Computational Methods for Parametrization of Polytopes

Steve Kelly

April 9, 2016

Contents

Abstract

The combinatorial and geometric realization of polytopes are outlined in mathematical and computational terminology. With these two representations in hand, various parametric forms may be constructed using vertex locations, edge angles, and symbolic values. We have implemented software which represents polytopes in a way useful for combinatorial inspection and solid modelling using the Julia programming language. These packages have been published to GitHub and are accessible to mathematical researchers around the world through the Julia package manager.

expand

Chapter 1

Introduction

Our objective is to develop a computational environment for the exploration of parametric polytopes. A computational environment is one in which we can apply rigourous defintional constraints on symbolic constructions, and likewise manipulate them to reveal properties that may be of interest. A parametric polytope is the union of two concepts. The first is the idea of parameters, which are our unknown constraints in a system. The second is a polytope, which is a some what geometric construction. We will build an intuition of a polytope in this section, and formally define it in the next chapter. Parameters will be applied in the computation realm and are expanded in Section 3.

1.1 A Geometric Intuition of a Polytope

A polytope is a geometrically realizable graph composed of linearly connected vertices.[1] A "graph" is meant in the combinatorial sense of a structure composed of vertices and edges that show connectivity. Thus a geometric realization of a graph with linearly connected vertices is highly analagous to how we would traditionally draw a graph. Figure 1.1 shows a graph in with vertices labeled with numbers and edges between them.

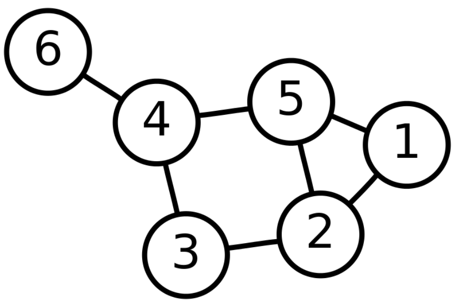

Figure 1.1: An example of a graph.

The informal term "flat" implies the path between two vertices on our polytope are connected by line segments. Thus for something to be considered geometrically realizable, we must be able to assign tuple values to the vertices.

An N-polytope then exists in the corresponding euclidean space dimension. For our purposes we will primarily use the real numbers, $\mathbb{R}^N$, for an N-polytope.

Figure 1.2 shows a two dimensional polytope, more commonly known as a polygon. The polgons we will primarily consider are 1-cycle graphs. 1-cycle implies that given any point there exist one path that exits then returns to the starting point. In three dimensions, polytopes are commonly known as a polyhedra. Figure 1.3 shows a polyhedral representation of a dolphin. Using this picture as reference, we see that given a vertex on a polyhedra there may be multiple cycles, or paths that exit then return to the vertex. Thus many assumptions about the properties of polytopes are contigent upon their dimensionality. Polytopes may have properties of convexity, connectedness, and closure associated with them. In order to study these properties we will illucidate a combinatoric and geometric representation in the coming sections. More importantly we will do so rigourously! These representations are somewhat distinct and we will see the implications as we later develop a computational type framework.

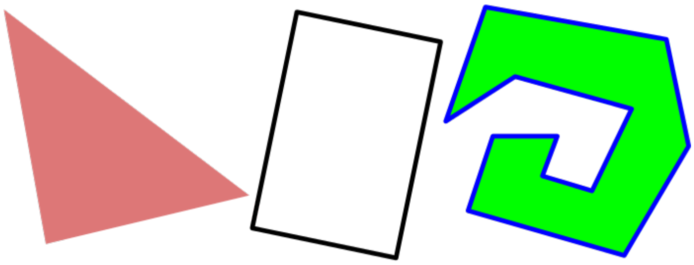

Figure 1.2: Polygons, or two dimensional polytopes

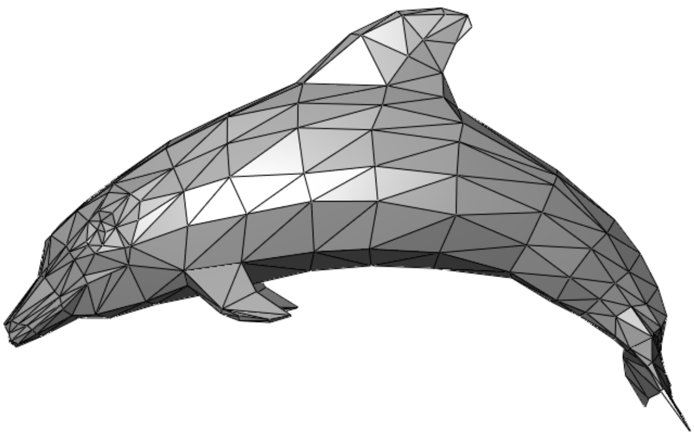

Figure 1.3: A polyhedral representation of a dolphin, or a three dimensional polytope

1.2 Flexibility of Polyhedra

Our initial problem comes via the flexibility of polyhedra. In order to understand flexibility of polyhedra it may be easier first to introduce the concept of rigidity. In this case we will be concerned with 3-dimensional polytopes, polyhedra, and observing the shape of the faces. If we give each edge (connection between vertices) a fixed length and the freedom to pivot around the vertex, a rigid polyhedra will not be able to deform. Cauchy's Ridgidity theorem proves this for any convex polyhedra. [2] If we allow the same degrees of freedom on the edges and the shape of the faces does not change it is called *flexible*. [3]

In 2015 Maria Hempel presented an analysis of this problem using a representation of a polyhedra with edges and faces specified via angles and lengths. [4] Such representations may be useful in a variety of dicisplines, but our focus will be strictly structural and for enhancing the foundations for discovering flexible polyhedra. We will expand on the significance of this representation in later sections.

Chapter 2

Mathematical Definitions

We are interested in constructing parameteric polytopes for mathematical exploration. An informal geometric picture of a polytope has already been developed. In this chapter we will develop a mathematical perspective of our problem such that we can focus clearly on the computational aspects in the following sections.

2.1 Spacial Terminology

2.1.1 Vector Spaces

Given a field (in the algebraic sense) of numbers, a vector of dimensionality N is formed by an N-tuple of numbers in the field. A vector space must be closed under element-wise addition with another vector and element-wise multiplication by a scalar value. Symbolically, given $s \in F$ where F is a field, and $X, Y \in V$ where V is the vector space over F then $X + Y \in V$ and $s * x \in V$ implies our closure property. In this paper we will primarily use the real numbers in euclidean N dimensional space, denoted as $\mathbb{R}^N$. The term "point" will generally be used to describe a vector that may described using numerical values.

2.1.1.1 Closed Vector Spaces

A "closed" space means a subset of something that contains all points in it's boundary, including the boundary. An "open" set conversely will not include the boundary, but everything inside. We sometimes use brackets to aid the representation of sets on a number line for example: (1,2) for an open set and [1,2] for a closed set. The quantity of the objects contained in our sets will vary depending on the numeric domain we choose. For example if we are using integers we have no elements in the open set and 2 in the closed set. If it is in rationals we have 1/2 included, and in the reals we have infinite points! What we have constructed on the number line can also be thought of as open and closed intervals.

start with proper definition

boundary is ambiguous

2.1.2 Solids

Before we define a polyhedra, we must introduce a few notions. These are solids and orientation. Solids have been studied since antiquity and for our purposes we will define them as constructions in three dimensional space with finite volume. For example, a plane which partitions space is not a solid since either partition is unbounded, however the intersection of planes could form a solid.

Orientation is a parallel concept which allows us to specify how geometric objects contain space. As an example, let us go back to our partition of space with a plane. If we are on our way to construct a solid, it is neccessary to choose the one part to keep and the other to discard. This is the purpose of orientation. In the case of the plane, this follows from the definition, $ax + by + cz + d = 0$. More lucidly, lets look at the signed distance of a point from the plane, computed as:

$$D = \frac{ax + by + cz + d}{\sqrt{a^2 + b^2 + c^2}} \tag{2.1}$$

For simplicity, let's look at a plane parallel to the X and Y axes passing through z = 1. Thus a and b are set to 0, c set to 1, and $d = -1$ Simplifying our formulation we have:

$$D = 0x + 0y + z - 1 = z - 1 \tag{2.2}$$

At $z = 1$ we see we are on the plane, however at $z = 0$ and $z = 2$ we get -1 and 1 respectively. The sign of the distance is our indicator of orientation. We can choose an arbitrary convention as to which partition we will count, but akin to the "right hand rule" in physics, the normals of the partition must point outwards. In the case of the plane the normal is the vector (a, b, c), which in our realization is the upwards vector $(0, 0, 1)$. Since the convention is such that the normals point outward, the partition we would consider in a solid is all points *in the opposite direction of the normal*. Also to note is the importance of sign in our distance function. We can exploit this behavior to indicate containment when performing set operations on spatial partitions. Negative values thus indicate a point inside the partition of interest. We have chosen this convention for this paper due to it's ubiquity in computation frameworks. In the field of computer graphics using descrete polytopes, this is often referred to as "winding order".

2.2 Combinatorial Representation of a Polytope

2.2.1 Simplices

A simplex (plural simplices) in N dimensional space is the minimal set of descrete points whose convex hull form a closed subset of dimensionality N. It is often thought of as the generalization of a tetrahedra into N dimensions. In one dimensional space, this is a closed interval or line segment. In two dimensions it is the triangle, and in three it is the tetrahedra. These are called a 1-simplex, 2-simplex, and 3-simplex respectively. We will represent them as

set definition

We notice that the 1-simplex is formed by two 0-simplices, a 2-simplex is formed by three 1-simplices, and the 3-simplex by four 2-simplices. The components of these compositions are called "faces". The 0-face is often called a vertex and the 1-face an edge. If we tabled the quantities of each M-face in an N-simplex out, they form Pascal's triangle and thus follow the binomial coefficient.

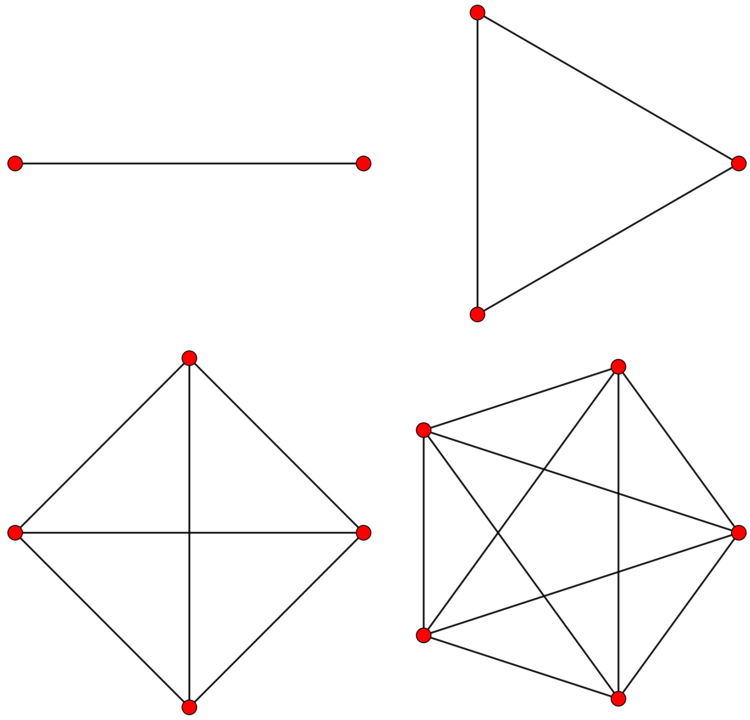

Figure 2.1: Examples of Simplices

$$\binom{N+1}{M+1} \tag{2.3}$$

Simplices will be our most basic geometry we use for formulating the descrete form of a polytope. In fact, the two- and three-simplices are already polytopes!

2.2.1.1 Orientation

Since we will eventually like to construct a functional representation of a polytope from a combinatorial form, we must consider orientation. In our case, we will consider how the construction is ordered. More specifically, how the points are specified. Let us consider the most basic case on a number line of an interval or 1-simplex. If we choose 1 and 2 to form our closed interval we may express this as [1,2] or [2,1]. In this case the result really would not change to any practical effect, and more so [2,1] may be considered an incorrect representation in some dicisplines. This example simply shows how ordering may be communicated symbolically.

In the case of the 2-simplex it becomes more clear. Given three unique points a, b, and c specifying a triangle we have two possible orderings. Namely a "clockwise" and "counterclockwise" orientation. We see in effect the orientation of the edges is the same in the forumulations $[a, b, c]$, $[c, a, b]$, and $[b, c, a]$, since the edges $[a, b]$, $[b, c]$, and $[c, a]$ are all formed identically. However the formulations $[c, b, a]$, $[a, c, b]$, and $[b, a, c]$ form the edges $[b, a]$, $[c, b]$, and $[a, c]$. This is opposite to the first formulation we specified!

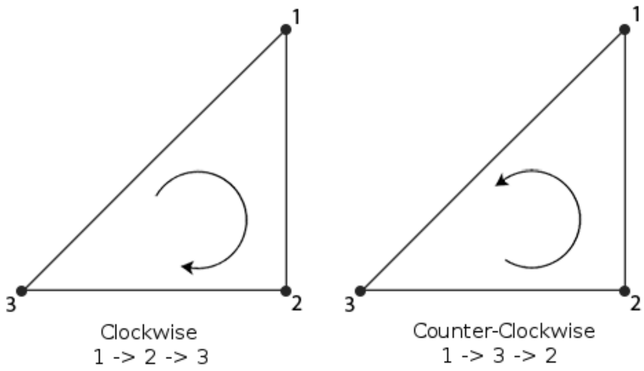

Figure 2.2: The two possible orientations of a triangle given 3 points.

Thus we may define the orientation of an simplex as a class of orderings taken by even permutation (cite Hempel Thesis). More so we see that the orientation of the edges follow from the orientation of the simplex. This property will be important when we analyze the structure of Polyhedra.

[5]

We now have the tools neccesary to construct a combinatorial representation of a polytope. In the functional representation we were interested in extracting geometric information about points and their distances from the polytope. The combinatorial representation will help us observe the relationships between the faces. To construct an N-polytope we will use N-1 simplices. using set

John M. Lee Introduction to Topological Manifolds

Simplicial Complexes

2.2.2 Construction of the Combinatorial Representation

2.3 Implicit Functional Representations of Solids

2.3.1 Hyperplanes

In multiple dimensions this is more interesting since we may construct various, rather arbitrary, geometries to make a closed space. One common example is a hyperplane. A hyperplane is simply a generalization of the a plane into arbitrary dimensions, with the property it is of dimensionality N-1. For example if we are in 2D space, our hyperplane is a line since it partitions our space into two parts. Likewise in 3D space this is a plane. If we define a hyperplane functionally using vector notation we can extract some interesting properties. For simplicity in this example let us assume we have a hyperplane which cuts through the origin. If we let $\vec{x}$ be an arbitrary point in N dimensional space, and $\vec{a}$ be the slopes of each axis, then one functional construction is simply the dot product, $dot(\vec{a}, \vec{x})$. If x is on the hyperplane the function will be equal to zero. If it is not zero, then we may determine which side of the partition the point lies on. Thus as a set representation the hyperplane is:

$$\{f(x) = dot(\vec{a}, \vec{x}) | f(x) = 0, x \in \mathbb{R}\} \tag{2.4}$$

define hyperplane sooner

We notice that the hyperplane cuts space, but does not create a closed subspace. What we are after is a "solid", which is the composition of such objects forming a closed space. In fact, this leads to a proper definition of a closed set in topology. We will say that *a set is closed if and only if it contains all of its limits*[6]. In the case of the lone hyperplane it is not bounded, so itself therefore not closed (by not having limits). Thus we must compose hyperplanes to form closed spaces!

Implicit Surface and Distance Field

If we were to functionally define the boundary of a solid a predicate of the form $f(x) = 0$ would suffice, where the solid is defined by all x. However we previously mentioned that it is useful in computation to define a function that returns infomation about the membership of a point in the solid. Further more we can define our functional solid to return a value corresponding to the shortest distance to the boundary.[7] These functions are sometimes called "implicit" forms in solid modelling, but more precisely they generate a distance field.

A sketch of this behavior in one dimension can be seen in Figure 2.4.

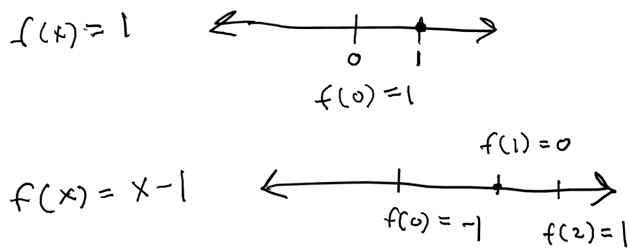

Figure 2.3: Number line illustrating the construction of an implicit signed function

2.3.1.1 Affine Transforms

Functional representations can naturally deal with affine transforms[8]. Given some transform associated with a solid, one simply applies the inverse transform to check membership. The key word here is "associated" since for our purposes we will define geometries without consideration of their tranforms. This is one aspect in which computation will aid us extensively. If we let f be a functional representation of a solid or spatial partition which generates a signed distance field, A be the transform of the object, and x be the point whose membership in the solid is in question. It then follows that the distance field formed by f transformed by A is obtained by $f(A^{-1} * x)$.

Affines are groups

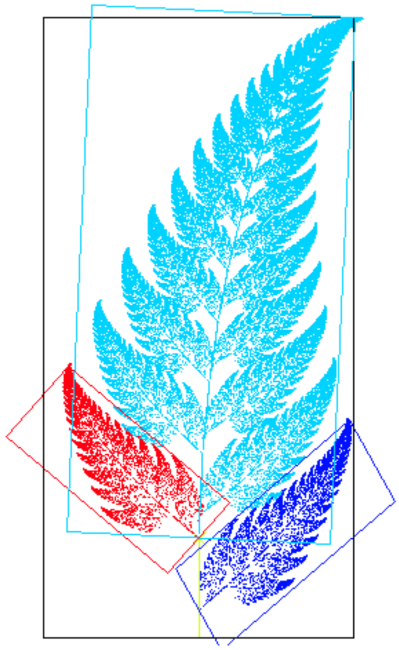

Figure 2.4: Number line illustrating the construction of an implicit signed function

2.3.1.2 Orientation

2.3.2 Set operations on Distance Fields

One can also compose distance fields with logical operations. Below are basic set operations defined for these functions using our sign conventions:

$$\cap : \mathtt{min}(f_1, f_2)$$
$$\cup : \mathtt{max}(f_1, f_2)$$
$$\neg : -\mathtt{f}_1$$

proof/cite

It follows that the "difference" of f_1 and f_2 is the intersection of f_1 with the negation of f_2, $\mathtt{max}(f_1, -f_2)$. Thus we may compose functional representations of geometry using the language of set operations, namely, union, difference, intersection, and negation. The mathematical analyst might have trouble with these formulations because such operations create discontinuities. To illucidate this problem we may use the language of norms. A distance field as we have defined it behaves like the inf-norm, or maximum norm. If we choose a point and take a projection to the surface which forms the shortest path, our distance value will be linear. This is not very useful for any kind of differential analysis since we only have first order derivatives. It may be useful for a parametric mathematical construction of a polytope to have differentiable properties. Rvachev functions provide one solution to this problem.

2.3.2.1 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.[9] 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 he generalized 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.

clean

The Rvachev construction is of hypothetical interest in the presentation, but shows how geometric constructions can converge with analytic constructions.

In 2000 Rvachev published an english-language proof of the completeness [10]

[11] In addition Pasko, et. al. have shown that Rvachev functions can serve to replace a geometry kernel by creating logical predicates. [12] 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.[13]

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". [11] 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. [14]

Such a system for analytic geometry can be developed further. In the context of an Eulerian flow field, a distance field over a function that generates partial derivatives could be a fast numerical computation method.

clean

2.3.3 Construction of the Implicit Functional Representation

We now have the mathematical concepts needed to define a polytope using the language of hyperplanes. Using vector notation we may define a hyperplane as an implicit function generating a distance field by:

$$f(x) = \frac{a \cdot x}{\sqrt{a \cdot a}} \tag{2.5}$$

where a are the slopes and x being the point in question. This function will be zero when x lies on the plane.

[15]

Chapter 3

Computational Definitions and Grammar

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 using the intersection of mathematical and computational rigor. We have chosen to use the Julia programming language due to comfort of development, and an abundance of supporting libraries for mathematical computation. In this chapter we will give a brief introduction to many computing concepts and illustrate how Julia advances them to meet our needs well.

Needs massive refactor and redux

3.1 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.[16] [17] More simply, the language is designed to be dynamic in a way that allows rapid prototyping of code and understandable to a reader, yet provides a design amicable to performance optimizations and specialization. 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.

[1] `http://opensource.org/licenses/MIT`
[2] `http://julialang.org/benchmarks`

3.2 Comparisons

Many languages are as fast as Julia but sacrifice expressibility. In Figure 3.1 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 3.2 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.[18] We will discuss these more in depth later.

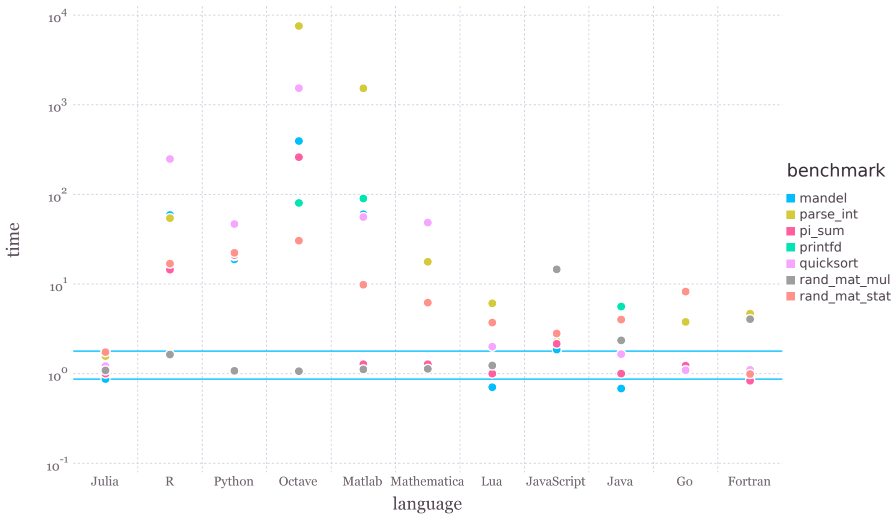

Figure 3.1: A comparison of programming languages and performance.

In 1972 Alan Kay introduced the terms "class" and "object" to describe a coupling of data and functionality.[3] An object is an instance of a class, which contains the definitions of functions and member data. 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 3.3 we can see a comparison of type systems used in scientific computing languages. Here "Type system" can be either dynamic or static, which indicates if the programmer needs to specify to the program compiler

[3] http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

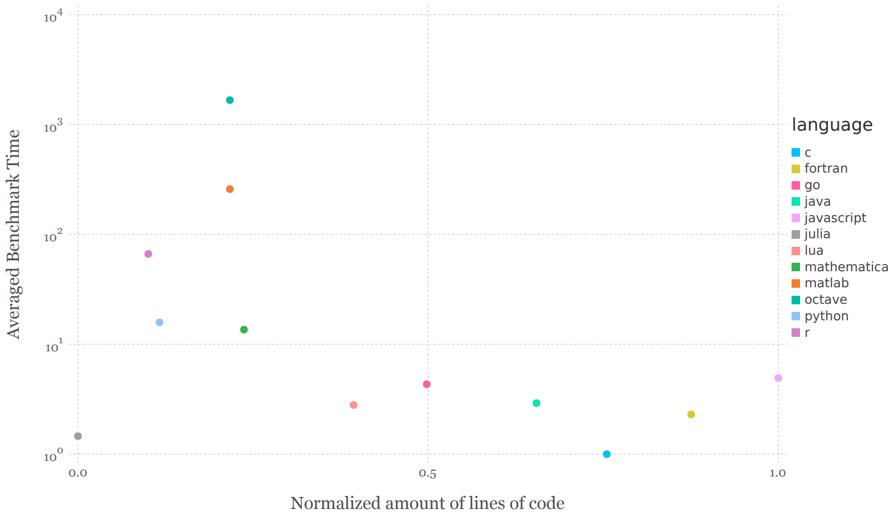

Figure 3.2: The results in Figure 3.1 normalized for code length. (Courtesy of Simon Danish)

how data is transformed in a function. Generic functions allow a single function name, for example "sum", to have multiple defintions with execution contigent upon the matching of argument types. The definition of a parametric type is more nuanced, but generally means that the definition of a type may vary based on the types of it's member data. In the next few sections these ideas will hopefully be clarified and the implications of multiple dispatch and the relation to OOP will be developed further.

Figure 3.3: 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

3.3 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 by the compiler without user intervention. This is an idea first utilized in the Hadley Milner's "ML" which was created to develop theorem provers.[19]. A basic example of inference in Julia is shown below:

```
julia> increment(x) = x + 1
increment (generic function with 1 method)
```

```
julia> increment(1)
2

julia> increment(1.0)
2.0
```

[4] 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>
}
```

[4]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.

```
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.

3.4 Types

3.4.1 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.

3.4.2 Parameters

need to demonstrate why this is HUGELY important for performance and expression

3.5 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.[20] 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. [21]

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 before compilation.

Generated functions perform a similar function as macros, but at the function level. They enable source code to be procedurally generated based on types. Surveys of Computer Science literature show that such a concept is new.

More on generated functions. Not sure if they should even be mentioned since they are usually unnecessary

3.5.1 Example

[22]

3.6 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. [23] Such tools complicate software development and are not an accessible option for the casual researcher.

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

One of the most common problems formulated is to determine whether or not a point is collinear with a line segment. Shewchuk has one of the most pragmatic and robust treatments on this topic.[24] Kettner, et. al. have also developed more examples where numerical robustness is critical. [25]

Julia's GeometricalPredicates package [5] uses the approach outlined by Volker Springel, which requires all floating point numbers to be scales between 1 and 2.[26] This has the downside of significantly reducing the available resolution.

why

A simpler, although less applicable, approach is to work 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.

[5] `https://github.com/JuliaGeometry/GeometricalPredicates.jl`

Chapter 4

Implementation

4.1 Simplex Implementation

We began by implementing a Simplex type, defined as follows:

```
"""
A `Simplex` is a generalization of an N-dimensional tetrahedra and can be thought
of as a minimal convex set containing the specified points.

* A 0-simplex is a point.
* A 1-simplex is a line segment.
* A 2-simplex is a triangle.
* A 3-simplex is a tetrahedron.

Note that this datatype is offset by one compared to the traditional
mathematical terminology. So a one-simplex is represented as `Simplex{2,T}`.
This is for a simpler implementation.

It applies to infinite dimensions. The sturucture of this type is designed
to allow embedding in higher-order spaces by parameterizing on `T`.
"""
immutable Simplex{N,T} <: AbstractSimplex{N,T}
    _::NTuple{N,T}
end
```

With the definition in GeometryTypes, we afford ourselves two notions of dimensionality. Our first parameter ‘N‘ gives us the total dimensionality of the simplex. ‘T‘ is the type of the points. For example in Julia we can prefix a colon to an identifier and make it a symbol which is reflected in the type information:

```
julia> using GeometryTypes

julia> Simplex(:x,:y,:z)
GeometryTypes.Simplex{3,Symbol}((:x,:y,:z))
```

Symbolic representation will allow us to create parametric geometry. Likewise we can construct concrete types:

```
julia> Simplex(Point(0,0,0), Point(1,1,1))
GeometryTypes.Simplex{2,FixedSizeArrays.Point{3,Int64}}((FixedSizeArrays.Point{3,Int64}((0,0,0)),FixedSize
```

This last example illustrates how this type can give us extra generalization. Here we have constructed a line segment in 3D space. The Simplex is of size two but the space it occupies is three dimensional. This way it acts similar to a fixed size vector, but the type implies all points are on the convex hull. (Should update decription to be accurate here).

Below is an example of a high performance implementation of Simplex decomosition:

```
"""
Decompose an N-Simplex into tuple of Simplex{2}
"""
@generated function decompose{N, T1, T2}(::Type{Simplex{2, T1}},
                                         f::Simplex{N, T2})
    # other wise degenerate
    2 <= N || error("decompose not implented for N <= 2 yet. N: $N")

    v = Expr(:tuple)
    append!(v.args, [:(Simplex{2,$T1}(f[$(i)],
                                      f[$(i+1)])) for i = 1:N-1])
    # connect vertices N and 1
    push!(v.args, :(Simplex{2,$T1}(f[$(N)],
                                   f[$(1)])))
    v
end
```

4.2 Issues with Existing Polyhedral Mesh Types

Prior to this project, GeometryTypes primarily provides for Polygonal Mesh type that is well tuned for operations on the CPU and GPU. It is defined as follows:

```
"""
The `HomogenousMesh` type describes a polygonal mesh that is useful for
computation on the CPU or on the GPU.
All vectors must have the same length or must be empty, besides the face vector
Type can be void or a value, this way we can create many combinations from this
one mesh type.
This is not perfect, but helps to reduce a type explosion (imagine defining
every attribute combination as a new type).
"""
immutable HomogenousMesh{VertT, FaceT, NormalT, TexCoordT, ColorT, AttribT, AttribIDT} <: AbstractMesh{Ver
    vertices            ::Vector{VertT}
    faces               ::Vector{FaceT}
    normals             ::Vector{NormalT}
    texturecoordinates  ::Vector{TexCoordT}
    color               ::ColorT
    attributes          ::AttribT
    attribute_id        ::Vector{AttribIDT}
end
```

The first thing to note is the provisions for attributes, colors, and textures. These are used for mapping textures and/or colors to polygons via APIs such as OpenGL. We do not need these (at least yet) in a rigourous mathematical definition. Likewise, in a HomogenousMesh we structure the realization as follows: 1. Insert all vertices of the mesh into ‘vertices‘ 2. Construct Faces of at least 3 indices referencing the points in ‘vertices‘.

This gives us certain properties that are nice for computation. Primarily this allows us to observe the combinatorial properties of the mesh by analyizing the faces. In addition, this compacts the data representation of vertices since shared vertices can be represented with a common face index. Affine transforms only need to operate on the vertices. Thus, in the optimistic versus pessimistic case these operations can be 3x faster with this layout assuming all vertices are in 3 faces.

4.3 Parametric Triangle

4.4 Signed Distance Field

4.4.1 Signed Distance Fields

rewrite to be salient

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.[27] 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. [28] Many algorithms for generating polyhedra from an SDF exist. The most common are Marching Tetrahedra, Marching Cubes, and Dual Contours.[29][30][31]

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.[32] DiFi was introduced in 2004, which demonstrates an algorithm for creating SDFs on multiple types of geometry [33].

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.

[34]

4.5 Polytope

4.6 Combinatorial Operations

Chapter 5

Conclusion

5.1 Future Work

5.1.1 Automatic Differention of Solids

```
julia> using DualNumbers

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

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

5.1.2 Ray Tracing and Marching

When we look at the natural world we observe the propogation of light energy. Our eyes recieve this light energy in the form of photons. The study of ray tracing seeks to mimic such behavior for computer visualizations and simulations.

leave this section, would be good to discuss angle-based polyhedra and or deficits for these ops

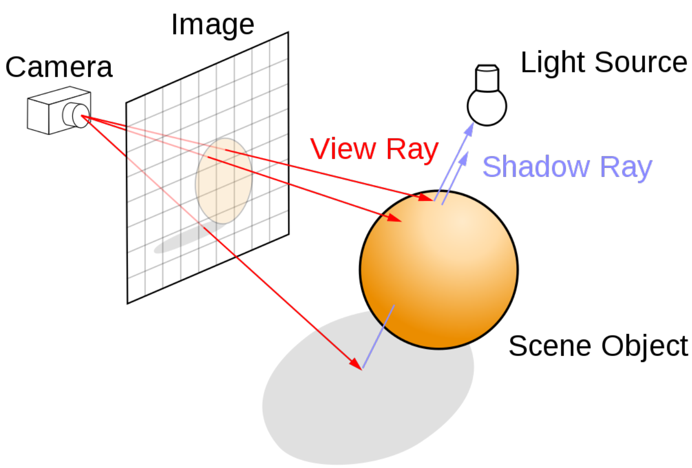

Figure 5.1: An illustration of a Ray tracing.[1]

Íñgo Quílez has done some of the most accessible work on real-time ray tracing. His technique is called ray marching, and leverages the properties of functional geometry.[35]

5.1.3 Engineering Solid Analysis

[1] By Henrik (Own work) GFDL or CC BY-SA 4.0-3.0-2.5-2.0-1.0, via Wikimedia Commons

Bibliography

[1] H. S. M. Coxeter, *Regular Polytopes.* Dover Publications, 1973.

[2] H. Gluck, *Almost all simply connected closed surfaces are rigid*, p. 225–239. Springer, 1975.

[3] R. Connelly, "A counterexample to the rigidity conjecture for polyhedra," 1977.

[4] M. Hempel, "An attack on flexibility and stoker's problem," *arXiv preprint arXiv:1512.05230*, 2015.

[5] J. M. Lee, *Introduction to Topological Manifolds*, vol. 202 of *Graduate Texts in Mathematics.* Springer New York, 2011.

[6] W. Rudin, *Principles of Mathematical Analysis (International Series in Pure and Applied Mathematics) (International Series in Pure & Applied Mathematics).* McGraw-Hill Education, 1976.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15]

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

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

[18] J. Chen and A. Edelman, “Parallel prefix polymorphism permits parallelization, presentation & proof,” in *Proceedings of the 1st Workshop on High Performance Technical Computing in Dynamic Languages*, (New York, NY), ACM, 2014.

[19] R. Harper, *Programming in standard ML*. 1997.

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

[21] C. Hoare, “Hints on programming language design.”

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

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

[24] J. Shewchuk, “Fast robust predicates for computational geometry.”

[25] 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.

[26] 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.

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

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

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

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

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

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

[33] 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.

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

[35] I. Quilez, "Rendering worlds with two triangles with raytracing on the gpu in 4096 bytes," Aug 2008.