Geometric Modelling with Constraints

[Extended Abstract]

Fábio Miguel Gonçalves Pinheiro
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
fabiopinheiro@tecnico.ulisboa.pt

ABSTRACT

Geometric modeling requires the specification of position, orientation and size of geometric entities. However, the modeler often does not know *a priori* these properties, but knows other properties that can be used to bound and solve the first. These constraints must be calculated in order to discover the missing properties. Currently, modelers make this process or part of it manually, making it time consuming.

This document presents the tools most widely used for geometric modeling in the area of generative design, through programming, containing functionality for specifying relationships and constraints between geometric entities, and that automate part of the geometric modeling process. In this work we propose a solution which automates the entire process of calculating the geometric properties from constraints. The proposed solution addresses the problem of values accuracy, functionality expansion and integration with other tools.

Our geometric constraints solving system is based on transforming the geometric model into sets of mathematical equations that describe it, which are solved by Maxima, an algebraic computing systems capable of symbolic manipulation.

Our solution is intended to assist modelers with the mathematical complexity of the geometric modeling, helping them to follow the geometric construction modeling approach using their preferred tools.

Keywords

Macro; Maxima; Geometric Modelling; Geometric Constraints; Racket; Rosetta

1. INTRODUCTION

The geometric modeling requires the specification of the location, orientation and size of geometric entities. However, modelers often do not know *a priori* these properties, but know others that can be used to constraint and solve the first. The computer-aided design (CAD) tools cover most of users's requested needs in this area, however there is still a great need for functionality that allow making more specific modeling.

Nowadays, many modelers manually calculate the specifications of geometric entities, which becomes very time consuming, because the tools used provide a limited set of features in this area.

Geometric constraints can also be used *a posteriori*. I.e., a set of geometric constraints can be used to automatically validate a geometric model. The building information modeling (BIM) tools are indicated to impose such constraints on the geometric model. For example, by law, the stairs may not have a slope greater than a predetermined threshold, and each step must be less than a certain height.

Modelling by specifying geometric constraints is very useful as it enables the modelers to express on the model, without having to go through the lengthy process of finding the location, orientation and size of geometric entities.

These constraints must be processed in order to discover the properties missing. Currently, modelers make this process or some of it manually, making it time consuming.

This document focuses on the using of restrictions *a priori* as a way of modeling, allowing users to make a modeling approach for geometric construction.

For example, considered the creation of a pentagon as described in article [1], that uses the traditional geometric construction approach to create the model illustrated in Figure 1.

The usefulness of this type of modeling (geometric constraints) is overlooked by CAD applications which means that you spend your time thinking about how to model the problem. However, a declarative modeling by geometric constraints specification could facilitate the process of modeling, assisting the designer expressing their intentions on the geometric model.

1.1 Goals

The purpose of this work is to create a system that allows the user to geometric modeling by specifying geometric constraints in a programming language. The secondary goal is to integrate our system with the system developed in the Rosetta project in order to extend the descriptive power of geometric modeling of Rosetta with the specification of geometric constraints.

It is intended to allow the end user to model using a geometric construction approach. This way the user can describe relationships and constraints between geometric entities, without their having to be the fully pre-defined. This way, we intend to withdraw from the user the mathematical complexity involved in geometric modeling.

The integration with Rosetta offers the user the possibility of combining this type of modeling approach with the approach and functionality provided by their preferred CAD tools. This allows us to focus on offering the user the possibility to create models by describing geometric restrictions.

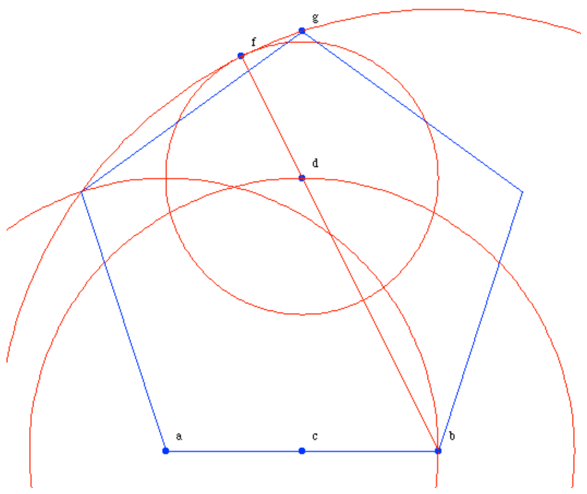

Figure 1: Geometric construction of a pentagon with ruler and compass. Image taken from [1].

In addition it enables us to visually demonstrate our system capabilities.

Taking into account the secondary objective, it is assumed that most users are architects with reduced programming knowledge. Therefore, it is intended to make the created features to describe geometric constraints simple to use and easy to understand. However, it is also intended to provide users with programming skills the possibility of extending our system with new functionality, in an easy and uniform way that can be easily shared with other users.

2. RELATED WORK

This section analyzes the state of the art related to geometric constraints, modeling languages and architectural design tools, including concepts involving geometric constraints.

2.1 Constraint Satisfaction Problem

The Constraint Satisfaction Problem (CSP) is a mathematical problem (definition 1) consisting of a set of objects whose state has to satisfy all constraints, that is, to find a value for each variable (set V), from the values associated with these variables (set D), and where the restrictions (set C) specify that certain values can not be used together.

Definition 1. *The CSP is defined as a triple $P = (V, D, C)$ [2, 3], where:*

- $\boldsymbol{V}$ *= v_1, ..., v_n is a **set of variables**.*
- $\boldsymbol{D}$ *= D_1, ..., D_n is a **set of the respective domains of values**, i.e., I represents the values that are associated with v_i and that this variable can take.*
- $\boldsymbol{C}$ *= c_1, ... c_m is a **set of constraints**, where each $c \in C$ is a subset of the Cartesian product $\Pi_{i \in V_C} D_i$, where Π is the product operator and $V_C \subseteq V$ is the set of variables into c.*

The CSP is a highly combinatorial problem and usually NP-Complete.

2.2 Degree of Freedom

The Degree of Freedom (DoF) of a CSP refers to the number of possible solutions of the CSP. One CSP can be Over-Constrained, Under-Constrained or Well-Constrained.

There are also some more generic problems that CSP, such as the *Constraint Satisfaction Optimization Problem* (CSOP) and the *Partial Constraint Satisfaction Problem* (PCSP).

The CSOP is an optimization problem whose objective is to find the solution or solutions that have a minimal cost. A typical example is the scheduling problem with the minimum of free time.

The PCSP is more generic that the CSOP, and is usually applied in a scenario where the DoF is *over-constrained.*

2.3 Techniques to solve CSP

[4, 5] presents five different methodologies to solve CSP, those are:

- **local propagation**: This methodology represents the CSP in a non-directional graph, used to propagate the geometric constraints. However, the graph can not contain cycles.
- **numerical constraint:** This methodology represents the CSP with a system of algebraic equations and tries to find solutions. The algorithms used are able to cope with large nonlinear systems, but assume that the CSP is fully restricted, and that each variable can assume only a finite set of values.
- **symbolic constraint:** This methodology represents the CSP with a system of equations and use of symbolic manipulation methods to solve them. This technique can conclude that the solution exists but is unable to find, or require time and exponential memory, according to the complexity of the CSP.
- **rule constructive:** This methodology represents the CSP with facts and rules that it uses to create a procedural test to find solutions, similar to Prolog. However, the algorithms used are not computationally scalable and thus are unable to handle large restriction systems [6].
- **graph constructive:** This methodology consists of two stages and represents the CSP with a graph. In the first phase, the dependencies between the nodes of the graph are analyzed so that the problem can be decomposed into several simpler sub-problems called *clusters*. The solution of the initial problem is the combination of the solutions of these clusters, which are usually independent or contain few dependencies between them. In the second phase the *clusters* are formed and individually solved using other techniques (numerical resolution constraints and symbolic manipulation).

2.4 Robustness and Accuracy

The lack of robustness and accuracy is a well known problem in the field of geometric modeling [7]. This happens because many of the geometric modeling algorithms are based from computer graphics algorithms, where the computation time is more important than the accuracy of the geometric model. However, the geometric modeling results can be reused for subsequent calculations, whereas in computer graphics results are mainly used for visualization.

There are a set of good practices to avoid problems with the accuracy of the values. For example, to compare the distance between two points, we do not need to calculate the distance itself (Formula 1). We can compare the square of the distance (Formula 2) points to the origin, avoiding the calculation of square roots.

$$\sqrt{x_a^2 + y_a^2 + z_a^2} < \sqrt{x_b^2 + y_b^2 + z_b^2} \tag{1}$$

$$x_a^2 + y_a^2 + z_a^2 < x_b^2 + y_b^2 + z_b^2 \tag{2}$$

For example, above we presented the constraint of point $A = (x_a, y_a, z_a)$, it is closer to the origin than the Point $B = (x_b, y_b, z_b)$, and to describe the constraint we compare the distances from each point to the origin. Unfortunately, this involves the calculation of square roots (Formula 1), which may produce inaccurate results. However, to compare the two distances we do not need to calculate the distance itself. As good practice, we should have compared the square of the distance (Formula 2) from the points to origin, avoiding the calculation of square roots, since this operation may involve rounding and inaccurate representations of values.

2.5 Modeling Tools

There are a lot of tools for geometric modeling. Recently there has been a constant need for tools that support programming, generative design and functionality to describe their geometric models with constraints. However, most of the tools used in this field still require the user to have a programming background, therefore they are not suitable for those starting to program. This represents an initial barrier that makes many architects give up at the beginning.

Within the analyzed tools we emphasize:

- **Grasshopper** [8] for the initial simplicity of modeling due to visual programming, expandability with acomponente enabling incerir code C# and VB.
- **Eukleides** [9] for the ability to impose geometrical restrictions on bidimencional model.
- **ThingLab** [6] for the ability of the model's information to flow in all directions, completing missing information in the geometric model from the remaining information.
- **GeoSolver** [10] for its CSP solving ability, applied to geometric constraints.
- **Tikz** [11] the modeling capacity (2d mainly), modeling by geometric construction, and extensible through macros.
- **CGAL** [12, 13] for the robust algorithms library for geometric modeling.

3. ARCHITECTURE

The **end-user** just wants to use the functionalities of the language, using or not extensions. This users just need to know the basic geometric entities and the high level operations to define constraints in the model.

The **advanced user** might want to expand the domain language. We created macros to help them expand the domain language, standardize and automate the integration of several modules, thus ensuring that the expansions of different users work together.

In particular, the advanced user needs to understand the solution's architecture, especially the part that he is expanding. For example, to add geometric entities, he needs to understand the macros entity-type and entity-build. If he add geometric functions to the BFG must use the macro mask, and if you want to add features and develop new types of geometric constraints have to understand how the mathematical problem is generated.

The solution comprises two alternative solvers that solve the geometric constraints differently, showing advantages and disadvantages relative to one another.

The solver called **Geometric Function Library** is based on a classical approach shared by most of the analyzed tools.

In this approach, geometric problems are solved by specialized functions in solving specific geometric problems. Consequently, their implementation has to ensure all possible cases, entities involved and different scenarios.

The solver called **Mathematical Calculation Kernel** addresses the problem in a generic way, using for this purpose a algebraic computation and symbolic manipulation system. This solver supports a declarative description of the model, in which the model is restricted iteratively, step-by-step, and you can apply operations and declare relations.

The Mathematical Calculation Kernel has a modular architecture inspired by the Thinglab [6], so that the operations implemented in this solver describe relationships between variables. This allows the variables to be resolved in multiple directions, i.e. the mathematical problem is manipulated so that the unknown variables to be resolved from the known. These functions can be made by combining various geometric constraints.

In this approach, the description of the geometric model is transformed into a mathematical problem, which is solved by the algebraic computation and symbolic manipulation system. The presented solution also has the great advantage of supporting geometric models where the *DoF* is *Under-constrained* or *Well-Constrained*, or even if the geometric model described by the user is not fully restricted, it is possible to interrogate the properties of geometric entities if they are mathematically deductible from the current description.

3.1 Geometric Entity

This section presents some of the geometric entities and their properties that are available in the language. The classical geometric forms have one, two, and three dimensions, which the end-user can use these entities to describe the geometric model. Additionally, there are other entities that the end-user does not need to know, e.g. the form entity is composed by a set of variables and it is used to represent a geometric space that can be empty or represent the universe.

- **Entities value and var** - This entities represents a value or a mathematical expression. They have structure field `v` that is a special type, implemented by us. In Racket is a number or a string with a mathematics expression. The `var` has other fields that will be able to link various entities. The value of entity `var` may depend on other entities. The entity `var` will replace de entity `value`.

- **Entity point** - This entity represents a point in space and has the following fields `x`, `y` and `z`, they are of the type value.
- **Entity line** - This entity represents a line in space and has the following fields: `start` and `end` are of the type point; `direction` are also a point but should some vector type; `length` are of the type value; `subtype` are of the type symbol. The field `subtype` must be one of the following symbol `straight` `semistraight` `segment`
- **Entity circle** - This entity represents a circle in space, but this time only supports inscribed circle in a plane parallel to the 'xy' plane. The fields of this entity are `center` with type point and `radius` with type value.
- **Entity sphere** - This entity represents a sphere in space and have the same fields than circle.

The entities do not need be totally restricted, in other words, the entities fields can be undefined or just *Under-constrained.* Some field have default values, for example the field `subtype` have the symbol `straight` by default.

3.2 Geometric Function Library

In this section we analyze the Geometric Function Library. The library aims to implement functions that allow solving geometric constraints for common scenarios, effectively and efficiently. This library follows the logic of the other existing tools on the market, implementing specialized algorithms to solve each of the geometric constraints.

On the created language, the geometric problems are addressed in a generic way, using the algebraic computation systems and symbolic manipulation to solve equation systems that represent the constraints of the geometric model. We devoted most of the implementation effort in this general approach through the Mathematical Calculation Kernel. This library only to demonstrate the advantages and disadvantages compared to the general solution, and how both approaches can be complemented between them.

We found that the packages of repositories Racket, there are no libraries that natively implement functions that let you work with geometric constraints. A valid solution would be to integrate with native libraries from other languages, using modules that let you integrate other languages into Racket. For example, "Python for DrRacket" [14] and "P2R - Processing to Racket" [15], which allows to use respectively packages Python and Processing in Racket.

A tool that would be interesting to integrate with our Geometric Function Library would be the CGAL library. This library is a project implemented in C ++, which provides efficient and accurate geometric algorithms. The CGAL is used in various areas that need geometric calculation, such as computer graphics; computer aided design; molecular biology; among others.

However the solution followed to create Geometric Function Library was to implement was geometric algorithms, in Racket by hand.

The BFG consists of five modules, four of which are located in the folder "function-module":

- "utilities-2d-lib"
- "utilities-3d-lib"
- "intersections-2d-lib"
- "geometric-functions-lib"

These are totally independent from the rest of the system, i.e., none of the functions, using the structures implemented in other modules.

All library functions are available in the module "geometric-functions-lib", yet each function expects its particular set of parameters and have an output format. In this way, we create the macro called `mask` to facilitate the integration with the others modules and data structures. The macro `mask` allows you to set the parameters that each function accepts and how are automatically extracted from the geometric entities. Furthermore, it is also defined how the results of the functions are processed. This part of the code is implemented in the module "functions-mask" in the "core" folder whose features depend already from the rest of architecture, particularly the geometric entities.

Modules "utility-2d-lib" and "utility-3d-lib" provide useful functions for users to implement other functions, are used by the functions of the module "intersections-2d-lib".

In the module "intersections-2d-lib" there are three algorithms for intersection calculation, specialized to address the intersection between: two lines; two circumferences; a circle and a line.

- **Algorithm of intersection of two lines:** The algorithm to calculate the intersection between two straight lines is based on the web page Wolfram [16] and article [17].
- **Algorithm of intersection between circle and lines:** The algorithm to calculate the intersection between a line and a circumference is based on the web page Wolfram [18] and article [19].
- **Algorithm of intersection of two circles:** The algorithm to calculate the intersection of two circles is based on article [20].

The user who wishes to use only the **Geometric Function Library**, can simply import the module "geometric-functions-lib", it contains all library functions and depends only on the modules in the folder "function-module", being totally independent of the structures which support the geometric entities. The user can easily expand these libraries by creating new algorithms or integrate with other libraries already implemented with the help of the macro `mask`. This macro helps to encapsulate the function, providing a high level function with meta information to support the geometric entities.

3.3 Macros

This section presents the structures of geometric entities and how these are automatically created by the macros. We also analyze the code generation and the different phases of execution. The section presents the created macros and care that we had to make the language easily expandable.

The pillars of domain language are based on this features. It presented along this section its specific syntax. Macros store metadata about the geometric entities in auxiliary structures, which exist only during the expansion of the code, in order to be able to share this information between the entities, which will have a direct impact on the generated code and will allow future entities to expand correctly and separately.

All our implementation based on macros is inspired by the work of Matthew Flatt, particularly in three of his papers: [21], [22], and [23].

3.4 Creation of Geometric Entities

All entities are created through the sequence call of the macros `entity-type` and `entity-build`. The code 1 exemplifies the use of those macros, defining the point entity which represents a location in space.

Listing 1: Structure of the geometric entity 'point'.

```
(entity-type point
   ([x value]
    [y value]
    [z value 0]))
(entity-build point)
```

The macro `entity-type` generates the code for two distinct phases, *phase 0* execution-time and *phase 1* expansion-time. For the *phase 0* it creates the mutable structure that will store the instances of `point` "`[(struct point entity (x y z) #:transparent #:mutable)]`".

For the *phase 1* it adds to the auxiliary structure, named '`IDtable`', which contains information about the entity that is being defined, in order to share this information between the multiple executions. This programming logic through macros is explained in [22]. All geometric structures are substructures of the `entity` structure, which contains only a field `id` that is automatically generated and identifies every instance of any entity. The macro `entity-build` generates code only for phase 0. Is with this macro that every basic functions from the entities are defined.

3.5 Dispatcher

In this section it is analyzed how the dispatcher operation can be implemented that decides where the computation will be done. Usually, using the Geometric Function Library to compute the solution is the best option. However, if there are no available functions to compute the solution in the Geometric Function Library the Mathematical Calculation Kernel is used.

At this moment the dispatcher is only implemented for the intersection functions as prove of concept and it was used during the benchmark evaluation.

3.6 Integration

It is part of the objectives of this study to extend the expressive power of Rosetta, but as it was also important to think of a more general integration so that our solution can be used in a decoupled from a context Rosetta, developed more generic functions of integration that facilitate the integration with other systems.

Racket is a multi-paradigm programming language which supports several languages, i.e., it is easy to integrate with the code written in other programming languages and even languages with different paradigms, such as functional programming, object oriented programming, etc. Thus, the Racket allows the user to define in the first line of each module the dialect that is being used with the command "#lang". Each dialect have an associated parser (also known as *language reader*), which is what enables the support if languages with different syntaxes.

In our project, we avoid creating our own dialect, so that we can use Racket's own parser ("#lang racket"). Thus, any language that uses the Racket dialect can be directly integrated with the created language. Rosetta was written using this dialect, so the user can use our tool directly form Rosetta.

4. EVALUATION

In this section we evaluate our system, in particular we evaluate the performance of the system and the language architecture.

Our evaluation strategy is the following:

- Robustness and accuracy of the language, the solutions are acorrect and precise.
- Performance benchmark between the Mathematical Calculation Kernel and the Geometric Function Library.
- Language Extensibility.

4.1 Robustness and Accuracy of Language

The robustness and accuracy of the results depends on the solver used to solve the constraints.

The Geometric Function Library uses the capabilities of Racket to minimize the robustness and accuracy problems of the solution, since Racket divides the numbers in two categories, accurate and inexact.

- Inexact numbers are represented with floating point and exponents, which raise the accuracy problem.
- Exact numbers are divided in two categories, integers or a rational.
  - Integer numbers are represented in binary without precision loss.
  - Rational numbers are exactly the ratio of two arbitrarily integers.

The user should use the exact numbers as much as possible, however there is an additional cost in the operations to compute the exact numbers and some even compute inexact numbers, for example the `sqrt 2`.

In the Mathematical Calculation Kernel, accuracy is solved by the manipulation capabilities of the Mathematical Calculation Kernel, which automatically convert the numbers to rational ones.

4.2 Benchmark

As expected the benchmark proved the Geometric Function Library performance is three orders of magnitude faster than the Mathematical Calculation Kernel. Such difference happens because the Mathematical Calculation Kernel has to communicate with the calculation system and the computation of a equation set using analytical methods, whereas the Geometric Function Library is a procedure in Racket.

5. LANGUAGE EXPANSION

Here it is presented how easy it is to extend the language with a set of functionalities completely new, using the modular architecture that allows to combine several capabilities to describe complex constraints.

The sphere geometric entity can be implemented with the following code 2. The field "`raius`" represents the sphere radius, however the radius field is not constraint as being only a positive number because it is not yet possible to define constraints using the macro.

The structure that supports the sphere geometric entity is created using the macros "`entity-type`" and "`entity-build`", and it uses the entities "`point`" and "`value`".

Listing 2: Geometric Entity sphere Implementation.

```
 #lang racket
 (require "../entity-macro.rkt")
 (require "point.rkt")
 (require "value.rkt")

 (entity-type sphere
   ([center point (new-point 0 0)]
    [radius value 1]))

 (entity-build sphere)
```

The next step is to create the function that create mathematical equations using the sphere variables in order to restricting the location of the sphere. The code represents a possible implementation of this function.

Listing 3: Mathematical equations used to describe the sphere surface

```
(define (sphere-form-equation sphere form)
  (let ([c (sphere-center sphere)]
        [r (sphere-radius-s sphere)]
        [Fx (point-x-s form)]
        [Fy (point-y-s form)]
        [Fz (point-z-s form)])
    (make-property
      sphere
      "(~a-~a)^2+(~a-~a)^2+(~a-~a)^2 = ~a\^2"
      `((,Fx ,(point-x-s c) ,Fy ,(point-y-s c)
          ,Fz ,(point-z-s c) ,r))
      `(,Fx ,Fy ,Fz)
      `(,(~a "is(" r ">0)")))))
```

We can incorporate the community expansions to the base project by importing the geometric entities in the module entities-declaration; import the specialized functions in the "function-mask" module; import the functions that use the Mathematical Calculation Kernel in "entities-properties" module.

The initial use case was designed to only have functionality to constraint the sphere using only four surface points, we obtain instead much more than just this functionality. The main advantage of the modular architecture is the possibility to combine functionalities, the additionally functionalities obtained are the following:

- Possibility of constraint the surface points of the sphere (the opposite of the initially wanted, the inverse function);
- Possibility of constraint the surface points using a subset of previously defined surface points and other sphere properties (e.g radius);
- Possibility to constraint any geometric entity, using surface points, as long as the entity is supported by the function "`entity-form-equation`".
- Possibility to constraint the sphere using other restrictions already implemented constraints.

6. CONCLUSION

This document addresses the issue of modeling by geometric construction, through the description of relationships and constraints between geometric entities. The proposed solution for resolving the constraints on the model description, is based on the transformation of these descriptions of the geometric model in a mathematical problem. For this purpose, a specific domain language was developed, so that descriptions are defined using geometric constraints descriptions, and is designated as **ConstraintGM**.

We also analyzed a diverse set of tools and programming languages dedicated to geometric modeling, including their strengths and weaknesses were raised. These were taken into account during the development of the solution leading to the domain language developed, would support a declarative description of the geometric model, and that expansion of functionality and integration of new geometric constraints in the domain of language were available. In addition, the problem of robustness and accuracy of the results was discussed with analytical techniques and symbolic manipulation.

The solution comprises two alternative *solvers* that solve the geometric constraints differently, one that is faster solving a specific set of geometric problems and, other that is computationally heavy but is more generic and is able to solve more geometric problems.

Regarding the problem of robustness and accuracy, the results differ depending on the solver used. Using **Geometric Function Library**, the problem is mitigated by the existence of fractional numbers from Racket. Using **Mathematical Calculation Kernel**, the problem is solved with the symbolic manipulation capabilities from the algebric computation system.

There is also a set of best practices that users should follow to avoid such problems, described in Article [12]. In short it is to avoid, whenever possible, the inexact numerical representation types and operations that are by nature computationally inaccurate.

We conclude that our work has reached the main objective proposed, which consisted in providing the user with the possibility of geometric constructions modeling trough description of geometric constraints. Additionally, we also managed to achieve the secondary objective, because it is possible to use this tool with Rosetta, since our system is imported as a module from the Racket. In this regard, we consider to have developed a tool that is easy to use and understand by our end users, and at the same time, we managed to create an easy tool to extend for more advanced users.

7. REFERENCES

[1] H. G. Mairson, "Functional geometry and the traité de lutherie: Functional pearl," in *ACM SIGPLAN Notices*, vol. 48, pp. 123–132, ACM, 2013.

[2] O. Le Roux, V. Gaildrat, and R. Caube, "Constraint satisfaction techniques for the generation phase in declarative modeling," in *Geometric Modeling: Techniques, Applications, Systems and Tools* (M. Sarfraz, ed.), pp. 193–215, Springer Netherlands, 2004.

[3] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[4] H. A. B. W. F. de Regt, Rogier; van der Meiden, “A workbench for geometric constraint solving,” *Computer-Aided Design and Applications*, vol. 5, no. 1-4, pp. 471–482, 2008.

[5] M. Dohmen, “A survey of constraint satisfaction techniques for geometric modeling,” *Computers & Graphics*, vol. 19, no. 6, pp. 831–845, 1995.

[6] A. Borning, “The programming language aspects of thinglab, a constraint-oriented simulation laboratory,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 3, no. 4, pp. 353–387, 1981.

[7] C.-K. Yap, “Towards exact geometric computation,” *Computational Geometry*, vol. 7, no. 1, pp. 3–23, 1997.

[8] A. Payne and R. Issa, “The grasshopper primer,” *Zen ‘Edition. Robert McNeeI & Associates*, 2009.

[9] “Eukleides - A geometry drawing language.” From Eukleides Web Page: http://www.eukleides.org. Accessed: 2015-05-20.

[10] “GeoSolver.” From GeoSolver Web Page: http://geosolver.sourceforge.net. Accessed: 2015-05-20.

[11] T. Tantau, “Tikz & pgf: Manual for version 2.10,” 2012.

[12] M. I. Karavelas, “Exact geometric and algebraic computations in cgal,” in *Mathematical Software–ICMS 2010* (K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, eds.), vol. 6327 of *Lecture Notes in Computer Science*, pp. 96–99, Springer Berlin Heidelberg, 2010.

[13] E. Berberich and E. Berberich, “CGAL–reliable geometric computing for academia and industry,” in *Mathematical Software–ICMS 2014* (H. Hong and C. Yap, eds.), vol. 8592 of *Lecture Notes in Computer Science*, pp. 191–197, Springer Berlin Heidelberg, 2014.

[14] P. P. Ramos and A. M. Leitão, “Implementing Python for DrRacket,” in *3rd Symposium on Languages, Applications and Technologies* (M. J. V. Pereira, J. P. Leal, and A. Simões, eds.), vol. 38 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 127–141, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[15] H. Correia and A. M. Leitão, “Combining processing with racket,” in *Languages, Applications and Technologies*, pp. 101–112, Springer, 2015.

[16] “Line-line intersection..” From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/Line-LineIntersection.html. Accessed: 2016-04-27.

[17] M. Berg, M. Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, ch. Computational Geometry, pp. 1–17. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[18] “Circle-line intersection..” From MathWorld–A Wolfram Web Resource: http://mathworld.wolfram.com/Circle-LineIntersection.html. Accessed: 2016-04-27.

[19] A. Johnson, R. Rhoad, G. Milauskas, and R. Whipple, “Geometry for enjoyment and challenge, rev. ed.(s),” 1991.

[20] E. Hartmann, “Geometry and algorithms for computer aided design,” *Darmstadt University of Technology*, 2003.

[21] M. Flatt, “Submodules in racket: you want it when, again?,” in *ACM SIGPLAN Notices*, vol. 49, pp. 13–22, ACM, 2013.

[22] M. Flatt, “Composable and compilable macros:: you want it when?,” in *ACM SIGPLAN Notices*, vol. 37, pp. 72–83, ACM, 2002.

[23] M. Flatt, “Binding as sets of scopes,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 705–717, ACM, 2016.