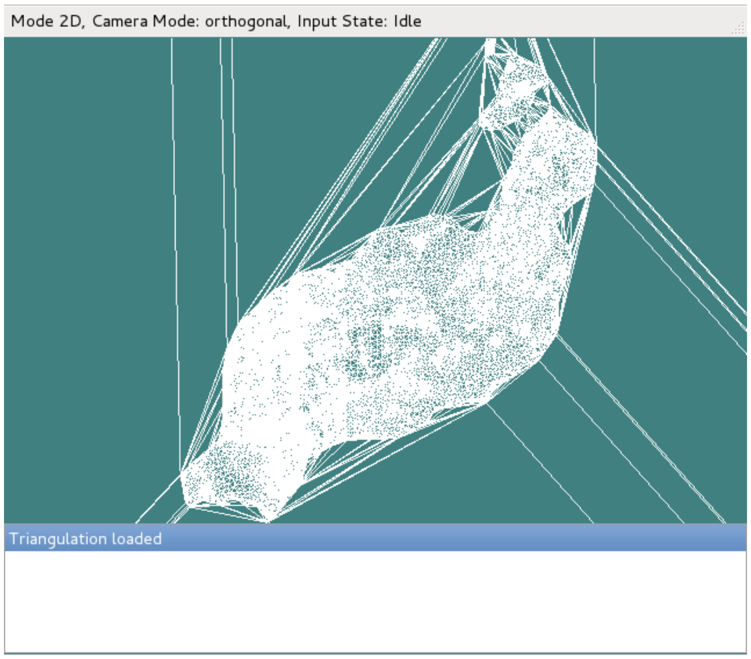
Triangulation Views

The application for this exercise lets us switch between different triangulation view modes, three of which are 3D views. To enable these 3D views, you must pass the flag `--3d` to the application:

```
$ ./run ex4-5 tests/lanzarote.txt --3d
```

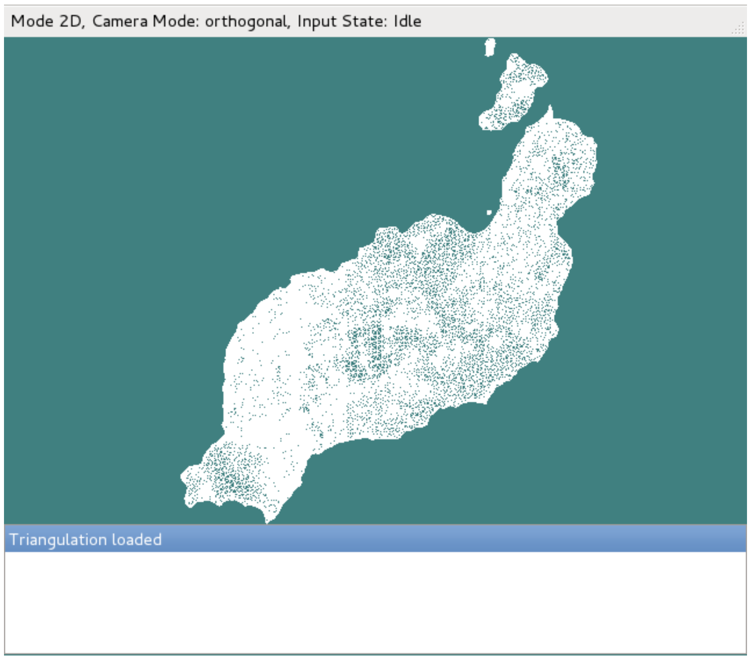
The keys to switch between the different modes are the following:

- `KEY_B` 2D triangulation. This option draws the segments in the `segments` vector or the triangles in the `triangles` vector, depending on which is filled, all in wireframe mode.


Mode 2D, Camera Mode: orthogonal, Input State: Idle

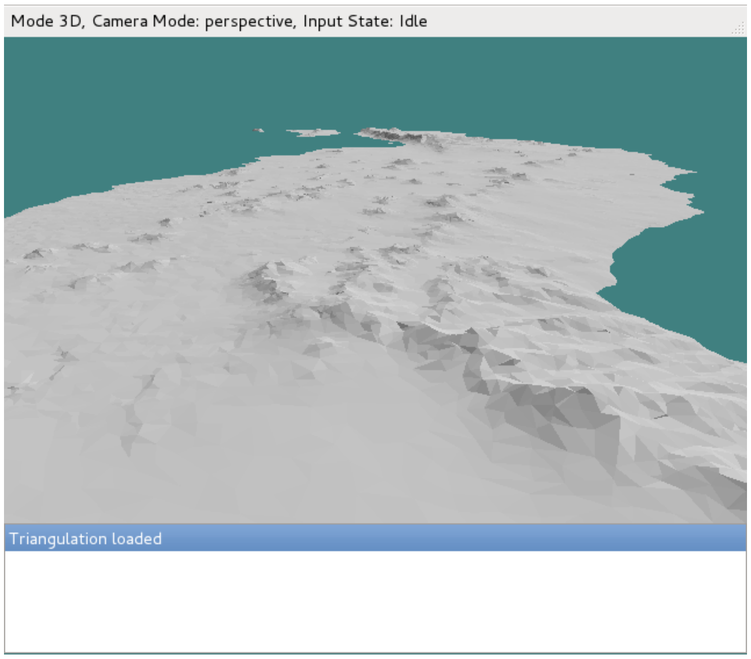
Triangulation loaded

- `KEY_H` 2D pruned triangulation. This draws the triangles in the `triangles_pruned` vector, omitting their `Z` coordinate and thus producing a flat, pruned wireframe view of the triangulation.

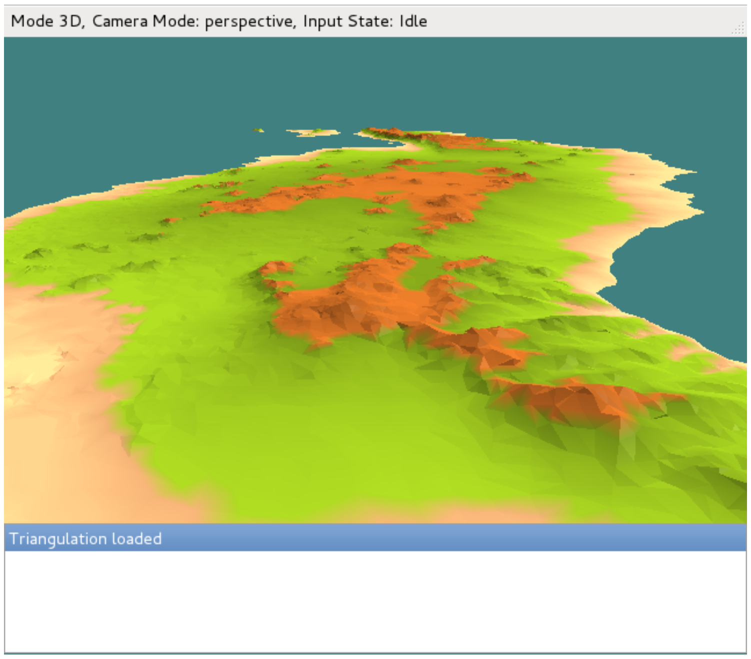

Mode 2D, Camera Mode: orthogonal, Input State: Idle

Triangulation loaded

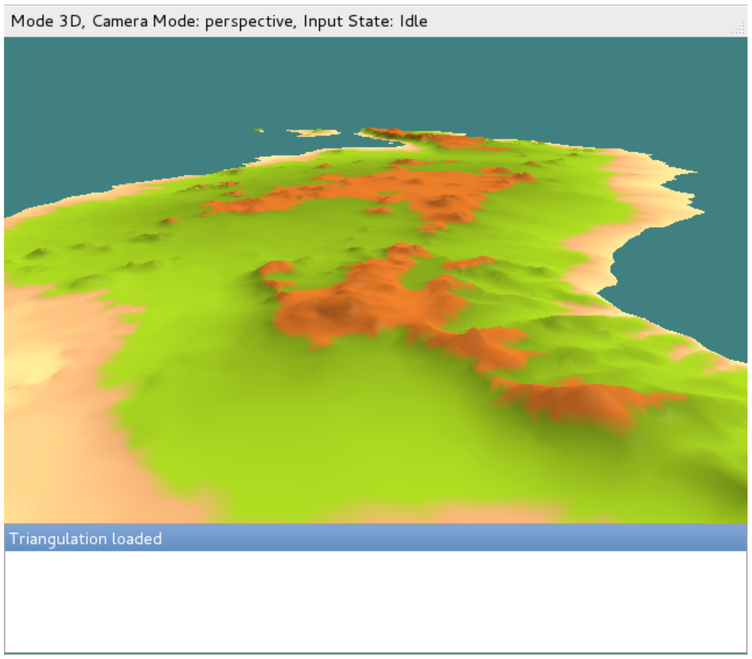
- `KEY_N` 3D gray triangulation (requires *3d* flag). Same as the above, except that the `Z` coordinate is taken into account and the triangles are colour-filled, producing a gray 3D terrain.



- KEY_J 3D colour triangulation (requires *3d* flag). Same as the above, though colour is now added by colouring each vertex based on its height.



- KEY_M 3D colour and smooth triangulation (requires *3d* flag). Same as the above, but with smooth vertex normals, showing smooth transitions between triangle edges.



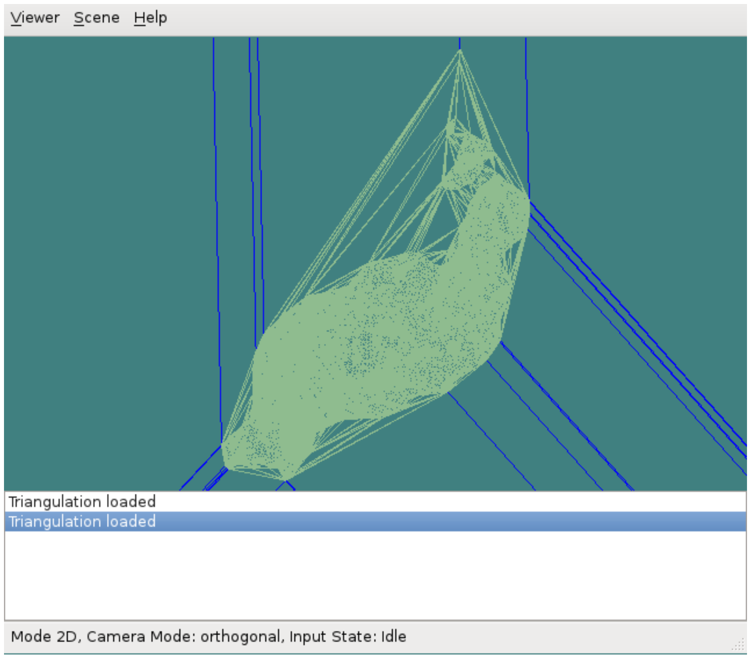
Debugging with CGAL

You can tell the application to triangulate the input set of points using the Delaunay triangulation algorithm that ships with the CGAL library. This will let you compare the output of your algorithm with that of a trusted, working one.

To enable the CGAL triangulation, pass the flag --cgal to the application:

```
$ ./run ex4-5 tests/lanzarote.txt --cgal
```

When your triangulation matches that of CGAL, you should see CGAL's triangulation hiding yours, as shown in the following screenshot:



Triangulation loaded
Triangulation loaded

Mode 2D, Camera Mode: orthogonal, Input State: Idle

Here, the CGAL triangulation appears in green and the student's triangulation (which is hidden, except for the edges connecting to the auxiliary triangle) appears in blue.

Tests and Delivery

The test files for this exercise are the same as for the previous one, and can be found in the folder `viewer/bin/tests/TestLab4-5`.

Even if you are aiming at solving goal (ii) (i.e. handling degeneracies), it is recommended to first implement your algorithms assuming that no degeneracies occur, and test your program thoroughly with the Lanzarote test file that contain points in general position. Only after you are sure that the output produced by your algorithm is correct, extend your code to handle degeneracies and test your program with the remaining test files.

Delivery. Deliver at the *Racó* an archive `lastname_name_5` containing the files that you have modified as well as an updated version of the pdf document that you delivered in Exercise 4. This new version must include an image showing the non-Delaunay version of the `Lanzarote.txt` test file, and another image showing the Delaunay triangulation of the same point set.

5 The Viewer Framework further explained

We shall briefly explain each of the components that come into play in a typical viewer application so that you can build your own for the free lab exercise.

The file `uml/framework.png` contains a UML class diagram of the viewer framework which should be helpful to look at while reading along.

The Output System

The `OutputSystem` class is an aggregation of `OutputStreams`. When a message is sent to the `OutputSystem`, the `OutputSystem` forwards that message to all `OutputStream` objects attached to it. This gives us a centralised access to all attached output streams.

The scroll list in a viewer application is actually an `OutputStream`. Other `OutputStreams` provided by the framework are `ConsoleOutput` and `FileOutput`, which output messages to the console and a file, respectively. You may write your own output stream class by deriving from `OutputStream`.

Input Components: The Object Loader

The class `ObjectLoader` is an aggregation of `Loaders`. We add functionality to `ObjectLoader` by attaching concrete `Loader` objects.

The `Loader` class is a template class taking a single parameter which is the type to be loaded; for instance, `Loader<Point>` would be a point loader.

Any class that you might want to promote as a load-able type must implement certain methods. We will discuss these in Section 7 where we explain how to add your own geometric type to the system. For now, all we need to know is that classes like `Point` and `Triangle` implement these methods to be `Loader`-friendly.

The `Loader` class derives from `Subject`. The `Subject` template class provides a way to handle observers and to communicate with them. `Observer` is also a template class, the template parameter of which is the type to be observed.

To illustrate this, consider `Loader<Point>`. The `Loader<Point>` class would derive from `Subject<Point>`, enabling us to attach `Observer<Point>` objects (or any subclass) to the point loader, in which case the observers would be notified of every `Point` loaded by it.

Input Components: The Screen Input

The class `ScreenInput` represents a state machine that deals with screen input. We add functionality to `ScreenInput` by attaching `ScreenState` objects to it.

The `ScreenState` class is the class all concrete states derive from. For example, `PointState`, which reads points, derives from `ScreenState`.

All concrete states, like `PointState`, also derive from the `Subject` template class. For example, `PointState` derives from `Subject<Point>`, enabling us to attach `Observer<Point>` objects to `PointState` and allowing these to be notified of every `Point` read by `PointState`.

The Scene Manager

The class `SceneManager` handles the scene for us. Class `SceneManager` only knows about `Entity` objects; it does not know about `Point`, `LineSegmentEnt`, or any other concrete type. `Entity` basically represents anything that can draw and write itself.

The `SceneManager` class exposes methods to attach and detach entities to/from it. Attached entities are then drawn on the screen.

A class that wishes to be part of the scene must derive from the `Entity` class and implement its abstract methods. We do this when adding our own geometric types to the system to make the new types `SceneManager`-friendly. This is discussed in Section 7.

Scene Types

You might have noticed that classes like `Vector3` or `Triangle` do not derive from `Entity`. How then, are the applications we have seen attaching these to the scene manager? The answer is that together with the geometric types, the framework provides what we will refer to as scene types.

Remember classes `LineSegmentEnt` and `TriangleEnt` from exercises four and five? These are actually scene types. The scene types provided by the framework are `Point`, `LineSegmentEnt`, `TriangleEnt`, `CircleEnt`, and `TriangulationEnt` (the *Ent* termination stands for `Entity`). A `Point` encapsulates a `Vector3` and makes the vector `SceneManager`-friendly. `LineSegmentEnt`, `TriangleEnt` and `CircleEnt` derive from `LineSegment`, `Triangle` and `Circle` respectively. At the same time, they all derive from `Entity`, promoting the geometric types to scene types and allowing to draw points and triangles on the screen.

The reason for this distinction between geometric types and scene types has been made for performance purposes. Geometric types hold only the data that is necessary to represent a particular type, while scene types have an added overhead to become `SceneManager`-friendly; precisely, each of them holds a bounding box and a label.

If we had not made this distinction, the geometric types would have a significant overhead that would make them impractical. For example, if we made a `Polygon` class represented by a set of line segments, each of the line segments would hold a bounding box and a label, which would certainly not be something we desire.

GeocWidget

The class `GeocWidget` encapsulates the above subsystems except for the output subsystem into a single class. The reason the latter is not included is that we may have several `GeocWidgets` in an application, and in such cases a centralised output is generally preferred.

The class `GeocWidget` exposes methods to retrieve each of its components. Please see *the docs* for more details.

GeocApplication

The class `GeocApplication` is a base class for all viewer applications. This class holds a reference to an instance of `GeocWidget` that is accessible through one of `GeocApplication`'s methods.

To write our own application and add functionality, we derive from `GeocApplication` and override certain `hooks`. A `hook` is a method called when a certain event takes place, such as when the application is initialised or when a key is pressed.

For a full list of hooks and methods exposed by the class `GeocApplication`, check out *the docs*.

Class `GeocApplication` has its own share of pros and cons. The big pro is that we may write an application in just a few lines of code. The con is that the appearance of the application may not be customised. For guidelines on how to write an application from scratch, please see Section 8.

6 The Example Viewer Application: A Close-Up Look

Recall that:

The viewer framework:

- Provides an `object loader` for handling file input.
- Provides a `screen input component`.
- Provides a `scene manager`, with undo and redo functionality baked into the framework.
- Provides a `camera control component`; the controls to manipulate the camera are baked into the framework.

A viewer application:

- Customises the `object loader` by attaching loaders to it.
- Configures the `screen input component` by configuring its states, setting up key bindings in the process; the key bindings used to switch between states are application-specific.
- Provides an observer which communicates with the input components and the scene manager.

With this in mind, let us now examine the very first viewer application we played with in Section 2, that is, the one in the `viewer/demos/example` directory. The file `uml/framework.png` describes graphically the parts of the framework relevant to the following discussion.

The Custom Observer

The example viewer application is able to handle all sorts of geometric types. To do that, it provides an observer which is able to observe each of these particular types. This observer is defined in `MyObserver.h`:

```
class MyObserver :  public geoc::Observer<geoc::Point>,
                    public geoc::Observer<geoc::LineSegmentEnt>,
                    public geoc::Observer<geoc::TriangleEnt>,
                    public geoc::Observer<geoc::CircleEnt>
{
    ...
}
```

The `Observer` class is a template class taking a single parameter which corresponds to the type to be observed. By deriving from all these concrete observers, `MyObserver` is able to handle points, segments, triangles, and circles.

`MyObserver` then defines an `enters(Type* item)` method for each geometric type it observes. When a geometric type is read from a file or from the screen, the appropiate `enters` method is called, notifying the observer that a new geometric object has entered the application.

All `MyObserver` does is attach whatever geometry it receives to the `SceneManager` so that it gets rendered on the screen. Notice that the constructor for `MyObserver` takes a pointer to a `SceneManager` instance.

The constructor also takes a pointer to an `OutputSystem` instance. This is so that `MyObserver` can print a certain message when it receives a new geometric object.

The Custom Application Class

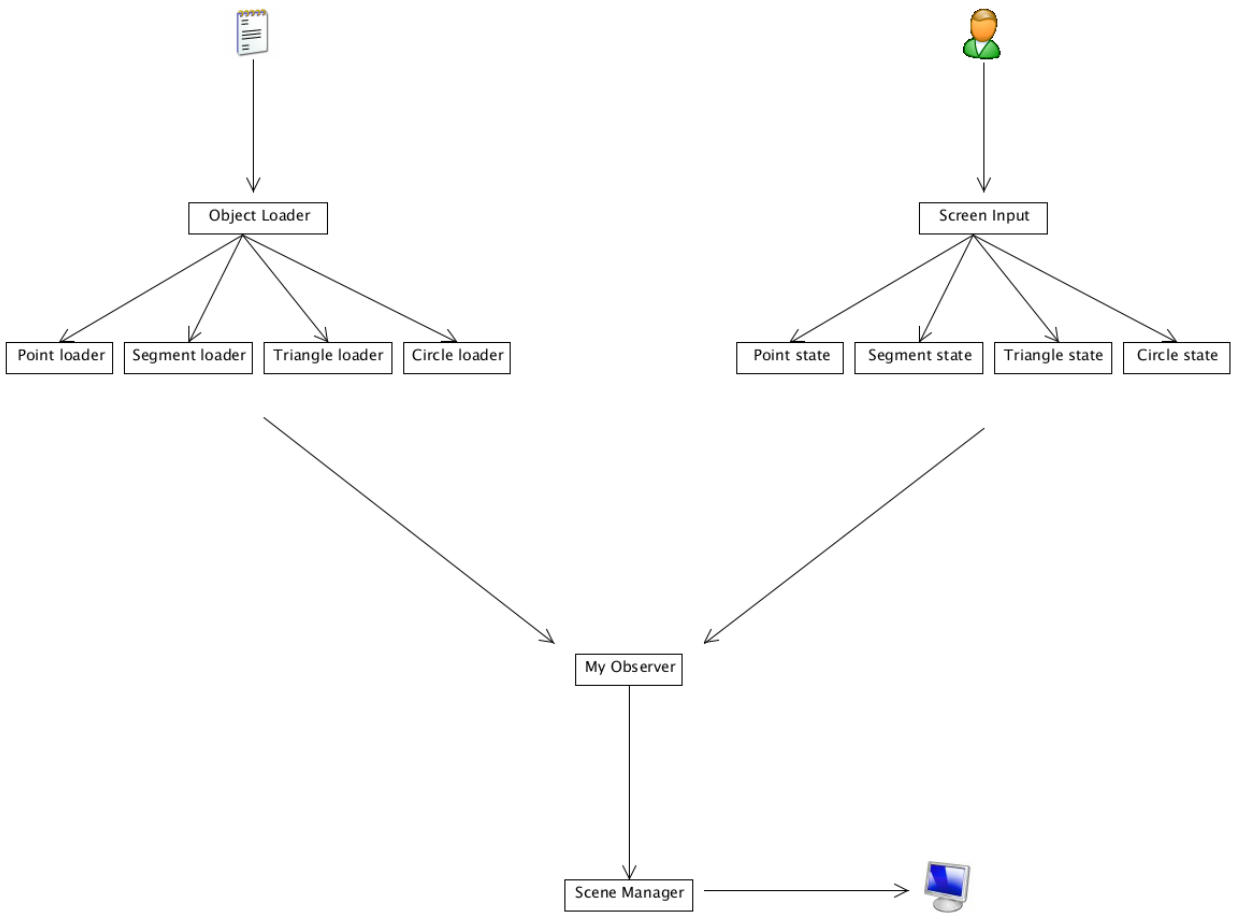
We just said that `MyObserver`'s `enters()` methods are called when geometric objects arrive. This is not going to happen unless we attach an instance of `MyObserver` to the input subsystems. What we do is subclass `GeocApplication` and override the `init` hook, which is called during initialisation. It is there where we wire everything up.

The example application defines a class called `ExampleApp` which derives from `GeocApplication` and overrides the `init` hook. The definitions for class `ExampleApp`'s methods can be found in `ExampleApp.cc`.

In the `init` method, we instantiate machine states and attach them to the screen input subsystem, configuring key bindings in the process. Next we instantiate concrete loaders and attach them to the object loader. Finally, we attach an instance of `MyObserver` to all states and loaders.

Now that the `MyObserver` instance is attached to every state and loader, when a state reads a geometric object from the screen or when a loader loads an object from a file, the appropiate `enters` method is called, notifying `MyObserver` that an object has entered the application.

After initialisation, we end up with the following:



Main

As with any C++ application, we must provide a `main()` entry point. The `main` function can be found in `Main.cc`. Here we instantiate `ExampleApp`, call its `initialise` method to get everything set up, and finally call its `run` method.

7 Adding a Custom Geometric Type to the Framework

The viewer framework allows to add your own custom geometric types to the system without modifying existing code. This section discusses how to do this using the example application in `viewer/demos/polygon`.

The Polygon Application

The `polygon` application integrates a `Polygon` class with the rest of the system. Switch to the `viewer/bin` directory and run the application by calling:

```
$ ./run polygon input/polygon_input.txt
```

The application should display the word `GEOC` on the screen. Each of the letters is an instance of `Polygon`.

You may also add polygons via the screen. Press `KEY_O` to switch to polygon mode and add a polygon with `LMB`. `RMB` closes the polygon.

Let us now discuss each of the steps needed to make this `Polygon` class framework-friendly. Please note that the steps described below are optional. For example, we can make a `Polygon` class that can be loaded from a file but that cannot be written to a file, or one that can be entered via the screen but cannot be loaded from a file, etc. The framework gives us the flexibility to integrate a custom type only as much as we want.

Defining the Polygon Class

We had two choices when defining the `Polygon` class: we could add it to the viewer library, or we could make it only part of the demo application. We chose the latter so as to not bloat the library.

The definition of class `Polygon` can be found in `include/Polygon.h` inside the `viewer/demos/polygon` directory. The definitions for its methods can be found in `src/Polygon.cc`.

A basic definition of class `Polygon` would be the following:

```
class Polygon
{
    std::vector<geoc::Vector3>  points;

public:

    geoc::Colour3   colour;

public:

    void addVertex(const geoc::Vector3& v);
};
```

We represent a Polygon with a vector of points (of type Vector3, not Point). We also add colour to it and a method addVertex, which adds a point to the points vector.

Making the Polygon Class Drawable

To display a Polygon on the screen we make it SceneManager-friendly. To do this, we derive from Entity and implement its abstract methods, which are two:

- draw() Draws the entity.
- bb() Returns the bounding box around the entity.

We now add a BoundingBox member variable to class Polygon and these two methods.

```
class Polygon : public geoc::Entity
{
    std::vector<geoc::Vector3>  points;
    geoc::BoundingBox3          m_bb;   // New in this section.

public:

    geoc::Colour3   colour;

public:

    void addVertex(const geoc::Vector3& v);

    void draw(geoc::Graphics& gfx) const;   // New in this section.

    const geoc::BoundingBox3 bb() const;    // New in this section.
};
```

Notice that the member variable m_bb is of type BoundingBox3. The BoundingBox class is a template class, taking a single parameter corresponding to the dimension of the box. BoundingBox3 is a typedef for BoundingBox<3>, which represents a three dimensional box.

The definitions for each of the methods can be found in src/Polygon.cc. The draw method takes a pointer to a Graphics instance to draw the polygon, and bb just returns a reference to the m_bb member variable.

We also modify the addVertex method to feed new points to the bounding box m_bb so that it is updated to engulf the polygon.

The scene manager requires a bounding box from each of its entities so that the camera can be properly centered around the scene.

Making the Polygon Class Loadable (Loader-friendly)

To make `Polygon` loadable, we need to implement two methods:

- A static `getHeader()` method which returns the header for polygons.
- `read()` to read a polygon from a given input stream.

The *header* is what precedes polygons in an input file. The file `polygon_input.txt` contains a first line reading *polygon* followed by some data. This *polygon* string is the header for class `Polygon`; it tells the `Loader` that what follows after that line is polygon data.

Our `Polygon` class now grows only slightly bigger to incorporate this new functionality:

```
class Polygon : public geoc::Entity
{
    std::vector<geoc::Vector3>  points;
    geoc::BoundingBox3          m_bb;

public:

    geoc::Colour3   colour;

public:

    void addVertex(const geoc::Vector3& v);

    void draw(geoc::Graphics& gfx) const;

    const geoc::BoundingBox3 bb() const;

    static const char* header();     // New in this section.

    static void read(std::istream& is, Polygon& p); // New in this
                                                    // section.
};
```

The definitions for these two new methods can be found in `src/Polygon.cc`. Since a `Polygon` may have any number of vertices, the way `read` works is that it first reads an integer N and then reads N vertices from there. The static method `header` returns the C string *polygon*, matching the string found in the input file.

Once these methods have been implemented, we can instantiate a `Loader<Polygon>` and attach it to the object loader, making the object loader able to load polygons.

Making the Polygon Class Writable

Recall that `KEY_G` orders the `SceneManager` to dump the contents of the scene into a file. For our `Polygon` class to be writable, we overload the following methods derived from `Entity`:

- A non-static header() method which returns the header for polygons.
- A write() method which writes a Polygon to a file stream.

Our Polygon class now resembles the following:

```
class Polygon : public geoc::Entity
{
    std::vector<geoc::Vector3>  points;
    geoc::BoundingBox3          m_bb;

public:

    Colour3 colour;

public:

    void addVertex(const geoc::Vector3& v);

    void draw(geoc::Graphics& gfx) const;

    const geoc::BoundingBox3 bb() const;

    const char* getHeader() const;      // New in this section.
    static const char* header();

    static void read(std::istream& is, Polygon& p);
    void write(std::fstream& fs) const; // New in this section.
};
```

The new, non-static getHeader method just calls the static one to avoid code duplication.

The write method writes a Polygon to a file; it first writes the number of vertices in the Polygon, followed by the vertices themselves. In general, it is a good idea to write data in the same format that it is expected to be read so that the scene can be saved and later reloaded.

Reading Polygons from the Screen: Making a custom InputState

The file PolygonState.h defines the PolygonState class, which derives from ScreenState<Polygon> and overrides a few methods to get polygons read from the screen.

```
class PolygonState : public geoc::ScreenState<Polygon>
{
    std::list<geoc::Vector3>    vertices;

public:

    GEOC_APP_REQUEST_CODE mouseClick(const geoc::Vector3& pos);
    GEOC_APP_REQUEST_CODE mouseRightClick(const geoc::Vector3& pos);
    GEOC_APP_REQUEST_CODE mouseMove(const geoc::Vector3& pos);
    void cancel();
    void draw(geoc::Graphics& gfx);
    const char* description() const;
};
```

Remember that ScreenState derives from Subject. In this case, PolygonState derives from ScreenState<Polygon> which in turn derives from Subject<Polygon>. This allows PolygonState to communicate with Observer<Polygon> objects. For a graphical description, see uml/framework.png.

The only member variable PolygonState has is a list of vertices to keep track of the points the user has inserted when entering a polygon.

The definition for each of the class' methods can be found in src/PolygonState.cc.

Methods mouseClick, mouseRightClick, and mouseMove are called when LMB is clicked, RMB is clicked, and when the mouse is moved, respectively. Each of these methods returns a GEOC_APP_REQUEST_CODE, defined in src/geoc/app/update_requests.h as shown below:

```
enum GEOC_APP_REQUEST_CODE
{
    GEOC_APP_NO_REQUEST,         // No action.
    GEOC_APP_REDISPLAY,          // Redraw the screen.
    GEOC_APP_STATUS_BAR_UPDATE  // Redraw the status bar.
};
```

On every LMB click we add a point to the list of vertices. On a mouse move we update the position of the last vertex entered (which is the first one in the list since we push vertices to the front of it). When we receive a RMB we add the last vertex, construct a new Polygon, and send it to the observers attached to the PolygonState by calling the method enters(Polygon* p), which is derived from Subject<Polygon>.

The cancel method is called when ScreenInput switches state; this gives us a chance to forgive any progress that we may have made. In this case we empty the list of vertices.

The description method returns the string that is displayed on the status bar.

The draw method is called on every redisplay. Here we draw a point if we only have a vertex in the list, and a list of segments in any other case.

The Polygon Application

The rest of the polygon application should look pretty familiar, so we won't discuss it in detail. Basically we derive an ExampleApp class from GeocApplication again and override the init hook to set up all subsystems. We also define a class MyObserver, which observes Polygons and attaches them to the SceneManager.

8 GUI Programming

So far we have discussed the framework's internals but we have left out a rather important topic towards building viewer applications: GUI programming.

The `GeocApplication` class lets us write simple applications quickly, but it is very prohibitive when it comes to customising the user interface. As it turns out, the class `GeocWidget`, the one `GeocApplication` has an instance of, is actually a Qt widget.

Remember that the `GeocWidget` class is basically an aggregation of the following components:

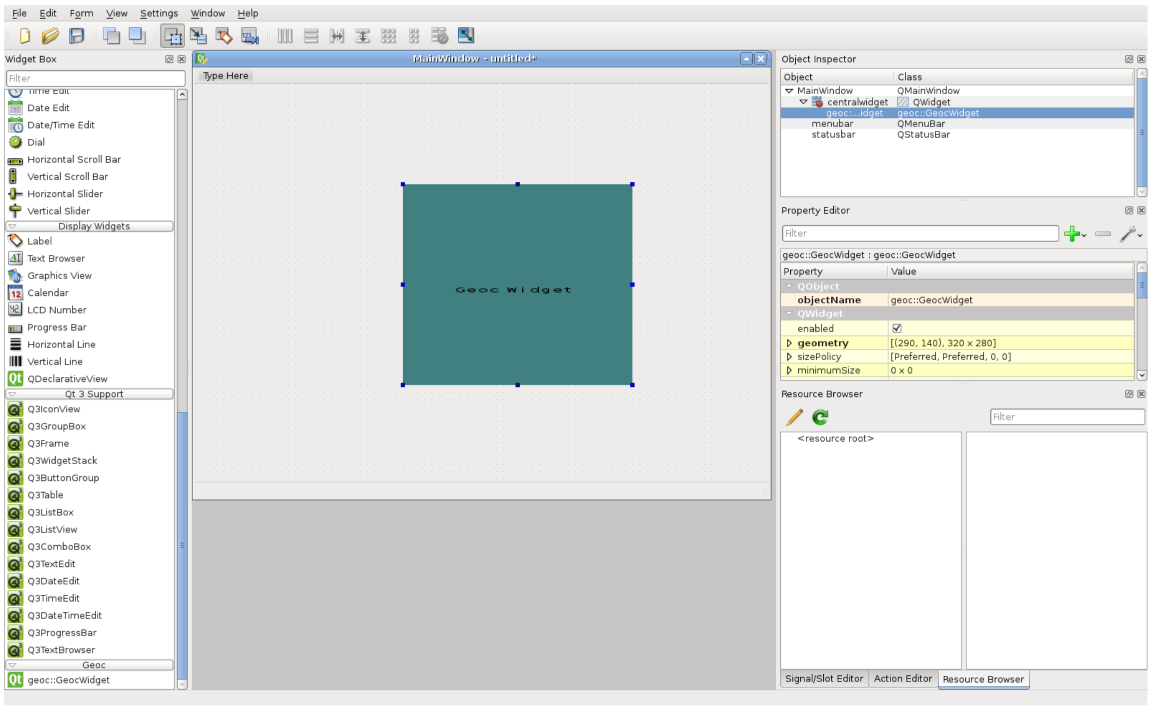
- SceneManager
- Camera
- CamCtrlContext
- Graphics
- Input
- ScreenInput
- ObjectLoader

Because the `GeocWidget` class is a Qt widget, we can use Qt Creator or Designer to build a complex GUI and then throw an instance of `GeocWidget` into the project and get access to all of the aforementioned components.

To enable `GeocWidget` in Qt Creator/Designer, we must set the `QT_PLUGIN_PATH` environment variable. In order to avoid having to copy files around, it is a good idea to set this variable to the `viewer/bin` directory where the viewer runtime library resides. The variable may contain several paths, all separated by a system-dependant symbol (`:` on Linux, `;` on Windows).

Once the variable is set, Qt Creator and Designer should detect and load `GeocWidget`, making it avaiable in the widget pane as shown below.

We can then create a window and drag an instance of `GeocWidget` onto the window we just created.



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
geoc:...idget	geoc::GeocWidget
menubar	QMenuBar
statusbar	QStatusBar

geoc::GeocWidget : geoc::GeocWidget

Property	Value
QObject	
objectName	geoc::GeocWidget
QWidget	
enabled	☑
geometry	[(290, 140), 320 x 280]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0