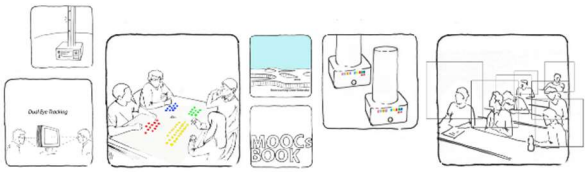
CHILI LAB

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Cellulo Learning Activity

Semester project report



Bastien Beuchat & Andrea Scalisi

2019-01-11

Acknowledgments

First of all we would like to thank all the people from the CHILI lab that helped us to accomplish this semester project. We are grateful to Pierre Dillenbourg, who allowed us to do this project in his lab, and Ayberk Özgür who supervised us, always had answers to our questions, and gave us good advices. We sincerely thank Wafa Johal for supervising us and being available during the whole semester, and Arzu Özgür Güneysu for her attendance and opinion at our first meeting. We finally thank Florence Colomb who allowed us to have access to the lab during the holidays, and Sergei Volodin, whom we have never met but whose work has made it possible to add augmented reality to our project.

Table of contents

1 Introduction

For our bachelor semester project, we decided to create a new learning activity with the Cellulo robots. We had a few different ideas, but after a couple of meetings with Wafa Johal and Ayberk Özgür, the following idea came out : a geometric plug-in which will allow to create interesting situations with the robots, involving shapes and constraints.

Such a plug-in would be really useful for school teachers for example. Indeed, children are taught geometry at school, but it is known that mathematics are not the funniest activity in the world from the point of view of schoolchildren. However, when it comes to play with robots and other technological stuff, everything becomes more interesting. By combining both, we strongly believe that their curiosity could be awaken, and that it could lead them to make their own experiences with geometric shapes, which would develop their instinct and comprehension in this domain.

The goal of our project was then defined : find an already existing geometric tool that allows to create shapes and add constraints to them in two dimensions, and adapt it to make a simple usable plug-in for the Cellulo robots.

This is why the first step of our project was to find an open source geometric tool.

2 SolveSpace

It was not easy to find a convenient geometric tool. We had the following criteria : open source, light, implemented in C/C++, allows 2D geometric constraints. We spent a quite important amount of time trying several different programs.

We eventually found a tool named SolveSpace, which seemed promising.

2.1 What is SolveSpace ?

By its definition, *SolveSpace is a free and open source 2D and 3D computer-aided design program. It is a constraint-based parametric modeler with simple mechanical simulation capabilities.*[1]

2.2 Why SolveSpace ?

As you may have noticed, the definition of SolveSpace matches exactly what we needed. It is implemented in C++, which means that it is easily compatible with Qt to integrate it with the robots. Moreover, it provides its constraint solver in a library.

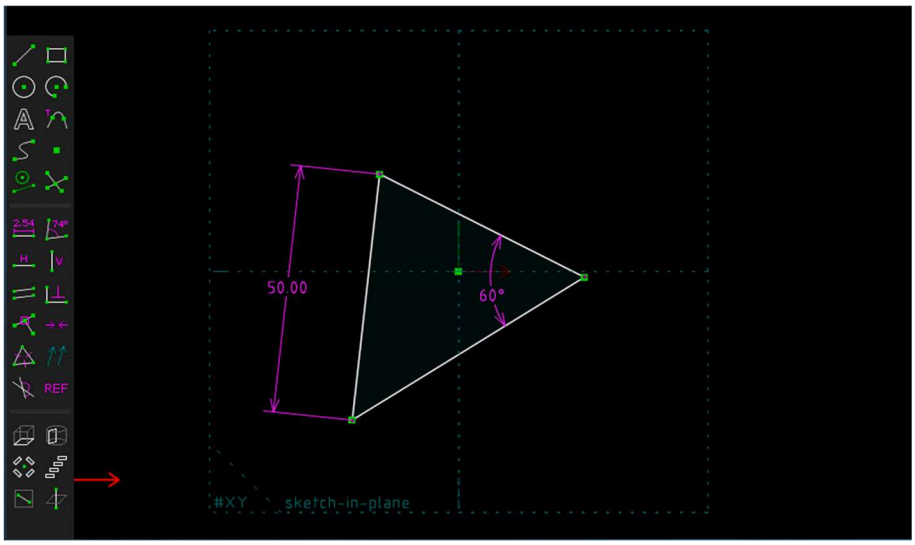
2.3 How does it work ?

For our project, we really care only about the two dimensional part of SolveSpace. When the program is launched, a simple editor which looks like this is displayed :

[1] Definition taken from : https://en.wikipedia.org/wiki/SolveSpace

Insert entities to the situation (points, segments, etc.)

Add constraints to the entities (angles, distance, parallel lines, etc.)



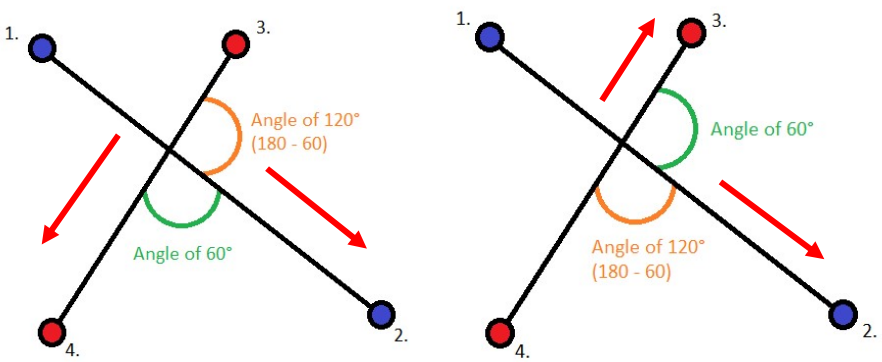
In this example, we can observe three points, three segments, and two constraints : one distance constraint on a segment (i.e. a distance constraint between two points), and an angle constraint between two segments.

As you can see, SolveSpace allows us to add lots of points and segments (referred as “entities”), in order to create geometric shapes such as lines, triangles, quadrangles, etc. and then add constraints on them. This means that lots of interesting geometric situations can be created, and the points and segments can be moved to see how the shapes behave under the constraints. The solver computes a new solution to the problem every time an entity is moved in the plan.

3 Limitations

As said earlier, SolveSpace is simple. This means that users can add as much constraints as they want to a situation, which can make the problem unsolvable. Another limitation is the fact that when adding an angle constraint between two lines, the order in which the points have been added to each line matters for SolveSpace. Imagine that a line is a vector, going from the first point to the second point that has been added to the line. Then, when SolveSpace adds an angle constraint between two lines, the angle is set as an angle between two vectors.

Here are two examples. On both cases the two same lines were created, but the points were not added in the same order. On the left, the two lines were created like this : : line1(point1, point2) and line2(point3, point4) and on the right, they were created like this : line1(point1, point2) and line2(**point4, point3**). Then, in both situations, an angle of 60° is added between those two lines. SolveSpace will set the constraint like this :



3.1 Consequences for our plug-in

The first limitation mentioned above means that, by using our plug-in, users can create unsolvable situations, in which case the solver will simply stop trying to solve the problem. But in this case, we can give the user a feedback, and thus this limitation is not a real issue. However, the angle assignment could be counter-intuitive for some users. This is why any developer that will implement activities with our plug-in should keep this limitation in mind when designing his/her program.

4 Integration to the project

In order to make SolveSpace work with Cellulo, we indeed had to create our own C++ classes to adapt and use the code from the SolveSpace library. This was the most challenging part of this project, and this paragraph explains the structure of the plug-in we created, with a few details about the decisions we had to make.

4.1 Structure of the code

As you may have understood, a geometric situation in SolveSpace is a system, in which you can add entities and constraints. We chose a structure for our plug-in based on the actual structure of SolveSpace[2]. To make it consistent, we decided to add the prefix “CelluloGeom” to every class. Then, in our plug-in, the system is represented by a “CelluloGeomSystem” object, the constraints are represented by “CelluloGeomConstraint” objects, and the entities are sub-divided into two groups : “CelluloGeomPoint2D” and “CelluloGeomLine”.

Every entity and constraint has a “geomId”, which is a unique identifier used both by SolveSpace and by our plug-in.

Here are basic explanations about the behaviour of the classes.

4.1.1 CelluloGeomPoint2d

A CelluloGeomPoint2D simply represents a point in two dimensions. A point needs two parameters, X and Y which correspond to its coordinates.

4.1.2 CelluloGeomLine

A CelluloGeomLine represents a line, made of two already existing CelluloGeomPoint2d.

4.1.3 CelluloGeomConstraint

A CelluloGeomConstraint represents a constraint, which has a value, a type, and references to the entities to which it applies. It can be two points, two lines, or one point and one line. All the different types of the constraints are defined in an enum in the header file.

[2] Please refer to the document « solvespace_library.txt » in the root folder of the plug-in

4.1.4 CelluloGeomSystem

The system itself. This is where all the links with SolveSpace are made. The CelluloGeomSystem contains a reference to a SolveSpace system, arrays with CelluloGeomLines, CelluloGeomPoint2d, CelluloGeomConstraint, and some maps that link objects from the CelluloGeomSystem to the real SolveSpace system.

All the Q_INVOKABLE methods are implemented in this class. That means that a QML developer will only use methods from this class when using our plug-in.

To sum up everything, the CelluloGeomSystem class invoke the solver method from SolveSpace, and then maps the results from SolveSpace to the internal representation of the system.

4.2 Interaction with the robots

Once the C++ side of the plug-in was written, we had to bind it with the robots. This paragraph explains how we did it in our examples in QML and why we think it is the most suitable way of doing it.

Even though we remembered that Pr. Dillenbourg told us during his class “Introduction à l’informatique visuelle” that *“intuitive” doesn’t really exist, something that we find “intuitive” is something that we have previously learned*, we still wanted to find a way to interact simply with the robots to make the system feel like it is intuitive.

First of all, we decided to map robots and points together. Hence, the coordinates X and Y of a robot correspond to the coordinates X and Y of a CelluloGeomPoint2d.

Then, a line is simply composed of two robots. We did not find useful to make it possible to add more than two robots on a line since a line is mathematically defined by two points. Moreover, robots belonging to the same line can be identified with the same colour, and it is possible to add augmented reality to our plug-in[3].

Finally, we had to decide how the user can inform the system that he desires to move an entity. We chose the following principle : if a robot is simply moved by the user, then it will go back to its place. If the user wants to modify the position of the robot (i.e. the position of a point), he has to press any button of the robot while moving it. Doing so, it updates the system which is solved by SolveSpace, and the other robots start moving to maintain the constraints of the system.

There are two reasons for this choice, the first one being that pressing a button ensures that the user really wants to modify the system. The second reason is more technical. Indeed, we need to know which robot is moved by the user. Otherwise, since all robots would update their position every time they move, there would be a conflict : they would try to go to their new position computed by SolveSpace and at the same time SolveSpace would try to re-compute a solution with the new positions of all the robots, which would cause an infinite loop and would lock all the robots.

[3] Please refer to the last paragraph of this section for more details

4.3 Make it a plug-in

Another thing we had to do with our C++ code was to make it a real installable plug-in. Indeed, we do not want future users to have to include all our C++ classes in their CelluloGeom projects[4].

To do so, we had a look at the others QML plug-ins of the Cellulo project and created a QT plug-in that wraps our classes. Our plug-in is called "CelluloGeomModel" and it involved the creation of two files : "cellulogeommodel_plugin.cpp" and "qmldir" which configure the plug-in when we install it with the command "make install".

Then, once it is installed on a computer, our plug-in can be used in any Cellulo project simply by adding the following line : "import CelluloGeomModel 1.0".

4.4 Compatibility

While developing the plug-in, we tested it on its desktop version (i.e. on the computer). Naturally, in order to add more interesting features as augmented reality and to improve its portability, we also wanted it compatible with Android.

We have to admit that we had a mini hearth attack when, half-way of the project we realized that we did not know if SolveSpace was compatible with Android. But luckily for us, life is fair and it was not too complicated to make it work on Android.

Hence, our plug-in works both on desktop and Android, and is indeed compatible with the Pool and the Hub developed for the Cellulo project.

4.5 Augmented Reality

As said earlier, we managed to make our plug-in work with augmented reality[5], thanks to the already existing plug-in "qml-ar" developed by Sergei Volodin.

This feature is really useful to visualize shapes created by the robots. In our samples, an AR component is created for each line in the system. An AR line is a 3D rectangle with a small height and width, its length is simply the distance between the two robots (i.e. points) composing the line, and its orientation is computed with the angle of the line relatively to the X axis. Labels for lines, points and angles are also displayed in AR.

However, we noticed an improvement that could be made with the augmented reality and this will be discussed in the section "Conclusion : possible improvements".

[4] You can find the installation procedure in the README of our plug-in

[5] Later referred as AR

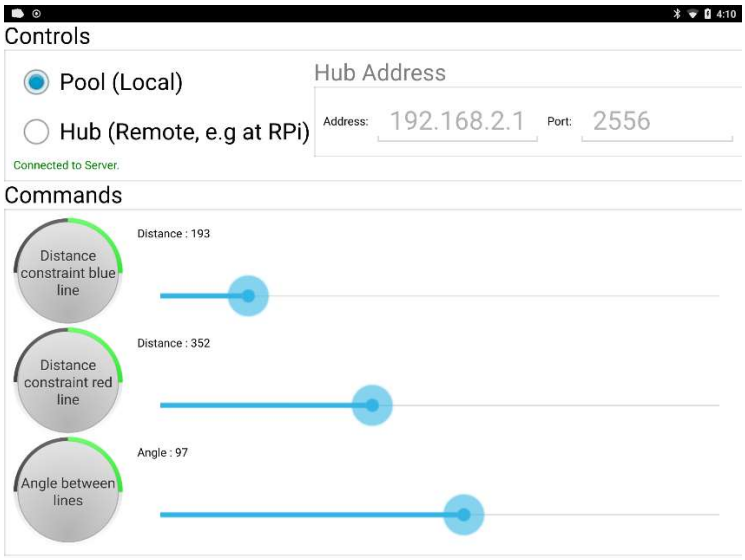
5 Activities based on the plug-in

Since the theme of this project is “Learning Activity”, we had to provide a real learning activity in addition to the plug-in. We made three different samples[6], which show the possibilities of our plug-in.

5.1 Simple demo

The first one is called “cellulo_constraints” and can be run both on desktop and on Android. In this demo, there are four robots which form two lines. The lines are identified by their colours : one is made of two robots with red lights, and the other is made of two robots with blue lights.

On the control panel of the application, there are three toggle buttons that can activate or deactivate three constraints : the length of the blue line, the length of the red line, and an angle between the two lines. Each constraint has a slider, which allows to adjust its value. Then, the evolution of the system can be observed when the robots are moved, depending on the number of constraints enabled and their values.



The interface of « cellulo_constraints »

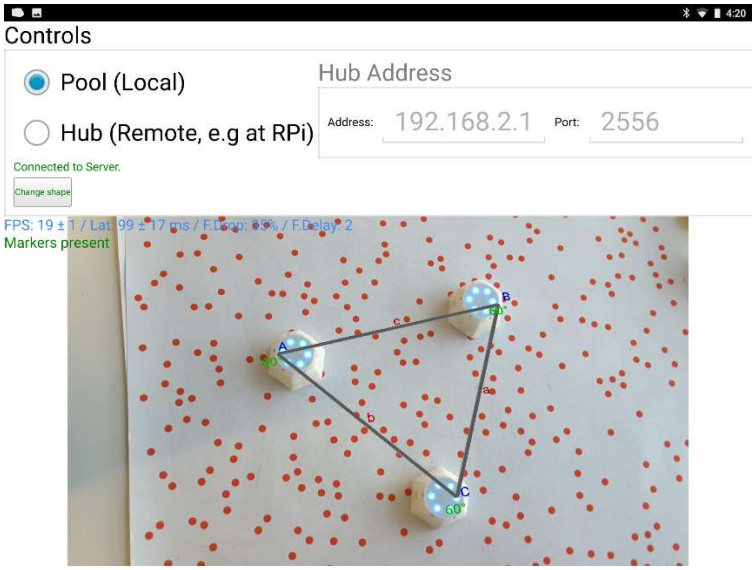
5.2 AR demo

The second one is a basic demo to show how augmented reality works with our plug-in. Since it needs the camera, it is better suited for Android but could be run on a computer with a camera. It is called “ar_cellulo_triangle”.

It is composed of three robots that form a triangle, which is rendered in AR on the screen. There are initial constraints on the triangle : two angles of 60 degrees, which forces the third one to be 60 degrees as well and hence make the triangle equilateral. Since there are no length constraints on the sides of the triangle, the robots can be moved to see how the triangle remains proportional.

We added a button called “Change shape”, which modifies the angle constraints to the following values : 90 degrees and 45 degrees, which forces the third angle to be 45 degrees as well. This button has for purpose to show how the AR shape can evolve relatively to the system.

[6] The samples can be found in “Samples” in the root folder of the plug-in



The interface of « ar_cellulo_triangle »

5.3 Editor with AR

The third demo, called “ar_cellulo_editor” is the most complex one, gives the user freedom, and should be considered as our official learning activity. As its name suggests, it is an editor which allows the user to create custom geometric situations.

There are just points at the beginning, indicated by robots with white lights. Up to twelve robots can be added. They can be selected by clicking on them on the screen. When a robot is selected, it turns red. Then, by clicking on the button “Create shape”, it will create a shape depending on how many robots have been selected. Once the shape is created, all its lines will be rendered in augmented reality and all the robots of this shape will be lightened with the same random colour, excluding red or white, since it is already referring respectively as a selected robot or as a point. The lines rendered in AR have the same colour as the robots they are attached to.

For example, if two robots are selected, a click on “Create shape” will add a line to the system. The two robots will become, say green, and a green line between those two robots will be rendered in AR. Then, three other robots could be selected and a click on “Create shape” will create a triangle (i.e. three lines). The three robots will become, say blue, and three blue lines will be rendered in AR.

Please note that creating a shape with one robot is useless and not possible, since a robot is already a point.

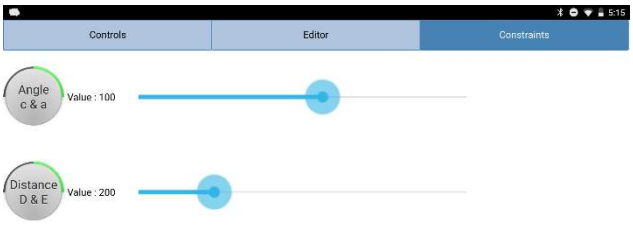
Obviously, the button “Add constraint” adds length or angle constraints to the created shapes.

To add a length constraint, two robots have to be selected on the screen. Then, by clicking on “Add constraint”, a length constraint will be automatically added between those two robots. It will open a new tab on the app, which allows the user to activate, deactivate, or modify the value of the freshly added constraint (as in the “cellulo_constraint” demo). Please note that it also works with robots that are just points and not necessary previously added in a shape.

To add an angle constraint, two lines must be selected by clicking on their AR component on the screen. Then, clicking on “Add constraint” will automatically add an angle constraint between those two lines, that can be tuned in a similar way to the length constraints.

We decided to let users freedom, and not checking that constraints are coherent between them, as SolveSpace does. The user will just be notified when the system is unsolvable, and the initial situation can be reloaded simply by clicking on the button “Reset”.

The tab « Editor » of « ar_cellulo_editor », with two shapes and one selected robot



The tab « Constraints » of « ar_cellulo_editor », that opens automatically when you add a constraint

6 Conclusion

6.1 Final product and utility

Our plug-in is, from our perspective, really complete, and could be exploited to create lots of different geometric activities.

Our “ar_cellulo_editor” activity could already be used by school teachers, who want either to create situations to give children instinct, or to test the ability of children to create specific situations according to the teacher’s instructions.

Indeed, we believe that it is more entertaining and motivating for a children who is asked : “Can you create an equilateral triangle ?” to play with robots and a tablet, and then be able to tangibly test their construction by moving the robots, than just drawing a triangle on a sheet of paper.

6.2 Possible improvements

Earlier, we mentioned a possible improvement with the augmented reality. It is not directly related to our plug-in, but it is related to its association with the augmented reality. Having AR components rendered, in addition to multiple calls to the solve function of SolveSpace when a robot is moved, make the overall computation heavy. This has for a consequence to make the application slow. This problem could be solved by executing all the SolveSpace computations or AR rendering in another thread.

Another aspect that could be explored is using more different constraints. Indeed, a CelluloGeomConstraint has 34 different types of constraints, and we clearly did not use all of them in our samples.

Eventually, new types of entities could be added to the plug-in, such as “CelluloGeomCircle” for example.

6.3 Personal feedback

During this project we experienced situations in which we had to be really innovative, as there were no answers to our questions on the internet. Indeed, it was the first time in our academic background that we had to really do something that nobody has ever done before. It taught us that with perseverance, almost every problem has a solution, or in the worst case an alternative.

The development of the activities also stimulated our creativity. The feeling of building something real and useful, that could be used in the real world, was really motivating, and gave us a preview of what it is like to live the life of a computer engineer.