A Research Proposal on Mathematically Rigorous and Computationally Efficient Representations of Geometry

Steve Kelly

October 19, 2015

Abstract

Geometry is fundamental to the development of natural problem statements. Opportunities exist in computational geometry to add rigourous type descriptions and algorithmic relations that do not sacrifice performance. This proposal will develop a foundational context for such a research project.

1 Introduction

Geometry is one of the earliest academic studies in mathematics. Research is constantly leading to new patterns and constructions. Many of these find highly practical applications in engineering. In this paper we will seek to study geometry by developing patterns that are sensible to computers and people. To obtain sensible patterns we address two different audiences. One is the computer which requires execution efficiency. The other is people, who require lucid representation of code. Such challenges are well known in the Human-Computer Interaction (HCI) field. [1]

This proposal will try to refine existing implementations of geometry for computers and explore new procedures. As such, we will review the existing systems, their applications, and areas for improvement. Later we will discuss some of the areas for improvement in more detail, and establish a proposed scope for the project going forward.

2 The Julia Programming Language

Programming languages are the grammar and syntax a computer presents to a user. This project is fundamentally exploratory in nature and seeks to generate understanding of geometric relationships. I will use Julia as a programming language for exploration. In the following sections I hope to develop some rationale for this choice, and give a brief introduction to the language concepts that will help make computationally effective geometry representations possible.

In addition we will use it to illustrate some concepts and relations. The REPL (Read-Eval-Print-Loop) allows interactive evaluation of Julia code. It is highly useful for exploration and testing of ideas in the language. Blocks starting with "`julia>`" represent input and the preceding line represents output of the evaluated line.

2.1 History

Julia is a programming language first released in early 2012 by a group of developers from MIT. The language targets technical computing by providing a dynamic type system with near-native code performance. This is accomplished by using three concepts: a Just-In-Time (JIT) compiler to target the LLVM framework, a multiple dispatch system, and code specialization.[2] [3] The syntactical style is similar to MATLAB and Python. The language implementation and many libraries are available under the permissive MIT license.[1]

Benchmarks have shown the language can consistently perform within a factor of two of native C and FORTRAN code.[2] This is enticing for a solid modeling application and for numerical analysis, as the code abstraction can grow organically without performance penalty. In fact, the authors of Julia call this balance a solution to the "two language problem". The problem is encountered when abstraction in a high-level language will disproportionately affect performance unless implemented in a low-level language. In the next sections we will compare the expressibility and performance to other languages.

2.2 Comparisons

Many languages are as fast as Julia but sacrifice expressibility. In Figure 1 we can see some comparisons to other programming languages. This was developed by the Julia core team, and illustrates that Julia is highly competitive in performance. Again, these results stem from the compiler and language design. In Figure 2 we can see these results normalized against code length. The Julia code is quite short, yet consistently achieves good performance. Much of this comes down to the innovated type and function system. We will discuss these more in depth later.

In 1972 Alan Kay introduced the terms "class" and "object", to describe a coupling of data and functionality.[3] An object is simply and implementation of a class. Computer Scientists call this "Object Oriented Programming" (OOP). Languages such as C++, Java, and Python all subscribe to this paradigm. In Python this looks like the following:

```
class Foo:
    foo1
    foo2
    def add_to_foo1(self, x):
        self.foo1 += x
```

This system positively enables specialization of functionality, but due to the coupling of data with functions it becomes a challenge to extend functionality. Languages for scientific computing generally avoid the "traditional" notions of OOP. In Table 3 we can see a comparison of type systems used in scientific computing languages. In the next few sections the implications of multiple dispatch and the relation to OOP will be developed further.

[1] `http://opensource.org/licenses/MIT`
[2] `http://julialang.org/benchmarks`
[3] `http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html`

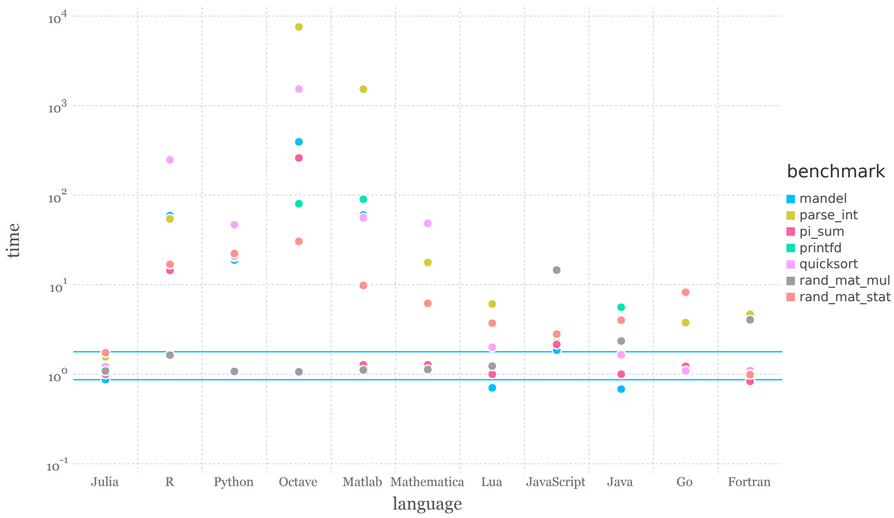

Figure 1: A comparison of programming languages and performance.

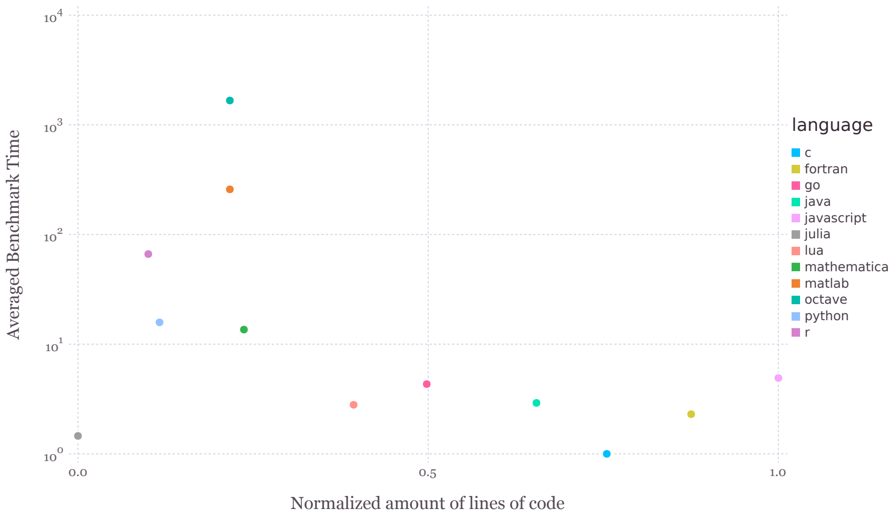

Figure 2: The results in Figure 1 normalized for code length. (Courtesy of Simon Danish)

2.3 Functions

Julia is an experiment in language design. Much of the advancement revolves around the representation of data and the execution of functions. The language is optionally typed, which means function specialization on types is inferred. We will use it See below:

```
julia> increment(x) = x + 1
```

Figure 3: A comparison of functions, typing, and dispatch.

Language	Type system	Generic functions	Parametric types
Julia	dynamic	default	yes
Common Lisp	dynamic	opt-in	yes (but no dispatch)
Dylan	dynamic	default	partial (no dispatch)
Fortress	static	default	yes

```
increment (generic function with 1 method)

julia> increment(1)
2

julia> increment(1.0)
2.0
```

the `increment` function was defined for any `x` value. When the `1`, an integer type was passed as an argument, an integer was returned. Likewise when a floating point, `1.0` was passed, the floating point `2.0` was returned.

Let's see what happens when we try a string:

```
julia> increment("a")
ERROR: MethodError: `+` has no method matching +(::ASCIIString, ::Int64)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
  +(::Int64, ::Int64)
  +(::Complex{Bool}, ::Real)
  ...
 in increment at none:1
```

The problem is that the `+` function is not implemented between the `ASCIIString` and `Int64` types. We need to either implement a `+` function which might be ambiguous, or specialize the function for `ASCIIString`. A specific implementation is preferrable in this case:

```
julia> function increment(x::ASCIIString)
           ASCIIString([increment(c) for c in x])
       end
increment (generic function with 2 methods)
```

The line `x::ASCIIString` is called a "type annotation" and states that `x` must be a subtype of `ASCIIString`. This allows one to control dispatch of functions, since Julia will default to the *most specific implementation*. Since`ASCIIString` is a series of 8 bit characters, we can iterate over the string and increment each character individually. The `[]` indicates we are constructing an array of characters to pass to be passed to the `ASCIIString` type constructor. Now we see our example works:

```
julia> increment("abc")
"bcd"
```

What was demonstrated here is the concepts of specialization and multiple dispatch, both are highly coupled topics. Each function call in Julia is specialized for types if possible. This means the author only has to write a few sufficently abstract implementations of functions. If special cases occur multiple functions with different arity or type signatures can be implmented. Explicitly this is called multiple dispatch. In practice by the user this looks like abstracted or generic code. To the computer, this means choosing the most specific, and thus performant method. Let's go back to the integer and floating point example. Below is the LLVM assembly generated for each method:

```
julia> @code_llvm increment(1)

define i64 @julia_increment_21458(i64) { // <return type> <function name>(<arg type>)
top:
  %1 = add i64 %0, 1
  ret i64 %1 // return <return type> <return id>
}

julia> @code_llvm increment(1.0)

define double @julia_increment_21466(double) {
top:
  %1 = fadd double %0, 1.000000e+00
  ret double %1
}
```

Note I have annotated the LLVM code so this is understandable. The only real similarity is the line count. Each one of these functions are generated by the Julia compiler at run time.

Many of the concepts used for performance also serve as methods for expressability. In this case, multiple dispatch used by the compiler for specialization of functions reveals it self as a way for the user to specialize over many types. Revealing the role in which this paradigm allows Julia to achieve high performance is a matter to be developed in further sections.

2.4 Types

2.4.1 Mutability and data packing

Types and immutables are containers of data. The primary difference between the two is the notion of "mutability". Types are mutabile, immutables are immutable. What does this mean? Let's break something first:

```
julia> type FooIsMutable
           a
       end

julia> f = FooIsMutable(1)
FooIsMutable(1)

julia> f.a
1

julia> f.a = 2
2

julia> f.a
2

julia> immutable FooIsImmutable
           a
       end

julia> f = FooIsImmutable(1)
FooIsImmutable(1)

julia> f.a
1

julia> f.a = 2
ERROR: type FooIsImmutable is immutable
```

What just happened demonstrates the contract defined by mutability. Mutable objects, which is an instance of a type (i.e. `f`), can have their fields (i.e. `a`) changed. Immutables cannot. The immutable contract helps develop a notion of functional purity. To the user this means immutables are defined by their values. Practically this can be of great benefit to the compiler. For example:

```
julia> a = (1,2,3)
(1,2,3)

julia> b = typeof(a)
Tuple{Int64,Int64,Int64}

julia> isbits(b)
true

julia> a = ([1],[2],[3])
([1],[2],[3])

julia> b = typeof(a)
Tuple{Array{Int64,1},Array{Int64,1},Array{Int64,1}}

julia> isbits(b)
false
```

`isbits` ask the question "will this type be tightly packed in memory"? A `Tuple` is a fixed-length set of linear, ordered, data. It has syntax for construction with `()`. In computations we want our data be close together for fast access. In modern times we call such data "cache friendly", or "cache localized". Immutability helps us achieve this. Let's look that the types inside the 3-tuples and see their `isbits` status:

```
julia> isbits(Array{Int64,1})
false

julia> isbits(Int64)
true
```

Why is this the case? We see that `Int64` is bits, because it is literally 64 bits. In Julia a `bitstype` behaves similar to an immutable, and is identified by value. `Array{Int,64}` is a mutable data type that can vary in size. This means the `Tuple` needs to store the arrays as references, in this case a pointer. When iterating over a data set, such a "pointer dereferences" (this is jargon for accessing the data in memory pointed to by a pointer), can be costly. Modern CPUs accell when data is linearly packed and pointer-free. The data can be brought into the CPU's memory cache once and computed without shuffling between cache and RAM.

2.4.2 Parameters

Type parameters are w

2.4.3 Macros and Generated Functions

Julia is a descendant of the Lisp family of programming languages. Lisp is a portmanteau for "List Processing". The language was designed to address the new notion of "types", specifically in application to Artificial Intelligence (AI) problems.[4] The notion of an "S-Expression" was introduced in McCarthy's seminal work. These statements use parenthesis to denote functions and arguments. Below is an an example of S-Expressions for addition and multiplication.

```
> (+ 1 1)
2

> (* 3 4)
12
```

This syntax is noted for it's mathematical purity. However it can be a syntactic difficulty for many. Most of the current popular programming languages use variants of ALGOL syntax, which is noted for being more readable. [5] Julia also uses ALGOL syntax, but is lowered to S-Expressions. This enables many of the mathematically pure relations we seek to achieve. In addition S-Expressions are highly conducive to source transforms. This develops a notion of "Homoiconicity", where the representation of program structure is similar to the syntax. In Julia we use this property to make "macros" which enable source code to be transformed based on structure or type information before compilation.

More on generated functions

Become more articulate like Graydon Hoare: `http://graydon2.dreamwidth.org/189377.html`

2.5 Example

2.6 On the changing state of Computation

[6]

3 Solid Modeling Paradigms

The expression of solid bodies is fundamental in the development of any natural problem statement. For example, in diffusion we model the transfer of energy throughout a domain. An engineer might define such a domain with a model, say of an injection molding nozzle. Such a domain is difficult to describe in terms of a functional boundary, so the engineer might prefer a boundary representation.

In addition to being fundamental to natural studies, solid modeling is growing in popularity due to low-cost digital manufacturing tools reaching the market. There have been 3D printers popping up in nearly every educational institutions over the past 3 years. In addition CNC routing, and laser cutting enable people to go quickly from design to fabrication.

The development of modern computational tools for solid modeling have vastly different paradigms. In the next few sections we will layout the mathematical and computational principles of these paradigms.

3.1 Implicit Functional Representation

Implicit functional representation (FRep) in computation centers around a signed, real-value function where the boundary is defined as $f(...) = 0$. In $\mathbb{R}3$ this looks like $f(x, y, z) = 0$. For modeling purposes we must add the additional constraint that the function evaluates to a negative inside the boundary. Further more the magnitude of the return value must correspond to the minimum distance between the point and the boundary.[7] A sketch of this behavior in one dimension can be seen in Figure 4.

Researchers at MIT have taken these principles to constrain geometric features to ensure a part can be manufactured.[8] They call their system "Fab-Forms" and shows how functional representations can easily accept constraints.

More importantly, functional representations can naturally deal with affine transforms. [9] Given some transform associated with a FRep, one simply applies the inverse transform to check membership.

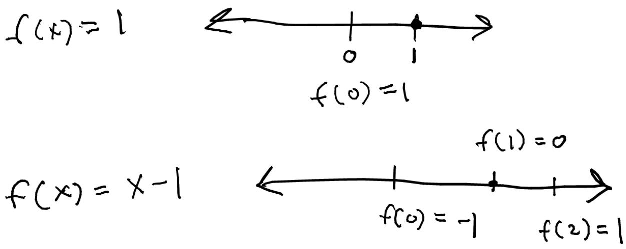

Figure 4: Number line illustrating the construction of an implicit signed function

One can also compose functional representations with set operations. The union of two functions, f_1 and f_2 can be expressed by $min(f_1, f_2)$. An intersection is expressed as $max(f_1, f_2)$, and it follows that the difference of f_1 and f_2 is the intersection of the negation of f_2, $max(f_1, -f_2)$. The mathematical analyst might have trouble with this because such operations create discontinuities. This is one area of exploration that will be discussed later.

3.2 Signed Distance Fields

A signed distance field (SDF) is a uniform sampling of an implicit function, or any oriented geometry. Below we can see this in action over the definition of a circle.

didn't introduce orientation, winding order, etc...

```
julia> f(x,y) = sqrt(x^2+y^2) - 1
f (generic function with 1 method)

julia> v = Array{Float64,2}(5,5) # construct a 2D 5x5 array of Float64

julia> for x = 0:4, y = 0:4
           v[x+1,y+1] = f(x,y)
       end

julia> v
5x5 Array{Float64,2}:
 -1.0  0.0       1.0      2.0      3.0
  0.0  0.414214  1.23607  2.16228  3.12311
  1.0  1.23607   1.82843  2.60555  3.47214
  2.0  2.16228   2.60555  3.24264  4.0
  3.0  3.12311   3.47214  4.0      4.65685
```

The results of `v` might be confusing since the matrix is oriented with the origin in the top left corner. At coordinate $(0, 0)$, or entry `v[1,1]`, we see that `f` is equal to `-1`. Likewise we can see $(0, 1)$ and $(1, 0)$ are points on the boundary since the value is `0` and everywhere else is positive.

Distance fields are interesting since they provide an intermediate representation between functional space and discrete-geometric space. However they are a very memory hungry data structure. Pixar has published OpenVDB which helps work around these concerns, but such compression can be lossy.[10] With the advent of shader pipelines for GPUs, distance fields have become more popular. Valve has used SDFs with great success for generating smooth text.text renders. [11] Many algorithms for generating polyhedra from an SDF exist. The most common are Marching Tetrahedra, Marching Cubes, and Dual Contours.[12][13][14]

Talk about vert and frag shaders.

Andreas Bærentzen and Henrik Aanæs published methods on the inverse problem of converting a mesh to a signed distance fields.[15] DiFi was introduced in 2004, which demonstrates an algorithm for creating SDFs on multiple types of geometry [16].

Need to re-read this paper

Many necessary algorithms in path planning for digital manufacturing tools fall out of distance fields. For example, offsetting simply becomes an addition or subtraction over the SDF. Computing the medial axis becomes a scan for inflection points. Many path planners need to simplify polygon representations as to not generate move less than the resolution of the machine. Assuming the machine uses a Cartesian system, a SDF can correspond perfectly to the lowest available resolution of the machine. Likewise as Stereolithographic 3D printers begin to use digital mirror devices (commonly known as DLP or DMD) , discrete representations of geometry will become more important in digital manufacturing.

3.3 Mesh

[17]

[18]

3.4 Boundary Rep

Boundary Representation (B-Rep) has been the dominant modeling paradigm for engineering since the 1970's. It relies primarily on the manipulation and representation of edges, vertices, and faces to build a model. The primary mechanism for the representation is a "feature tree". While B-Rep is intuitive for users of a graphical environment, it is unwieldy as a textual and functional representation. This methods is natural for engineers and designers, but sacrifices parametric design. In addition, B-Rep requires the use of a geometry kernel to handle the interpretation of constraints and geometric construction. [19]

Geometry kernels often decouple functional representations from a user's design hierarchy which complicates numerical analysis.[20] This middle step of Computer Aided Engineering (CAE) is known as pre-processing. For example in the Finite Element Analysis (FEA) process the requires establishing proper aspect ratio, area, and connectivity of nodes.

need more substance on how this is different from Mesh and the others

3.5 Constructive Solid Geometry

Constructive Solid Geometry (CSG), works using manipulation of geometric primitives (half-spaces) as a level of abstraction in design. CSG has been growing in popularity due to programs such as OpenSCAD[4], CoffeeSCAD[5], POVRay[6] and Thingiverse Customizer[7]. Each exclusively uses text representations to describe geometry and all are open source. These programs are par-

[4] `http://www.openscad.org`
[5] `http://coffeescad.net/`
[6] `http://www.povray.org/`
[7] `http://www.thingiverse.com`

ticularly popular for collaboration in conjunction with version control systems such as Git. Figure 5 illustrates the process of describing complex geometries with CSG.

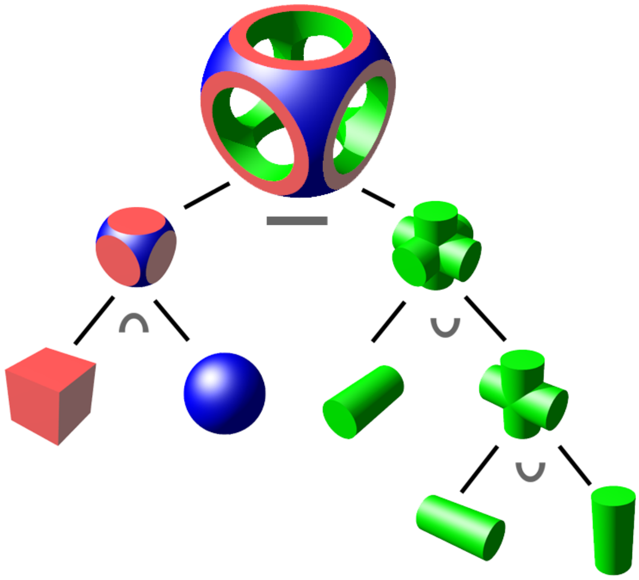

Figure 5: An illustration of a CSG tree.

Most of these systems use mesh or functional representations as the underlying method of operation.

3.6 Graphs

Geometry as a graph representation is probably the most exciting and underdeveloped field of study in this computational geometry.

Laplacian Contractions!!! [21]

3.7 Linear Algebraic Representation

[22]

3.8 Visualization

[23] [24]

4 Exploration

4.1 Rigorous Definitions of Geometry

GeometryTypes.jl is a package for Julia that provides geometric structures and relations. It was started early 2015 as the integration of Meshes.jl, ImmutableArrays.jl, HyperRectangles.jl, and FixedSizeArrays.jl. This package is able to resolve the relations between geometric structures for computational algorithms and fast visualization on the GPU. With the release of Julia version 0.4 is became possible to build the appropriate abstractions. For example ImmutableArrays represented a 3 dimensional vector with the concrete type `Vector3{Int64}`. FixedSizeArrays introduced the dimensionality as a parameter as `Vector{3,Int64}`. This means the notion of a fixed length vector can be abstracted over arbitrary dimensionality.

[25]

4.1.1 Numerical Robustness

Numerical robustness is a perennial problem in computational geometry. Multiple approaches exists for various numeric types. Floating points are by far the most difficult to deal with. Tools such as Gappa have been developed so algorithm writers can check their invariants when using floating points. [26]

One of the most common problems formulated is determining whether or not a point is colinear with a line segment. Shewchuk has one of the most pragmatic and robust treatments on this topic.[27] Kettner, et. al. have also developed more examples where numerical robustness is critical. [28]

Julia's GeometricalPredicates package [8] uses the approach outlined by Volker Springel, which requires all floating point numbers to be scales between 1 and 2.[29] This has the downside of significantly reducing the available resolution. why

My preferred approach is to avoid the difficulties of floating point by working within integer space. Developing a system around this is of interest. For example, it should be possible to specify a minimum unit (e.g. microns) and perform all computations in integer space assuming this does not exceed the needed resolution. More importantly, modern CPUs have integrated 128 bit Integer support. 170141183460469231731687303715884105727 is a lot of microns.

4.1.2 Simplices

Recently a Simplex type was added to GeometryTypes. A Simplex is defined as the minimum convex set containing the specified points. The initial prototype is very simple yet works well. Below we can see a 0th and 1st order simplex constructed in $\mathbb{R}2$ and $\mathbb{R}3$

```
julia> using GeometryTypes

julia> Simplex(Point(1,2))
GeometryTypes.Simplex{1,FixedSizeArrays.Point{2,Int64}}((FixedSizeArrays.Point{2,Int64}((1,2)),))

julia> Simplex(Point(1,2,3), Point(4,5,6))
GeometryTypes.Simplex{2,FixedSizeArrays.Point{3,Int64}}((FixedSizeArrays.Point{3,Int64}((1,2,3)),FixedSize
```

This representation makes it possible to write code based on the order and dimensionality of a simplex. Few algorithms have been developed around the new Simplex type, and unfortunately it is not integrated as a lower-level construct for the other types yet.

In addition I would like to add a Simplical Complex type.[9]

[8] https://github.com/JuliaGeometry/GeometricalPredicates.jl
[9] https://en.wikipedia.org/wiki/Simplicial_complex

4.1.3 Distance Fields

Lagrangian and Eulerian specification of the flow field

4.2 Automatic Differentiation

4.2.1 Dual Numbers

```
julia> using DualNumbers

julia> f(x) = 2x+1
f (generic function with 1 method)

julia> f(Dual(1,1))
3 + 2du
```

4.2.2 Rvachev Functions

In the 1960's Vladimir Rvachev produced a method for handling the "inverse problem of analytic geometry". His theory consists of functions which provide a link between logical and set operations in geometric modeling and analytic geometry.[30] I believe the following anecdote helps elucidate the theory. While attempting to solve boundary value problems, Rvachev formulated an equation of a square as

$$a^2 + b^2 - x^2 - y^2 + \sqrt{(a^2 - x^2)^2 + (b^2 - y^2)^2} = 0$$

Implicitly, the sides of a square can be defined as $x = +/-a$ and $y = +/-b$. The union of these two is a square. By reducing the formulation of the square we can generalize an expression for the union between two functions.

$$\cup : f_1 + f_2 + \sqrt{f_1^2 + f_2^2} = 0$$

Likewise we can see that intersections and negations can be formed for logical completion.

$$\cap : f_1 + f_2 - \sqrt{f_1^2 + f_2^2} = 0$$
$$\neg : -f_1$$

These formulations can be modified for C^m continuity for any m. [31] In addition Pasko, et. al. have shown that Rvachev functions can serve to replace a geometry kernel by creating logical predicates. [32] Their research also establishes the grounds for user interfaces and environment description. For this work a practical implementation will most likely leverage their insights. Rvachev and Shapiro have also shown that using the POLE-PLAST and SAGE systems a user can generate complex semi-analytic geometry as well.[33]

show this construction, it isn't obvious

While a functional representation for geometry is mathematically enticing on its own, the power it gives for numerical analysis might be its greatest virtue. Numerical analysis justified the initial investigation by Rvachev early on. A boundary value problem on a R-Function-predicate domain allows for analysis without construction of a discrete mesh.[33]

One of the most general expositions in the English language of R-Functions applied to BVPs is Vadim Shapiro's "Semi-Analytic Geometry with R-Functions". [31] Unfortunately, no monographs about R-Functions exist in the English literature. Most literature is in Russian, however many articles presenting applied problems using the R-Function Method. [34]

Such a system for analytic geometry can be developed further.

4.3 Geometry in Path Planning

4.3.1 Mesh Slicing

5 Conclusion

References

[1] B. Tognazzini, *Principles, techniques, and ethics of stage magic and their application to human interface design*, p. 355–362. ACM, 1993.

[2] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computin," *arXiv preprint arXiv:1411.1607*, 2014.

[4] J. McCarthy, "4e. recursive functions of symbolic expressions and their computation by machine, part i," *Programming systems and languages*, p. 455, 1966.

[5] C. Hoare, "Hints on programming language design."

[6] M. I. Shamos, *The Early Years of Computational Geometry—a Personal*, vol. 223, p. 313. American Mathematical Soc., 1999.

[7] C. Olah, "Manipulation of implicit functions (with an eye on cad) — christopher olah's blog." `https://christopherolah.wordpress.com/2011/11/06/manipulation-of-implicit-functions-with-an-eye-on-cad/`, Nov 2011.

[8] M. Shugrina, A. Shamir, and W. Matusik, "Fab forms: customizable objects for fabrication with validity and geometry caching," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, p. 100, 2015.

[9] P. Henderson, "Functional geometry," *Higher-order and symbolic computation*, vol. 15, no. 4, p. 349–365, 2002.

[10] `http://www.openvdb.org/`.

[11] C. Green, *Improved alpha-tested magnification for vector textures and special effects*, p. 9–18. ACM, 2007.

[12] H. Müller and M. Wehle, *Visualization of Implicit Surfaces Using Adaptive Tetrahedrizations*, vol. 0, p. 243. IEEE Computer Society, 1997.

[13] T. S. Newman and H. Yi, "A survey of the marching cubes algorithm," *Computers & Graphics*, vol. 30, p. 854–879, Oct 2006.

[14] *SIGGRAPH 95 conference proceedings: August 6 - 11, 1995, [Los Angeles, California]*. Computer graphics Annual conference series, ACM, 1995.

[15] A. Baerentzen and H. Aanaes, "Generating signed distance fields from triangle meshes."

[16] A. Sud, M. A. Otaduy, and D. Manocha, *DiFi: Fast 3D distance field computation using graphics hardware*, vol. 23, p. 557–566. Wiley Online Library, 2004.

[17] P. S. Heckbert and M. Garland, *Survey of polygonal surface simplification algorithms*. 1997.

[18] B. S. Bischoff, M. Botsch, S. Steinberg, S. Bischoff, L. Kobbelt, and R. Aachen, *OpenMesh–a generic and efficient polygon mesh data structure*. 2002.

[19] I. Stroud, *Boundary representation modelling techniques*. Springer, 2006.

[20] S. H. Lee, "A cad–cae integration approach using feature-based multi-resolution and multi-abstraction modelling techniques," *Computer-Aided Design*, vol. 37, no. 9, pp. 941–955, 2005.

[21] J. Cao, A. Tagliasacchi, M. Olson, H. Zhang, and Z. Su, *Point cloud skeletons via laplacian based contraction*, p. 187–197. IEEE, 2010.

[22] A. DiCarlo, A. Paoluzzi, and V. Shapiro, "Linear algebraic representation for topological structures," *Computer-Aided Design*, vol. 46, p. 269–274, Jan 2014.

[23] J. Tupper, *Reliable two-dimensional graphing methods for mathematical formulae with two free variables*, p. 77–86. ACM, 2001.

[24] Y. Hijazi, A. Knoll, M. Schott, A. Kensler, C. Hansen, and H. Hagen, "Csg operations of arbitrary primitives with interval arithmetic and real-time ray tracing," *Utah: SCI Institute, University of Utah*, 2008.

[25] A. Pasko, V. Adzhiev, and P. Comninos, *Heterogeneous objects modelling and applications collection of papers on foundations and practice*. Springer, 2008.

[26] `http://gappa.gforge.inria.fr/`.

[27] J. Shewchuk, "Fast robust predicates for computational geometry."

[28] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap, "Classroom examples of robustness problems in geometric computations," *Computational Geometry*, vol. 40, p. 61–78, May 2008.

[29] V. Springel, "E pur si muove: Galiliean-invariant cosmological hydrodynamical simulations on a moving mesh," *Monthly Notices of the Royal Astronomical Society*, vol. 401, p. 791–851, Jan 2010. arXiv: 0901.4107.

[30] V. Shapiro, "Theory of r-functions and applications: A primer," tech. rep., Cornell University, 1991.

[31] V. Shapiro, "Semi-analytic geometry with r-functions," *ACTA numerica*, vol. 16, no. 1, pp. 239–303, 2007.

[32] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko, "Function representation in geometric modeling: concepts, implementation and applications," *The Visual Computer*, vol. 11, no. 8, pp. 429–446, 1995.

[33] V. L. Rvachev, T. I. Sheiko, V. Shapiro, and I. Tsukanov, "On completeness of rfm solution structures," *Computational Mechanics*, vol. 25, no. 2-3, pp. 305–317, 2000.

[34] M. Voronyanskaya, K. Maksimenko-Sheiko, and T. Sheiko, "Mathematical modeling of heat conduction processes for structural elements of nuclear power plants by the method of r-functions," *Journal of Mathematical Sciences*, vol. 170, no. 6, pp. 776–793, 2010.