DEPARTMENT OF COMPUTER SCIENCE

Warm Fuzzy Things

A Domain Specific Language for Hypertexture

Samantha Frohlich

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Sunday 12th May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of BSc in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Samantha Frohlich, Sunday 12th May, 2019

Abstract

Hypertexture is an Oscar award winning method for describing three dimensional shapes. It allows for exciting phenomena such as fire and hair to be represented realistically in a digital format. A key issue with hypertexture is that it is unintuitive and previous work has focused on beautiful results, not giving much information as to how they were achieved.

This thesis makes the implementation the focus by proposing a hypertexture domain specific language. This language aims to make hypertexture easier to reason about and assist future creations of hypertexture implementations by providing a uniform interface.

The language has been created using type class morphisms, where the design process is guided by a mathematical model, to provide an implementer with a clear specification of what a correct implementation consists of. It is presented in the form of a Haskell type class, accompanied with denotational semantics to make the purpose of each function unambiguous. To illustrate the success of using this method, example implementations are also provided as part of this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

A continuous challenge of computer science is striving to find ever more realistic ways of faithfully describing the real world. From three dimensional models that help to build houses, to games to fight boredom, to meaningful fictional characters that stay with people, computer representations of the real world touch everyone.

To create these tools or pieces of art, continuous values of the real world must be converted into discrete binary values on a computer. Normally a three dimensional model is represented as points in a three dimensional space that are either part of an object, or outside the object. However, this is a simplification that robs three dimensional models of more complex phenomena such as clouds, fire or fur that cannot be categorised in such a binary manner.

In 1989, Ken Perlin introduced the notion of hypertexture [13], providing a way of including such exciting features. Hypertexture describes the boundary between the solid object and the outside world, for example, hair. This is achieved by giving all points a density value between 0.0 and 1.0, where 1.0 is part of the object and 0.0 is not. For example, a cloud could be created from points arranged in a fluffy shape that all have a density of 0.4.

Hypertexture has helped the film industry create engaging and realistic CGI shots. Any film that includes explosions or computer generated animals is bound to have used ideas introduced by hypertexture. For example, Pixar's Ratatouille uses a hypertexture to model the inside of bread [3]. Hypertexture makes use of pseudorandom values to create a natural and organic effect, perfect for a convincing blooming of bread. One method of creating these pseudorandom values is Perlin noise, which was so impressive that it earned a Technical Achievement Award from the Academy of Motion Picture Arts and Sciences [1] in 1996.

Undoubtedly the notion of hypertexture has influenced the software of some of today's most successful visual effects companies, making it an appealing target for the creation of a language.

1.2 Contributions

The focus of the 1989 "Hypertexture" [13] paper was the impressive results, providing very little detail about the underlying implementation. A frustrating situation for anyone with the curiosity to wonder how such breathtaking images of complicated objects such as fireballs were created.

This thesis shifts the focus towards the implementation. It provides a hypertexture domain specific language as a uniform interface for hypertexture implementations. The language is based on the idea of type class morphisms [4], meaning that the language itself gives clear guidance on what would make a correct implementation through the provision of denotational semantics. It will be clear and unambiguous how each function should behave. The language will also be based on a mathematical model, aiming to tackle the problem with hypertexture that it is difficult to control because of its unintuitive nature [5]. This thesis also covers some example implementations to demonstrate these features.

1.3 Criteria for Success

This thesis can be considered a success if it shifts the focus away from the results and makes the implementation the feature, by creating a language that:

- **Clearly Specifies Behaviour** - this is key to making the language useful. It should be unambiguous how the language behaves and what it does. This provides two things: guarantees for programmers that the computer will act as expected, and a good foundation for collaborative reasoning as a common ground of understanding is shared.
- **Is Mathematically Based** - it is desirable for programming languages to be mathematically based because it makes them easier to reason about because tools from the world of mathematics can be used. For example, mathematical proofs can be written guaranteeing certain behaviours and providing reliability.
- **Describes Hypertexture** - the purpose of this language is to describe hypertexture. It should achieve this. If the language can describe hypertexture mentioned in the "Hypertexture" paper [13], then this goal will have been achieved.
- **Makes Correct Implementations Easy to Create** - implementations are needed to use the language. These implementations should be correct to make use of the mathematical basis, and it should not be extremely difficult to create them, otherwise the langauge will not be used.
- **Makes Hypertexture Easier to Control** - currently hypertexture is unintuitive [5]. Ideally this language does not continue this. The main reason for designing a language for hypertexture was to provide an easier expressive way of interacting with it.

1.4 Structure of Thesis

This thesis will tell the story of the creation of the hypertexture language.

The story will begin with Chapter 2, where the groundwork of knowledge needed for the thesis will be laid out. The mathematical and technical knowledge needed will be explained, then the work that this thesis is based on will be presented, giving more details on what hypertexture is and how type class morphisms work. This chapter will be rounded off with a review of related papers.

Next, armed with this knowledge, this thesis will explain how the language was created in Chapter 3. Its different iterations will be explained, and the design choices made will be justified. Then the language's final form will be explained and its denotational semantics given. The process and language will then be evaluated, exploring if the language meets the success criteria specified in 1.3.

Chapter 4 will then provide details of the implementation provided as part of this thesis. All code is written in Haskell and includes a deep embedding, and some example implementation specified to different domains, such as the modelling environment Maya, showcasing different features of the language. This paper will then explain how hypertexture of this thesis is rendered, and all these elements will be evaluated.

After the main body of work has been presented, Chapter 5 will assess the success of the thesis and highlight any areas that could be improved upon. This leads nicely onto Chapter 6 where future work will be suggested and discussed. Finally, the whole thesis will be summarised in Chapter 7.

Chapter 2

Background

This chapter will summarise any mathematical or technical knowledge needed to understand the content of this thesis, including the mathematical models of the language (fuzzy sets and implicit functions), and functional programming techniques that will be used to create the language. It also outlines the previous work that this thesis is directly building upon. Then this chapter will fully explore what hypertexture is, and explain the denotational design method used. Finally, related papers and work in this area will be reviewed.

2.1 Technical Background

2.1.1 Fuzzy Sets

What In mathematics, a fuzzy set is a set where membership is not binary [16][6]. In normal sets an object is either an element, or it is not. Fuzzy sets introduce the idea that objects can be partly an element. Membership is described as a continuous value between 0.0 and 1.0.

Consider a Quidditch club. Its set of members could be expressed in a binary fashion where a person is either a member or they are not. Alternatively, this Quidditch club could have different tiers of membership: a full member (1.0), a social member (0.7), a previous member (0.5), a supporter (0.3), or just not a member (0.0).

Details Fuzzy sets are specified using a *characteristic function*, which specifies the membership value of an object. For example, the universal set (the set that contains everything) would be described by the characteristic function that always returns one:

```
characteristicFunction = const 1.0
```

Or, continuing the Quidditch club example, this characteristic function could look up a person's name, look up what membership they have bought and return the relevant membership value.

Common set properties such as *union* and *complement* are also lifted to the context of fuzzy sets. To perform these operations the characteristic functions are adapted:

$$\begin{aligned} union\ \mu_a\ \mu_b &= \lambda p \rightarrow max\ (\mu_a\ p)\ (\mu_b\ p) \\ complement\ \mu &= \lambda p \rightarrow 1 - (\mu\ p) \end{aligned}$$

Union takes two characteristic functions and applies each to a candidate element. The membership given to this element is the maximum of these two values. If an element was only a member of one set the point would also be in the *union* of the sets because the maximum of one and zero is one. *Complement* works by taking the value produced by the existing characteristic function and inverting it by taking it away from one. A membership of 0.7 would become a membership of 0.3.

Relation Fuzzy sets form the perfect mathematical model for hypertexture. In the same way fuzzy sets extend normal sets from binary membership to continuous membership, hypertexture extends the idea of three dimensional shapes to include objects with density values between 0.0 and 1.0. A hypertexture can be seen as a fuzzy set of points, where the membership is the density at that point.

2.1.2 Implicit Functions

What Mathematical functions can be represented implicitly or explicitly. The explicit representation, where one of the variables is the subject of the equation, is more common and normally takes the form of $y = f(x)$. Unlike this representation where the dependent (y) and independent (x) variables are split up, implicit functions are described using all variables together.

Uses Implicit functions can be used to describe shapes. Two dimensional shapes can be described using an implicit function with two arguments (e.g. $f(x, y)$), with three dimensional shapes requiring 3 arguments (e.g. $f(x, y, z)$). When this function is evaluated: a value of zero indicates that that point lies on the edge of the described shape; a value less than zero indicates the point is inside the shape; and a value greater than zero is outside the shape. Essentially, implicit functions provide a way of mapping polygons to the number domain.

Example Explicit unit circle: $y^2 = 1 - x^2$ Implicit unit circle: $f(x, y) = x^2 + y^2 - 1$

Relevance This representation is particularly useful for providing a quick and easy test to see if a point is in, on the edge of, or outside a shape. All that is required is a comparison to zero as specified above. For example, a set of points within a shape could be constructed using a characteristic function comprising of this test, where each candidate point is given to the implicit function, evaluated, and if the value is less than or equal to zero a membership of one is given.

2.1.3 Domain Specific Languages

What Unlike General Purpose Programming Languages, Domain Specific Languages (DSLs) focus their attention on the domain of one problem, exchanging generality for the ability to be specialised to the domain of the problem. For example, HTML is a DSL: while it cannot do everything, it is very good at describing the structure of webpages.

Types DSLs can either be created from scratch or embedded into a host language as an eDSL. Building one from scratch requires the creation of vital language elements like a parser and a complier; while embedded DSLs just borrow these from the host language. To avoid the great overheads of the former, this DSL will be created as an embedded DSL.

There are two extremes of DSLs: deep embeddings and shallow embeddings. In a shallow embedding the first class feature of the host language holds the semantics of the language (e.g. functions in Haskell); in a deep embedding data types are created and semantics are derived from them via semantic evaluation functions (e.g. Haskell data types).

Relevance Since the goal of this thesis is to design a hypertexture language that presents a uniform interface for hypertexture implementations, the language will take the form of a DSL. Focusing the domain should avoid the clutter and confusion created by unnecessary features, and create a more abstract language that allows the user to explore what can be done with hypertexture, worrying less about the fine details of how it will be done. These fine details of bespoke code that does the specific job of creating a particular type of hypertexture can be abstracted away. This should result in an expressive language where the implementation naturally falls out of the semantics of the language.

2.1.4 Functional Programming

The hypertexture description is based around the composition of functions, meaning that it lends itself perfectly to a higher order programming language and the functional style of programming. The snippets of code provided in the "Hypertexture" paper [13] even look as though they have been written in a functional programming language.

Because of this, this thesis uses the functional programming language Haskell for the embedding. Being in the functional domain should also encourage a close relationship to the mathematical semantics of the language. Haskell's similarity to denotational semantics will also contribute to this language's ability to clearly specify how a correct implementation should look.

2.1.5 Functors

Functors are the class of things that can be mapped over. Any data type that one can imagine to be a box is likely to be a functor. Functors have kind $* \to *$, which is helped by the image of a box since they can be thought of as a container for starry things. In Haskell, functors are encapsulated in a type class:

```
class Functor f where
  fmap :: (a -> b) ->  f a -> f b
```

For something to be a functor, it must provide a definition of the function `fmap`, which given a way to transform something of type `a` to something of type `b`, will perform this transformation to everything contained in the functor `f`. Syntactic functors, which will be used later, are functors that describe the syntax of a language and are functorial on the continuation point (i.e. the `k` is what changes type from `a` to `b`).

2.1.6 Language Design Tools in Haskell

Haskell is a language well suited for hosting a deep embedding because a modular language can be created using recursion schemes. Using recursion schemes keeps the data types simpler by avoiding confusion caused by mutual recursion. The recursion is put into a box making it easier to reason about. This section will detail tools that could be used to create a deep embedding in Haskell. See 2.1.7 for an example.

Fix Fix is used to remove recursion from a data type. Anywhere a recursive call is normally made is replaced by a continuation point to create a syntactic functor, then a special data type called *Fix* provides what the continuation should be as a type parameter.

```
data Fix f = In (f (Fix f))
```

This data type has one constructor called `In` that carries the functor `f`, where `f` has been told that its continuation point will have type `Fix f`. If a language is created with `Fix` then this is the only place recursion will appear.

Cata A catamorphism is a way of crushing a data structure into a value. These can be used as a way of providing semantics to a deep embedding.

```
cata :: Functor f => (f b -> b) -> Fix f -> b
cata alg (In x) = alg . fmap (cata alg) $ x
```

Syntactic functors can be crushed step by step using a function called an *algebra*. Given a syntactic functor with the recursive part already crushed to the correct semantics the algebra knows what to do next based on the current constructor. The function `cata` can be seen to work by first unwrapping the `In` constructor, then crushing any recursive calls by fmapping itself, and finally using the algebra to obtain the semantics of the current level.

:+: To add modularity to the language, syntactic functors can be composed using the coproduct functor:

```
data (:+:) f g k  = L (f k)
                  | R (g k)
```

It takes two functors `f` and `g` and uses two constructors to allow representation of either `f` or `g`. These can be chained together by making `f` or `g` coproducts, allowing for many parts of a language, and any part of this coproduct can be a continuation point.

Classy Algebras One convenient way of giving a language multiple semantics is to encapsulate algebras into a type class. Then each desired semantics can take the form of a new type with an algebra instance.

```
class Functor f => Alg f a where
  alg :: f a -> a
```

The above takes the idea of an *algebra* function and wraps it up in a Haskell type class. It demands that members of this type class must have a function `alg :: f a -> a`. The function takes the syntactic functor `f` with its continuation point already crushed into an `a` and crushes the structure into an `a` based on the constructor of `f` and what the `alg` says to do.

It is best to wrap this function up into a type class, rather than just having it as a stand alone function because then the result can be dictated by the desired output type. Adding new semantics can also be achieved without changing old code through the addition of a new type.

2.1.7 Language Design Tools in Haskell - Example

Consider the following language that describes numbers:

```
Peano ::= Z
        | S Peano
```

The process of creating a Haskell deep embedding for it using the tools from 2.1.6 would consist of the following steps:

1. Creating a Haskell data type.
2. Breaking this data type up into its component parts.
3. Fixing the data types, and providing Functor instances to create a syntactic functor.
4. Writing `Alg` instances.

Step One Since Haskell data types are so similar to Backus Normal Form (BNF) notation, the desired code falls out naturally, with one constructor per case:

```
data Peano = Z
           | S Peano
```

Step Two Each constructor should be cordoned off into its own data type. This change would be far more drastic with a more exciting language, and the constructors can be grouped semantically. For example, if the language also included adding and subtracting there could be two modules to the language: the number part, and the operations part. To represent the language as a whole the `:+:` functor from 2.1.6 is used.

```
type Peano = (Z :+: S)
newtype Z = Z
newtype S = S Peano
```

Step Three To make the types compatible with `Fix`, a continuation point (denoted `k`) should be added as a type parameter. Every recursive call formerly to `Peano` should be replaced by this `k`. Since `Z` has no recursive call the `k` does not need to be mentioned on the right hand side. Note that `Fix (Z :+: S)` will be equivalent to the initial Peano type. Additionally, Functor instances should be derived so that when the semantics are to be applied using `cata` it can use `fmap`.

```
type Peano = Fix (Z :+: S)
newtype Z k = Z deriving Functor
newtype S k = S k deriving Functor
```

Step Four Finally, semantics can be added to the language. To apply the semantics an `Alg` instance for each component part should be created. An `Alg` instance will also be needed for `:+:` to glue it all together.

```
instance (Alg f a , Alg g a) => Alg (f :+: g) a where
  alg (L x) = alg x
  alg (R x) = alg x

instance Alg Z Int where
  alg :: Z Int -> Int
  alg _ = 0

instance Alg S Int where
  alg :: S Int -> Int
  alg (S n) = n+1
```

The `alg` instance for `:+:` demands that each syntactic functor already has an `alg` instance then directs the program towards which one to use by unwrapping the `L` or `R` constructor. The syntactic functor `Z`'s `alg` instance applies the semantics of the number zero to it. `S`'s `alg` instance takes the value of the previous number and increases it by one.

Using the Language Now the semantics can be given to the langauge using `cata`:

```
> cata alg (In (R S (In (L Z))))
1
```

To get this result `cata` would have crushed the data type, step by step starting from the middle:

```
  cata alg (In (R S (In (L Z))))
= {def. cata}
  alg . fmap (cata alg) $ (R S (In (L Z)))
= {def. fmap}
  alg $ (R S (cata alg (In (L Z))))
= {def. cata}
  alg $ (R S (alg . fmap (cata alg) $ L Z))
= {def. fmap}
  alg $ (R S (alg $ L Z))
= {def. alg}
  alg $ (R S (alg Z))
= {def. alg}
  alg $ (R S 0)
= {def. alg}
  alg $ (S 0)
= {def. alg}
  0+1
= {def. (+)}
  1
```

`fmap` goes right to the bottom, creating a structure littered with the `alg` function, that if evaluated will produce the result semantics.

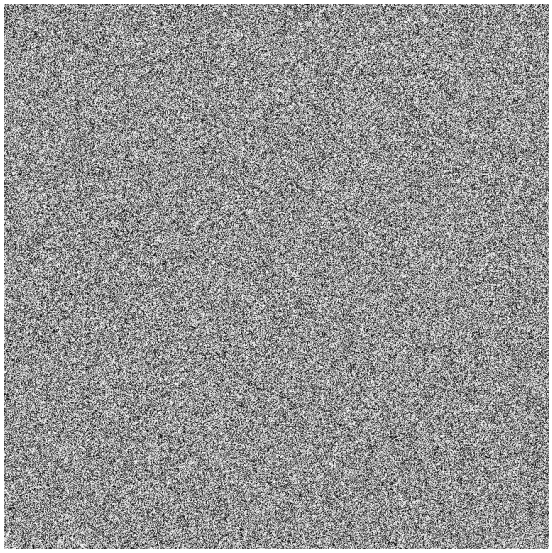
Alternative Semantics If instead of this language counting up normally, it was desired that it went up in twos, new semantics could be achieved using a `newtype` wrapper around `Int`:

```
newtype Twos = Twos Int

instance Alg Z Twos where
  alg :: Z Twos -> Twos
  alg _ = 0

instance Alg S Twos where
  alg :: S Twos -> Twos
  alg (S (Twos n)) = Twos (n+2)

> cata alg (In (R S (In (L Z))))
2
```

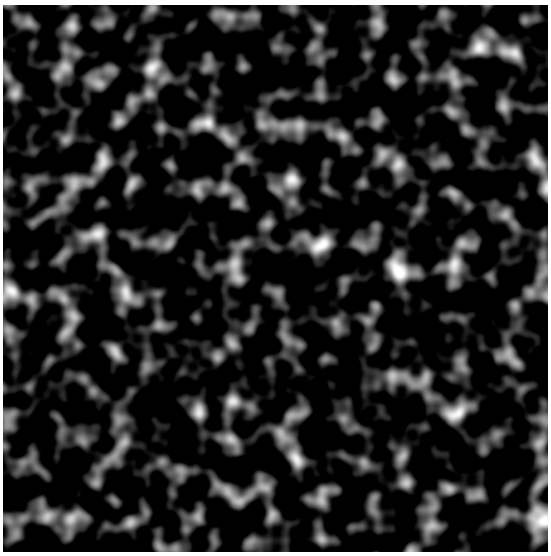
2.2 Previous Work

2.2.1 Perlin Noise

Why One of the selling points of hypertexture is that it appears natural. This is achieved by using Perlin noise, which was first introduced in the 1985 paper "An Image Synthesizer" [11]. Unlike normal noise functions where all the values are completely random, the values of Perlin noise are locally similar, creating a random yet natural effect (see figure 2.1).



Random Noise

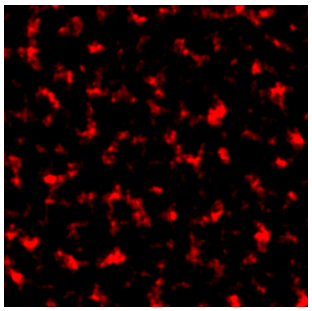


Perlin Noise

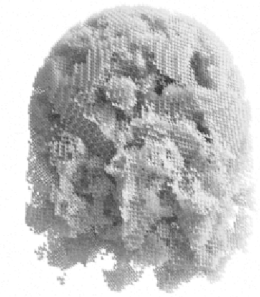
Figure 2.1: Random noise vs. Perlin noise.

Uses Perlin noise has many uses as it can be implemented in numerous dimensions (see figure 2.2). For example, it can be used in one dimension to give the effect of handwriting because it gives lines an organic wave. It can also be used in two dimensions to create textures of fire and wood, and as this thesis explores, one of its three dimensional applications is as part of hypertexture.

One Dimensional "Handwriting" [2]



Two Dimensional "Fire"



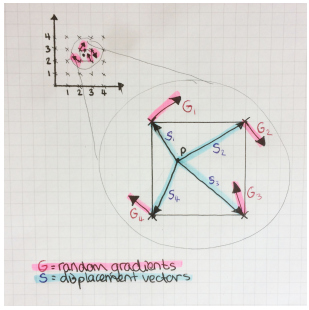
Three Dimensional Hypertexture

Figure 2.2: Examples of Perlin noise in different dimensions.

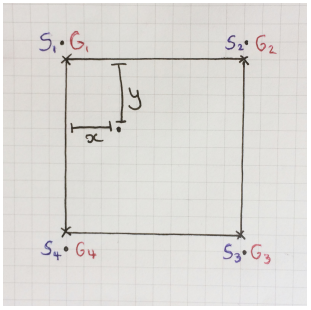
How Two dimensional Perlin noise works by defining a grid of cells over the two dimensional space and assigning each point a random gradient. The noise value of anywhere in the grid can then be found by:

1. Finding the four displacement vectors from where the noise is to be calculated to the corners of the cell surrounding it.
2. Calculating the dot product of the displacement vectors with their corresponding random gradients.
3. Interpolating these values.

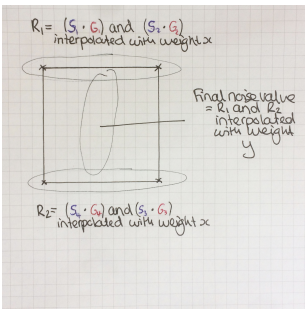
The dot product is implemented as a linear interpolation, giving us a weighted average of the surrounding gradients as the final noise value. The weight of the interpolation can be thought of as the coordinates of the point within its cell, where the x value is how far left/right the point is within its cell, and the y value is how far up/down it is. In two dimensions the interpolation is done in two stages: horizontally, then vertically. The top two gradients and the bottom two gradients are horizontally interpolated with a weight of x, then these results are interpolated based on y.



Getting displacement vectors.



Getting interpolation weights.



Interpolation.

Figure 2.3: Process for calculating Perlin noise at a point.

Improvements To increase the natural look an extra step was added to smooth the noise. Instead of using the x and y values as they are, these values are first smoothed by applying a "fade" function to them. This fade function is a cubic function (see `fade1`) and creates the desired, organic look.

In 2002 [12][10], the noise was further improved in two ways: firstly the fade function was changed to a 5th order polynomial (see `fade2`) and secondly Perlin realised that gradients did not have to be entirely random. Instead of being random they can be created from a permutation of scalar values. This thesis uses these improvements.

```
fade1 :: Double -> Double
fade1 t = t * t * (3 - 2*t)

fade2 :: Double -> Double
fade2 t = t * t * t * (t * (t * 6 - 15) + 10)
```

2.2.2 Hypertexture

What Hypertexture is Ken Perlin's proposed way of describing three dimensional shapes [13], and the focus of this thesis. In order to properly create a domain specific language for hypertexture it is essential to understand how the representation works. The key part of hypertexture is that shapes are not modelled as a surface, but rather as density values. The phenomena of hypertexture lies in areas on the boundary of the object with a density between 0.0 and 1.0, the soft region, where 0.0 is not part of the object and 1.0 is solidly the object. Intuitively, fire is hypertexture because it is not entirely solid.

Parts The description of hypertexture is made up of two types of function. The first type is called the Object Density Function (ODF) and defines the density of points within a three dimensional region as values with range [0,1]. A classic example of this is the soft sphere (see below for Haskell version) [5], where the density of a point depends on how far away from the centre of the sphere it is. The function takes the radius and centre of a circle and for each point: it finds how far the point is away from the centre, if it is outwith the radius the point is given a density of 0.0, if not the point is given a density proportional to how far away it is, precisely `d/r`. The second type of function is a Density Modulation Function (DMF). These are applied to ODFs and affect the soft region, turning a normal shape described by an ODF into a hypertexture. Hypertexture is created by applying a variety of DMFs to an ODF.

```
softSphere :: Double -> Point -> Point -> Density
softSphere r centrePoint pointToFindNoiseAt
  | d < r     = (1 - d/r)
  | otherwise = 0
  where
    d = euclideanDist centrePoint pointToFindNoiseAt
```

The paper goes on to describe how density functions of soft objects can be combined and provides numerous examples of DMFs and how they can be used to create Oscar winning graphics. One goal of this thesis is to replicate these results with the domain specific language.

2.2.3 Denotational Design

The method used to design the semantics for the hypertexture language follows that described in "Denotational design with type class morphisms" [4]. It was selected because it combines the idea of type classes in Haskell with denotational semantics, which map the meaning of language key words to their mathematical meaning. Conquering the pitfalls of ambiguity associated with type classes, the addition of the denotational semantics allows the language designer to specify exactly how an implementation of the language should behave.

Process The method uses a *type class morphism*, which consists of:

1. Selecting a mathematical model for what is to be described.
2. Adding operations. The meaning of added operations should be the meaning of those operations applied to the selected mathematical model.

Example If a language for a key to value map was to be created. The mathematical model could be selected to be partial functions because the designer of the language wants any implementations to fail gracefully. Now the meaning of any operation created for maps has the meaning of that operation on partial functions e.g. the meaning of *empty* for maps is the meaning of *empty* on partial functions. This is expressed in two parts. The model:

$$[\![\ Map\ k\ v\ ]\!]\ =\ k\ \rightarrow\ Maybe\ v$$

And semantic function translating a $Map\ k\ v$ to its semantic meaning:

$$\begin{aligned}
[\![.]\!] &:: Map\ k\ v\ \rightarrow (k\ \rightarrow\ Maybe\ v) \\
[\![empty]\!] &= \lambda k \rightarrow Nothing \\
[\![insert\ k'\ v\ m]\!] &= \lambda k \rightarrow if\ (k\ ==\ k')\ then\ (Just\ v)\ else\ ([\![m]\!]\ k)
\end{aligned}$$

Benefits This method has two main benefits: the desired behaviour of each function in the type class is clear; and any laws associated with the underlying mathematical model come for free with the type class. It will allow the creation of a uniform and consistent interface for different hypertexture implementations that gives guidance to implementers and reliability to users.

2.3 Related Work

2.3.1 ReIncarnate

What "Functional Programming for Compiling and Decompiling Computer-Aided Design" [9] presents a method for breaking up a shape into its component parts including: boolean operations, affine transformations and primitive shapes. It provides an implementation called ReIncarnate that can take .sdl files and turn them into their own three dimensional shape description language .cad3.

Revelance This paper is relevant to this thesis in two ways. Firstly, it also details a language for three dimensional shapes, the semantics for which and choice of primitives have inspired elements of the hypertexture language. Secondly, the implementation itself could be used in conjunction with the hypertexture language, for more details on this idea see 6.4.

2.3.2 Other Papers

A Framework for Interactive Hypertexture Modelling [5] When working with hypertexture it is unclear how to achieve a specific visual result. The issues leading to this unintuitive nature are highlighted in this paper, allowing the hypertexture language to tackle these problems. It made it clear that the hypertexture language should be readable with clear smart constructors, signposting the user towards which Density Modulation Functions to use. Arguments to the noise function are also clarified through a data type naming what each noise setting does.

Implicit Representations of Rough Surfaces [7] Implicit representations can also be defined for rough surfaces, making them all the more appealing as a mathematical model for hypertexture. A user of the hypertexture language could use some of these techniques to import more interesting shapes to apply hypertexture to, using the `Implicit3D` constructor.

Hypertexturing Complex Volume Objects [14] Objects that are not represented by an implicit function can still have hypertexture added to them. If this was to be included in the hypertexture language, then it could be used on data like CT scan results.

State of the Art in Procedural Noise Functions [8] There are many noise functions with different properties. It was useful to verify the implementation of Perlin noise for this thesis against these properties. The variety of noise functions also presents a choice to the user: noise is defined as a `modulatePoint` function, users of the language could use any noise function they like, not just Perlin noise.

Chapter Summary

This chapter provided the key knowledge necessary to understand how the language will be made and the mathematics that it will be based upon. It introduced fuzzy sets and implicit functions because of their relevance as mathematical models of the language. Then it detailed what a DSL actually is along with how one makes an eDSL in Haskell using functional programming tools. Work that the thesis is directly extending was also thoroughly explored. This thesis will use the denotational design method to create an eDSL for hypertexture in Haskell as will be seen in the next chapter when the design process of the language is reported 3.1. Additionally, previous work in this field was reviewed for its usefulness. The work presented many ideas on how the basic hypertexture language could be extended to include not just the work from the original paper, but also more recent developments.

Chapter 3

Language

This section will cover details of the hypertexture language. Firstly, the iterations that the language went through will be discussed to justify design choices and show what possibilities were explored. Then the final structure and denotational semantics of the language will be detailed. Lastly, the final language and design process taken will be evaluated.

3.1 Design Process

A key part of this thesis was the design of the language. The semantics of the language went through many iterations and involved much deliberation over possible mathematical models.

3.1.1 First Iteration

Following the denotational design method outlined in 2.2.3, the first step in designing the language was to decide upon a suitable mathematical model. Two strong candidates presented themselves, and both were explored with a sample version of the language. It was desirable to have a mathematical model that worked well with combining hypertextures so the operations explored were *union*, *intersection*, *complement*, and *odf*, a method of lifting hypertexture to this boolean domain.

Candidates The first candidate was using fuzzy sets (described in 2.1.1) since they are alluded to in the "Hypertexture" paper [13]:

$$
\begin{aligned}
[\![.]\!] &:: Hypertexture \to Fuzzy\ Set \\
[\![odf\ \mu]\!] &= \lambda p \to \mu\ p \\
[\![union\ \mu_a\ \mu_b]\!] &= \lambda p \to max\ ([\![\mu_a]\!]\ p)\ ([\![\mu_b]\!]\ p) \\
[\![intersection\ \mu_a\ \mu_b]\!] &= \lambda p \to min\ ([\![\mu_a]\!]\ p)\ ([\![\mu_b]\!]\ p) \\
[\![complement\ \mu]\!] &= \lambda p \to 1 - ([\![\mu]\!]\ p)
\end{aligned}
$$

These denotational semantics take the stripped down hypertexture language and apply the semantics of fuzzy sets, with the language key words on the left, and the semantic meaning on the right. The semantic meaning of fuzzy sets is expressed through presenting the characteristic function of a fuzzy set, that takes a point and returns its membership. *odf* takes a function μ that, given a point, will return its membership and makes this the characteristic function. For *union* and *intersection*, μ_a and μ_b represent two hypertextures, which using the semantic brackets are reduced to their fuzzy set meaning and the maximum/minimum taken. The use of maximum/minimum follows the definition of *union* and *intersection* for fuzzy sets.

The second candidate was simply modelling hypertexture as a function from point to density:

$$
\begin{aligned}
[\![.]\!] &:: Hypertexture \to (p \ \to \ d) \\
[\![odf \ \mu]\!] &= \lambda p \to \mu \ p \\
[\![union \ \mu_a \ \mu_b]\!] &= \lambda p \to max \ ([\![\mu_a]\!] \ p) \ ([\![\mu_b]\!] \ p) \\
[\![intersection \ \mu_a \ \mu_b]\!] &= \lambda p \to min \ ([\![\mu_a]\!] \ p) \ ([\![\mu_b]\!] \ p) \\
[\![complement \ \mu]\!] &= \lambda p \to 1 - ([\![\mu]\!] \ p)
\end{aligned}
$$

These semantics operate in the same way except this time the function is just a function, not the characteristic function describing a set.

Decision Looking at the latter candidate model it can be seen that every operation can be applied to any applicative functor: *odf* can be replaced with *pure*, a function which embeds a value in a minimal context, and the other operations can be implemented using *liftA2*, which combines two similarly shaped structures using a provided function. This gives hypertexture potential to be applied in many more domains than just a simple point to density function. Perhaps hypertexture does not only exist for shapes, but also for things like music. To see if there was more to discover with this mathematical model the next iteration adds more operations to the language.

3.1.2 Second Iteration

In order to turn normal shapes into hypertexture phenomena three classes of DMF can be used:

- Position Independent (*modulateDensity*) - only requires to know the density at a point. Allows for gradients of density change to be controlled.
- Position Dependent (*modulatePoint*) - requires knowledge of the point. Allows for noise to be added to a shape.
- Geometery Dependent (*modulateGeometery*) - requires knowledge of all density values in the shape. Allows for hair to be added to the shape.

The first two mostly keep with the applicative theme with *modulateDensity* being implemented as post-composition/*fmap* and *modulatePoint* being created as pre-composition/*cofmap*. Then *modulateGeometery*, which needs to have type $(p \to d) \to (p \to d)$, breaks this notion, indicating that perhaps fuzzy sets were the better model after all.

3.1.3 Third Iteration

Using fuzzy sets as the mathematical model for the hypertexture language has the advantage that the fuzzy set properties should come for free if the denotational design method is followed. The modulation functions can work in a similar way to how they operated in the function mathematical model, except they are now operating on a characteristic function that describes a fuzzy set.

Improvements Functions like the modulate functions and *odf* have a lot of power. The *odf* operation can do everything that the language can do if a complicated enough function is lifted. A better approach would be to have primitive shapes, limiting what can be described. Being so integral to hypertexture, the modulate functions cannot simply be removed or replaced by primitives, but they can be cordoned off into their own section. Creating a modular language allows the user to pick and choose which features they want. If a user only wants to write code to describe normal shapes, they do not need the modulate functions.

This third iteration consists of four modules:

- Shapes.
- Transformations.
- Booleans.
- Hypertexture.

Shapes The Shape module uses implicit functions (2.1.2) to describe primitive shapes. These implicit functions are used as part of the characteristic function for the fuzzy set semantics:

$$
\begin{aligned}
[\![.]\!] &:: Shapes \to Fuzzy\ Set \\
[\![Empty]\!] &= const\ 0.0 \\
[\![Cube]\!] &= test\ \circ (\lambda xyz \to (max\ |x|\ |y|\ |z|) - 1) \\
[\![Cylinder]\!] &= test\ \circ (\lambda xyz \to (sqrt(x^2 + y^2) - 1) \leq 0\ \wedge\ z \geq 0\ \wedge\ z \leq 1) \\
[\![Sphere]\!] &= test\ \circ (\lambda xyz \to x^2 + y^2 + z^2 - 1) \\
[\![Prism\ p]\!] &= test\ \circ (\lambda xyz \to ([\![p]\!]\ x\ y) + (|z| - 1)) \\
[\![Cone]\!] &= test\ \circ (\lambda xyz \to x^2 + y^2 - (z-1)^2) \\
[\![Torus]\!] &= test\ \circ (\lambda xyz \to (sqrt(x^2 + y^2) - 1)^2 + z^2 - 0.5) \\
[\![Implicit\ f]\!] &= test\ \circ f
\end{aligned}
$$

These semantics consist of the *test* function and different implicit functions that represent the shape named on the left. The result of evaluating the implicit function at the candidate point is passed onto the *test* function. If the result is $\leqslant 0$ then the point is deemed in the shape and given a membership/density value of 1.0. The most interesting constructors are *Empty* and *Implicit*, which work slightly differently. *Empty* does not care what the point is, it will always return a membership of 0.0. *Implicit* will use a user defined three dimensional implicit function.

Transformations The Transformations module allows for the primitive shapes to be moved, rotated and scaled. Translations (t) take the form of a 3x1 matrix, rotations (r) are a 3x3 rotation matrix, and scale (n) is just a scalar value. This works by first performing the reverse transformation on the point before testing membership with the characteristic function:

$$
\begin{aligned}
[\![.]\!] &:: Transformations \to Fuzzy\ Set \\
[\![Translate\ t\ s]\!] &= \lambda p \to [\![s]\!]\ (p - t) \\
[\![Rotate\ r\ s]\!] &= \lambda p \to [\![s]\!]\ (r^{-1}p) \\
[\![Scale\ n\ s]\!] &= \lambda p \to [\![s]\!]\ (p/n)
\end{aligned}
$$

Booleans The Boolean module allows for the combining of hypertextures using the boolean operations of fuzzy sets:

$$
\begin{aligned}
[\![.]\!] &:: Boolean \to Fuzzy\ Set \\
[\![Union]\!] &= \cap \\
[\![Intersection]\!] &= \cup \\
[\![Complement\ s]\!] &= \overline{s} \\
[\![Difference\ s_a\ s_b]\!] &= s_a \setminus s_b
\end{aligned}
$$

Hypertexture Finally, the Hypertexture module includes all the modulate functions needed to create phenomena. It also includes a soft primitive shape:

$$
\begin{aligned}
[\![.]\!] &:: Hypertexture \to Fuzzy\ Set \\
[\![SoftSphere]\!] &= \text{softSphere 1 (0,0,0) (defined in 2.2.2)} \\
[\![ModulateDensity\ f\ s]\!] &= f \circ [\![s]\!] \\
[\![ModulatePoint\ f\ s]\!] &= [\![s]\!] \circ f \\
[\![modulateGeometery\ f\ s]\!] &= f[\![s]\!]
\end{aligned}
$$

SoftSphere will create a sphere, with radius one and centre at the origin, that gets less dense the further away from its centre a point gets. *ModulateDensity* will take a shape s and a $(density \to density)$

function *f*. It will evaluate a point using the semantics of *s* to get a density that will then be modulated by *f*. *ModulatePoint* first modulates the point using *f*, a (*point* → *point*) function, then evaluates the resulting point with the semantic meaning of *s*. *modulateGeometery* is the most powerful. Its *f* adapts a (*point* → *density*) function into another (*point* → *density*) function. The new function is then used as the characteristic function for the fuzzy set.

3.1.4 Final Iteration

The final language that is detailed in the rest of this chapter takes the modular nature of the previous iteration further, breaking the language into smaller and more specific chunks. The sections of the language have also been grouped into three different mathematical models that complement each other. Employing multiple mathematical models should allow the semantics to be more specialised towards what they are describing.

3.2 Final Structure

The hypertexture language is a modular language with many components not only allowing the user to decide which parts they want to use, but also allows for new custom components to be added easily. Each component will belong to one of three semantic levels of mathematical model:

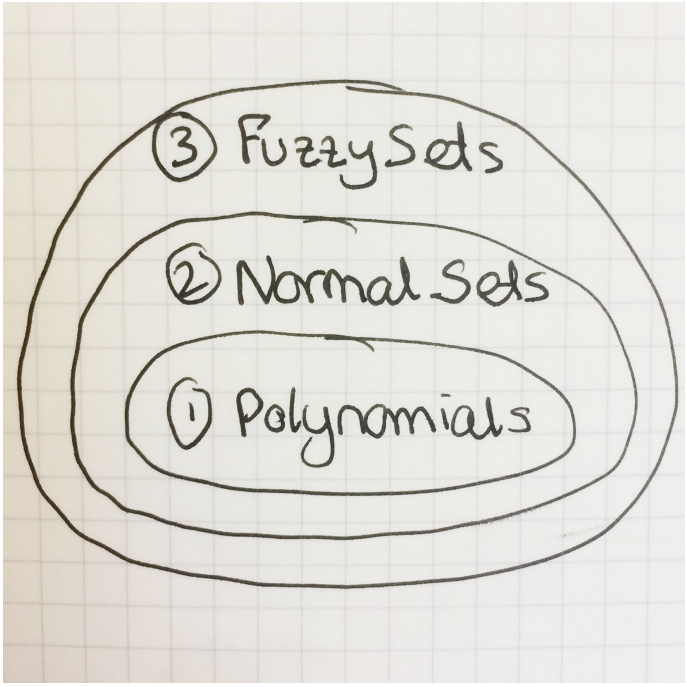


Figure 3.1: Language Semantic Layers.

Polynomials Used for the basic shapes. The two dimensional and three dimensional shapes will be modelled mathematically as polynomials using implicit functions e.g. $Circle = f(x, y) = x^2 + y^2 - 1$.

Normal Sets Used for combining shapes together. There is no need to have fuzzy sets if hypertexture is not going to be added. A subset of the language not involving these operations can live at the level of normal sets. These sets can use the polygon implicit functions as membership functions, with a point being a member if the equation evaluates to $\leqslant 0$. Operations at this level also work for fuzzy sets.

Fuzzy sets To make room for simulating hypertexture, set membership needs to be made less binary.

Here is how the different sections are distributed over these layers:

- Polynomials
  - Two Dimensional Primitives (3.3.1)
  - Two Dimensional Implicits (3.3.1)
  - Three Dimensional Primitives (3.3.2)
  - Three Dimensional Implicits (3.3.2)
  - Three Dimensional Prisms (3.3.2)
- Normal Sets
  - Booleans (3.3.3)
  - Transformations (3.3.4)
- Fuzzy Sets
  - Soften (3.3.5)
  - Soft Primitives (3.3.5)
  - Modulate Density (3.3.6)
  - Modulate Point (3.3.6)
  - Modulate Geometry (3.3.6)

3.3 Denotational Semantics

This section details the precise meaning of the language using denotational semantics.

3.3.1 Two Dimensional Shapes

These modules of the language live at the polynomial mathematical model, so all have the meaning of an implicit function with two arguments, one for each dimension. There are two options for creating two dimensional shapes: utilising a primitive or adding a custom via an implicit function.

Two Dimensional Primitives A collection of basic two dimensional shapes that will all be drawn at $z = 0$ in three dimensional space. Circle is a unit circle around the origin. Triangle is a unit length right angled triangle with the right angle situated at the origin. Square has side length two with its corners on the axes. For images of these shapes see Appendix A.

$$\begin{aligned} [\![.]\!] &:: Prim2D \to f(x,y) \\ [\![Circle]\!] &= sqrt(x^2 + y^2) - 1 \\ [\![Triangle]\!] &= x + y - 1 \\ [\![Square]\!] &= |x| + |y| - 1 \end{aligned}$$

Two Dimensional Implicits Allows the user to include their own custom two dimensional shapes given as implicit functions.

$$\begin{aligned} [\![.]\!] &:: Implicit2D \to f(x,y) \\ [\![Implicit2D\ f]\!] &= f \end{aligned}$$

3.3.2 Three Dimensional Shapes

Also situated within the polynomial mathematical model, these modules are also represented by implicit functions, but this time with an extra argument for the third dimension. There are three ways of creating three dimensional shapes. Primitives and implicits are provided just like in two dimensions, then additionally a way of extending a two dimensional shape into three dimensions is provided.

Three Dimensional Primitives A collection of basic three dimensional shapes. Cube is centred around the origin with sides of length two. Cylinder has radius one and height one with base centered on the origin. Sphere represents a unit sphere centred at the origin. The cone has its point at the origin and grows bigger in both directions of the z axis. Torus is centered around the origin with major radius of one and minor radius of a half. For images of these shapes see Appendix B, noting that the cone has been exchanged for the Maya primitive cone as explained in 4.2.1.

$$
\begin{aligned}
[\![.]\!] &:: Prim3D \to f(x,y,z) \\
[\![Cube]\!] &= max(|x|,|y|,|z|) - 1 \\
[\![Cylinder]\!] &= (sqrt(x^2+y^2) - 1) + z - 1 \\
[\![Sphere]\!] &= x^2 + y^2 + z^2 - 1 \\
[\![Cone]\!] &= x^2 + y^2 - (z-1)^2 \ + z - 1 \\
[\![Torus]\!] &= (sqrt(x^2+y^2) - 1)^2 + z^2 - 0.5
\end{aligned}
$$

Three Dimensional Implicits Allows the user to include their own custom three dimensional shapes given as implicit functions.

$$
\begin{aligned}
[\![.]\!] &:: Implicit3D \to f(x,y,z) \\
[\![Implicit3D\ f]\!] &= f
\end{aligned}
$$

Three Dimensional Prisms Lifts two dimensional shapes into three dimensional by adding height of one above and below in the z axis.

$$
\begin{aligned}
[\![.]\!] &:: Prism3D \to f(x,y,z) \\
[\![Prism3D\ p]\!] &= ([\![p]\!]\ x\ y) + (|z| - 1)
\end{aligned}
$$

The semantics of p are used on the x and y values, then to ensure that the point also lies no more than one along the z axis in either direction, the absolute value of z with one taken away is added. Any value within this height range will render a value $\leqslant 0$ for the second part of the sum deeming it inside the shape if it also fulfils the requirements of the two dimensional implicit function.

3.3.3 Booleans

This module has the semantic meaning of a normal set. Lower levels of the language represented by implicit functions, can be made into sets by using a comparison to zero of the implicit function evaluated at a candidate point. These sets can then be combined using the set boolean operations provided by this module, allowing for more than one shape to be described by the language.

$$
\begin{aligned}
[\![.]\!] &:: Boolean \to Set \\
[\![Empty]\!] &= \emptyset \\
[\![Universal]\!] &= U \\
[\![Union]\!] &= \cap \\
[\![Intersection]\!] &= \cup \\
[\![Complement\ s]\!] &= \overline{s} \\
[\![Difference\ s_a\ s_b]\!] &= s_a \setminus s_b
\end{aligned}
$$

3.3.4 Transformations

Also living at the normal set mathematical model this module allows the user to perform affine transformations. It creates a new characteristic function for the set by reversing the transformation on the point then using the pre-existing characteristic function to see if it is part of the shape. Translations (t) take

the form of a 3x1 matrix, rotations (r) are a 3x3 rotation matrix, and scale (n) is just a scalar value. Visually this moves, rotates and scales the shape.

$$\begin{aligned} [\![.]\!] &:: Transformations \to Set \\ [\![Translate\ t\ s]\!] &= \lambda p \to [\![s]\!]\ (p - t) \\ [\![Rotate\ r\ s]\!] &= \lambda p \to [\![s]\!]\ (r^{-1}p) \\ [\![Scale\ n\ s]\!] &= \lambda p \to [\![s]\!]\ (np) \end{aligned}$$

3.3.5 Creating Soft Shapes

These two modules incorporate the idea of softness into the language using the outermost mathematical model of fuzzy sets. Softness is needed because hypertexture is applied to the soft region of an object. Two ways are offered to create soft shapes: softening functions that take a normal shape and soften it, and soft primitives.

Soften Lifts normal shapes to soft shapes by changing their characteristic function. Shapes can either be softened uniformly using *ConstSoften*, or based on how far they are away from a point c as a percentage of some maximum distance r using *FromPointSoften*.

$$\begin{aligned} [\![.]\!] &:: Soften \to Fuzzy\ Set \\ [\![ConstSoften\ n\ s]\!] &= \lambda p \to n * ([\![s]\!]\ p) \\ [\![FromPointSoften\ c\ r\ s]\!] &= \lambda p \to (1 - d/r) * ([\![s]\!]\ p)\ where\ d = euclideanDist\ p\ c \end{aligned}$$

ConstSoften works by first evaluating the shape s at the candidate point to get its density, then scaling this density by n. For *FromPointSoften* instead of the scale value being given, it is worked out based on how far away the candidate point is from a centre point. The value is the Euclidean distance between the candidate point and the centre point as a fraction of the maximum distance taken away from one.

Soft Primitives Contains the most common soft shape: the soft sphere, a unit sphere that gets less dense the further away the point is from its centre. Other soft shapes could be added to extend this module.

$$\begin{aligned} [\![.]\!] &:: PrimSoft \to Fuzzy\ Set \\ [\![SoftSphere]\!] &= \text{softSphere 1 (0,0,0) (defined in 2.2.2)} \end{aligned}$$

3.3.6 Modulate Functions

Each of the three types of Density Modulation Function has its own module. Densities can be modulated to affect their values or rates of change. Points can be modulated to add noise. Geometry can be modulated to add effects like hair. Combining these sets hypertexture apart and allows the creation of the phenomena.

Modulate Density Adjusts the membership value of a point in the set.

$$\begin{aligned} [\![.]\!] &:: ModulateDensity \to Fuzzy\ Set \\ [\![ModulateDensity\ f\ s]\!] &= f \circ [\![s]\!] \end{aligned}$$

Expressed as post-composition, s if first applied to the candidate point to get its density, which is then modulated by f. This could be used, for example, to make a cloud more dense, or to make a soft sphere change more rapidly from dense to soft.

Modulate Point Adjusts the point before presenting it to the membership function.

$$[\![.]\!] :: ModulatePoint \to Fuzzy\ Set$$
$$[\![ModulatePoint\ f\ s]\!] = [\![s]\!] \circ f$$

Expressed as pre-composition, the candidate point is first modulated by f and this point is then evaluated for a density by s. Essentially the density value for one point is given to another. This is key to adding noise.

Modulate Geometry Changes the membership function based on the existing members.

$$[\![.]\!] :: ModulateGeometry \to Fuzzy\ Set$$
$$[\![ModulateGeometry\ f\ s]\!] = f[\![s]\!]$$

This time f takes s as an argument and transforms it into another $point \to density$ function. This allows it to find out information about the geometry surrounding the candidate point, for example finding the nearest hard point to project a hair down onto. $ModulateGeometry$ is dangerous because of the sheer amount of freedom it gives the user.

3.4 Evaluation

Overall, the design of the language has been very successful, meeting the criteria for success 1.3. The design process of the language was fruitful, with the use of the denotational design method 2.2.3 proving to be a wise choice. The main difficulty was settling on the most appropriate mathematical model, but once this was decided the process made attaining the goals of the thesis easier, leading to an expressive yet controllable language.

3.4.1 Goals Met

The hypertexture language...

Clearly Specifies Behaviour The language was to clearly specify how a correct implementation of hypertexture should behave. Through combining the idea of Haskell type classes with denotational semantics (3.3), each function is not only described with name and type, but also with a mathematical specification of how it works.

When creating an implementation of the language it is unambiguous as to how each language key word should behave. For example, creating the two dimensional primitives is clearly creating a function that takes two arguments and returns a value.

Circle as C code:

```
double circle(double x, double y) {
  return (sqrt((pow(x,2))+(pow(y,2))) - 1)
}
```

Circle as Haskell code:

```
circle :: Double -> Double -> Double
circle x y = sqrt(x^2+y^2) - 1
```

Is Mathematically Based With three mathematical models, the language has clear grounding in the world of mathematics, making it easier to reason about. In particular, the use of fuzzy sets as a mathematical model allows the application of the fuzzy set properties to hypertexture.

Describes Hypertexture The language has the breadth to describe all phenomena demonstrated in the "Hypertexture" [13] paper. See 4.4 for code samples accompanied with matching generated images.

Makes Correct Implementations Easy to Create As will be shown in Chapter 4 the provision of denotational semantics makes it much easier to create implementations, and the modular nature of the language makes the language easy to customise to them. Additionally, the mathematical basis makes the set properties come for free. Trivial proofs can be constructed reducing the language to its set semantics, where we know these properties to hold, in one step.

For example, the commutative law for set union proof:

$$\begin{aligned} & Union\ a\ b \\ = & \{\text{def. } Union\} \\ & a \cup b \\ = & \{\text{Commutativity of set Union}\} \\ & b \cup a \\ = & \{\text{def. } Union\} \\ & Union\ b\ a \end{aligned}$$

Makes Hypertexture Easier to Control An issue with hypertexture is that it is unintuitive [5]. One possible contributor to this is that previously hypertexture is likely to have been produced using very bespoke code. Having a uniform language that interfaces with easy to create implementations should overcome this, enabling easier experimentation whilst creating different types of hypertexture. All that would need to be changed is the description in the language, not the whole code. This code is also very readable as demonstrated in 4.4.

3.4.2 Drawbacks

Powerful Some modules of the language are too powerful, namely the implicits and the modulate functions. They are "too powerful" because each can perform the functions of other parts of the language. For example, `Implicit3D` could perform the job of `softSphere` using the `softSphere` function defined in 2.2.2 (see below). Likewise `ModulateGeometry` could simulate `Empty`:

```
softSphere' = Implicit3D softSphere
empty'      = modulateGeometry (\f -> const 0)
```

This can lead to a violation of the goal to clearly specify behaviour through misuse of the language. It allows the user to cheat on properties, neglecting to uphold them by hiding the grievance in a different area of the language. For example, `Implicit3D` could theoretically be used to create any creatable hypertexture in one go, given a complicated enough function, and this complicated function could break properties of the language.

Luckily, due to the modular nature of the language, access to these powerful features can be controlled. Additionally, it would be simple to add extra modules that are similar to implicits, but more limited. For example, instead of the three dimensional implicits there could be an OBJ import. This would mean that only valid OBJ files could be imported, rather than any three-dimensional implicit function. This notion will be explored further in Chapter 6.

Chapter Summary

This chapter covered the main body of work produced in this thesis. The design process of the language was reported, showing why the language is as it stands. Then the language was summarised, with a detailing of its structure and the provision of its denotational semantics. Finally, the work discussed was evaluated. In the next chapter the language presented here will implemented as a Haskell eDSL.

Chapter 4

Implementation

This chapter will provide a tour of the example implementations produced as part of this thesis. All code apart from the C++ rasteriser and raytracer is written in Haskell and complemented with tests and full Haddock documentation.

Code produced includes a deep embedding and two example instances. The Maya instance illustrates how a stripped version of the language can be adapted to a specific domain such as the three dimensional modelling environment AutoDesk Maya. The Fuzzy Set instance implements the language as its mathematical meaning: a fuzzy set. This chapter will also detail how this instance is used to produce cross sections of hypertexture and provide density values to the C++ rendering programs, providing examples of the code in use.

4.1 Deep Embedding

The deep embedding has been created using the tools described in 2.1.6. Each component of the language takes the form of a syntactic functor, with the components being glued together using `:+:`:

```
type Hypertexture = Fix (Prim2D          :+: Implicit2D    :+: Prim3D :+:
                         Implicit3D      :+: Prism3D       :+: Transformation :+:
                         Boolean         :+: Soften        :+: PrimSoft :+:
                         ModulateDensity :+: ModulatePoint :+: ModulateGeometry)
```

As this is a huge coproduct, smart constructors have been provided so that the user does not have to type all the Ls and Rs. For example:

```
softSphere :: Hypertexture
softSphere = Fix (L(L(L(R(SoftSphere)))))
```

As discussed in 2.1.6 and shown in 2.1.7 example multiple semantics can be applied to the language through different Algebra instances. The next section will cover the two example instances included in this thesis.

4.2 Instances

4.2.1 Maya

This instance has been provided to demonstrate how one can take advantage of the modularity of the language to produce a smaller language customised to a specific domain. The domain targeted is the three dimensional modelling environment AutoDesk Maya. Examples of the shapes produced using this language can be found in Appendices A and B.

Components Maya is a three dimensional modelling environment with support for two dimensional and three dimensional shapes and moving or combining them. The goal of this instance is to successfully map the primitives of the language to Maya's primitive shapes and operations.

To match up well the following components have been combined into a new coproduct:

```
type MELLang = Fix (Prim2D    :+: Prim3D :+:
                    PrismMEL  :+: Transformation :+:
                    BooleanMEL)
```

Custom versions of three dimensional Prisms and Booleans have been created for this language in the form of `PrismMEL` and `BooleanMEL`. Prisms could have been created to have the same semantics as the original hypertexture language, where a two dimensional shape is extruded, but Maya has an inbuilt Prism primitive that seemed silly not to take advantage of, equally the Maya primitive Cone has also been used. The new Boolean component was required because Maya only offers a subset of Boolean operations.

Semantics The semantics applied to this language is a function that takes a string detailing the name of the defined shape and produces MEL code to create this shape. MEL is Maya's inbuilt scripting language.

Evaluation This baby language successfully achieves its purpose of mapping to MEL primitives. It also constitutes a more readable language than MEL, although this could be partly due to its limited functionality. This language could be improved in two ways: extending functionality - giving it access to use more of Maya's features; and instead of producing MEL code the language could link directly to Maya through the Maya API. Both of these avenues offer themselves for further exploration (especially as a typed Haskell scripting language would be far superior to the existing Python one) but are beyond the scope of this thesis.

4.2.2 Fuzzy Set

This instance applies the language's actual semantics. It includes a Haskell data type for describing fuzzy sets by characteristic function.

Semantics The fuzzy set semantics take the form of a data type containing the characteristic function: a function from element to its membership value. The membership value is stored as a Double and to avoid numbers outside the range [0,1] being stored the constructor of this data type is not exported. Instead three functions to create fuzzy sets are provided:

- `emptyFuzzy` - Creates an empty fuzzy set.
- `universalFuzzy` - Creates the fuzzy set that contains every element.
- `createFuzzy` - Given a function of type element to Double will create a fuzzy set where the membership values have been constricted to be within the allowed range.

Fuzzy set booleans have also been defined.

Uses The characteristic function has been used to create renderings of hypertexture (as will be discussed in 4.3), and to create cross sections using `CrossSection.hs`, which contains two functions:

- `drawHypertexture` - Takes a fuzzy set and produces an `ImageArray` for `writePGM`. The user can decide the resolution and specify which z value they want as the cross section. It operates by evaluating the characteristic function at all points within that two dimensional slice of the hypertexture.
- `writePGM` - Takes an `ImageArray` produced by `drawHypertexture` and outputs it as a .pgm image file. The user can decide the resolution of the image where white represents fully dense, and black is fully transparent.

An exciting use of the cross section code is to create a physical glass engraving of a hypertexture! By layering cross sections of a hypertexture engraved on glass a three dimensional effect can be created (See Figure 4.1).

Fireball

Dripping Sphere

Figure 4.1: Physical Hypertexture Glass Engraving

Evaluation This instance successfully implements the language as a fuzzy set and further demonstrates how using the denotational design method means that the laws come for free. The implementation is accompanied by property tests (see `BooleanSpec.hs`) that verify this.

The laws can also be shown to hold by writing simple proofs on the Haskell functions. For example, the commutative law for set union proof:

$$
\begin{aligned}
& alg\ (Union\ a\ b) \\
&= \{\text{def. } alg\} \\
& unionFuzzy\ a\ b \\
&= \{\text{def. } unionFuzzy\} \\
& Fuzzy\ (\lambda x \rightarrow max\ (characteristicFunction\ a\ x)\ (characteristicFunction\ b\ x)) \\
&= \{\text{Commutativity of } max\} \\
& Fuzzy\ (\lambda x \rightarrow max\ (characteristicFunction\ b\ x)\ (characteristicFunction\ a\ x)) \\
&= \{\text{def. } unionFuzzy\} \\
& unionFuzzy\ b\ a \\
&= \{\text{def. } alg\} \\
& alg\ (Union\ b\ a)
\end{aligned}
$$

4.3 Rendering

Renderers capable of doing volumetric rendering can interface with the characteristic function of the fuzzy set instance as a look up for density values.

4.3.1 Image Acquisition

To obtain three dimensional renderings of hypertexture for this thesis the raytracer and rasteriser written for another course were used. These are both written in C++. The raytracer uses the ray marching

algorithm described in the "Hypertexture" [13] paper, and the rasteriser uses a novel equivalent method.

Both of these renders interface with a Haskell look up function that takes a hypertexture description written in the hypertexture language, reduces it to the fuzzy set semantics, and uses the fuzzy set to look up the density values needed for rendering. Initially, Haskell code was created that produced a C++ look up table to be embedded into the renderer, however this was infeasible due to the number of values that were needed. See 4.4 for images produced by the raytracer matched with the hypertexture code that described them.

To amplify the effect of the fire hypertexture a function that decides the colour of a point based on its density was used:

```
vec3 densityBasedColour(double density){
  float r = std::fmin(1.0, 0.9+(float)density);
  float g = 0.96;
  if (density < 0.1) g = std::fmin(0.96, (float)density);
  float b = 0;
  vec3 colour = vec3(r,g,b);
  return colour;
}
```

4.3.2 Raytracing

To render hypertexture with a raytracer the ray marching algorithm can be used as suggested in the "Hypertexture" paper [13]. This algorithm works by putting a bounding box around the hypertexture. When a ray hits the box, rather than just sampling the colour of the object like normal raytracing, the following is done:

1. The intersection point with the front of the box is noted.
2. The ray is allowed to continue its journey till it exits the bounding box, a point that is also noted. If this point is never hit, the camera is in the box, so the points used are the camera position and the first intersection.
3. Densities are then sampled at evenly spaced points between the entry and exit points. This can be achieved through interpolating along the ray. (See Figure 4.2)
4. The final colour of the pixel can be found by compositing the sampled densities in the form of RGBA (red green blue alpha) colours, where density forms the alpha channel (its opacity).

The following C++ code maps out this process:

```
void rayMarch(vector<vec4>& layers, vec4 frontPoint, vec4 backPoint) {
    vector<vec4> points;
    Interpolate(frontPoint, backPoint, points);
    lookUpColour(points, layers);
}
```

The `rayMarch` function is given the entry and exit points, `frontPoint` and `backPoint` respectively, and a vector of colours to fill. It then uses `Interpolate` to get a list of evenly spaced points between `frontPoint` and `backPoint`. `lookUpColour` then populates `layers` with RGBA values to be composited. These layers can then be given to a compositing function:

```
vec3 Composite(vector<vec4> layers) {
  vec3 colour(0,0,0);
  float totalTrans = 0;
  for (int k = 0; k<layers.size(); k++) {
    vec3 currentColour(layers[k].x, layers[k].y, layers[k].z);
    float transparency = layers[k].w * (1-totalTrans);
    colour = colour + transparency * currentColour;
    totalTrans = totalTrans + transparency;
  }

  return colour;
}
```

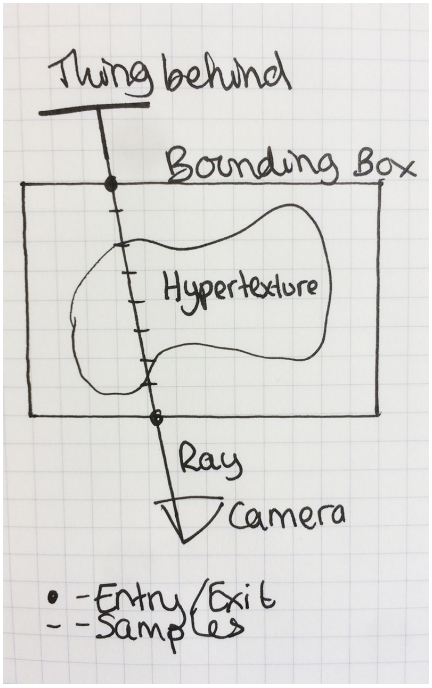


Figure 4.2: Ray Marching Sampling Process

This function takes a list of RGBA values and composites them down into one RGB value (no alpha channel because the composite will be fully opaque). It begins with black and adds each colour in the list proportional to its alpha channel to achieve the final colour. It is useful to add the colour of the thing behind the bounding box with full opacity to the list, so that a transparent hypertexture can be visualised as transparent.

4.3.3 Rasterisation

To rasterise a hypertexture the goal is also to get points that densities can be looked up for. However, instead of using a technique based on rays, the technique to get the points must use the bread and butter of rasterisation: interpolation.

Since no pre-existing method of doing this could be found, a custom algorithm to do the job of finding points to look up density for was concocted. It works as follows:

1. The eight corners of the bounding box are identified.

2. Corresponding corners (e.g. front bottom left and back bottom left) are then interpolated between to get two dimensional "slices" of the hypertexture.

3. The three dimensional locations of points within these slices can then be used to look up the density.

4. The final colour is found by compositing the slices.

Compositing in the rasteriser is not as easy as with the raytracer. Each slice must be drawn on layer by layer from the front. To remember the previous layer a buffer containing the colour so far is needed. As an optimisation the total transparency can also be accumulated so that computation can be stopped when the colour is saturated.

4.3.4 Evaluation

These renderers are very successful and achieve the secret goal of this thesis to produce pretty pictures. However, the process for rendering hypertextures could be improved in two ways: adapting the language to the underlying renderer, and writing the hypertexture language in the same language as the renderer.

Adapting If the underlying render had primitive methods for rendering soft primitive shapes, such as soft spheres, then more soft primitive shapes could be added to the language. If the underlying render had primitive methods for softening shapes then the soften functions would be a much more attractive option. The nature of the hypertexture language is such that it can be adapted to the underlying domain, instead of the underlying domain having to be specialised to the hypertexture language.

Embedding Haskell may have proven to be a suitable host for the hypertexture language, but if desired the language specification could be used to create a C++ implementation of the language allowing it to match perfectly with the renderer. As explored, the creation of this implementation would be well guided due to the provision of the denotational semantics, and it would be clear what a correct implementation should consist of.

For example, the *softSphere* could be implemented in C++ as follows:

```
vec4 softSphere(vec4 p) {
  vec4 center(0.5,0.5,0.5,1);
  float r = 0.3;
  float density;
  float dist = euclideanDist(p, center);
  if (dist < r) density = (1- dist/r);
  else density = 0;

  return vec4(1,0,0,density);
}
```

This function takes a point and works out the density value just like the Haskell function from 2.2.2, but since it has been implemented in the domain of the render it can be optimised to perform best with regards to that domain. The point can be represented as a vector with four components to allow for the optimisation of using homogeneous coordinates. Additionally, the function can skip straight to returning the RGBA colour of the point, which is what is actually required for rendering. This particular function allows the rasteriser to run in real time, demonstrating the speed benefits of implementing the language within the domain.

4.4 Language In Use

This section demonstrates the language in use. It shows code written in the hypertexture language to describe phenomena mentioned in the "Hypertexture" [13] paper. Each code snippet matches an image produced by that code and the raytracer.

All phenomena will use some form of noise that at its core uses the Perlin Noise described in 2.2.1. For full details of these noise functions see `Noise.hs`. The noise functions are controlled by a data type called `NoiseSettings`, allowing the user to tweek the noise to suit them:

```
defaultNoiseSettings
  = NoiseSettings { gradients = perm
                  , steepness = 1
                  , squiggliness = 2
                  , inverted  = True
                  }
```

Above is a default noise with settings that render a good result. The different aspects that can be controlled are: gradients - the set of random gradients, steepness - how quickly the noise changes from 0.0 to 1.0, squiggliness - how much the noise twists and turns, and inverted - whether 0.0 or 1.0 is the dominant value. The default noise uses `perm` the list of gradients introduced in [12] and relatively low values of steepness and squiggliness.

Since the noise functions only work on values with the range [0,1], they are called with ranges of x, y and z values, allowing each point can be scaled down appropriately.

4.4.1 Fractal Sphere

The fractal sphere (Figure 4.3) is created using the following code:

```
fractalSphere
  = modulatePoint
      (fractalNoise
        defaultNoiseSettings
        -- Ranges for scaling
        (minx,maxx) -- x Range. (min,max).
        (miny,maxy) -- y Range. (min,max).
        (minz,maxz) -- z Range. (min,max).
        iterations
        frequency)
      (scale s $ softSphere)
```

The hypertexture language takes a `softSphere` of scale `s` and modulates incoming points with `fractalNoise`. To get the density of a candidate point: the point has the fractal noise added to its x, y and z values, and the density value of this new point in the sphere used. As pointed out previously, `fractalNoise` is provided with ranges so that the points can be scaled down for the noise to be looked up.

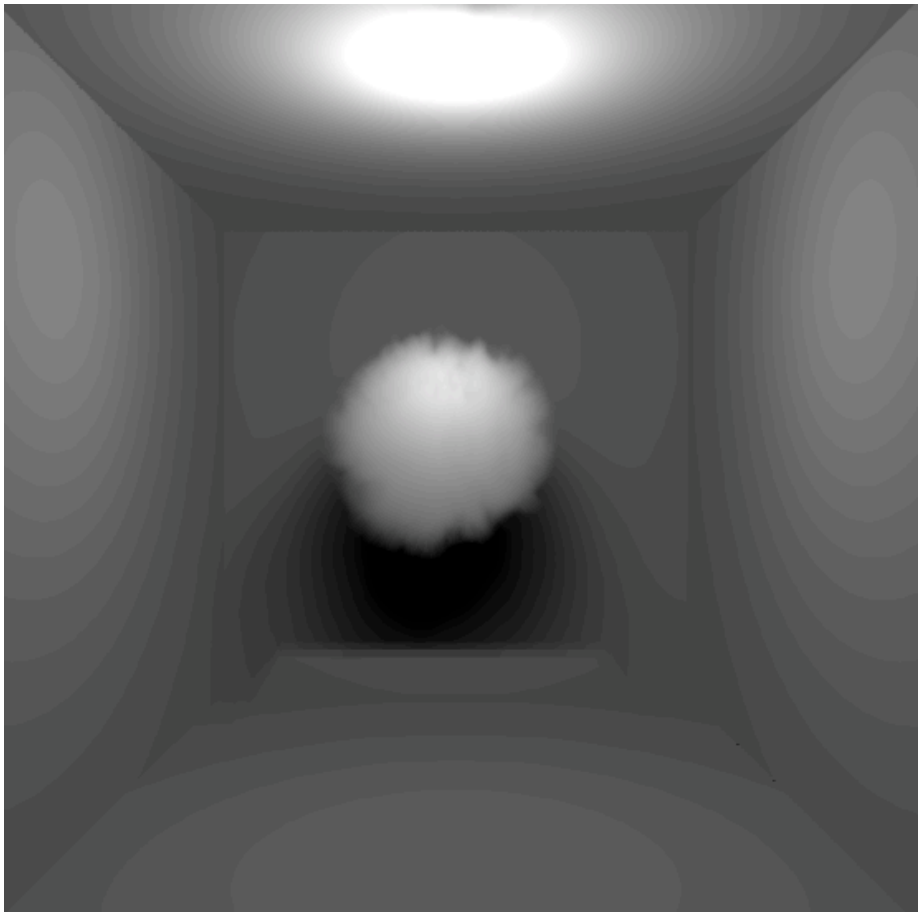


Figure 4.3: Fractal Sphere

4.4.2 Eroded Cube

The eroded cube (Figure 4.4) demonstrates how combining shapes with the boolean operations can create interesting new shapes. It is achieved by taking the intersection of a cube and the fractal sphere, giving the effect that the cube has eroded away at the edges. The code below implements this, with the cube and sphere being slightly different sizes to improve the appearance:

```
erodedCube
  = intersection
      (scale s $ cube)
      (scale (s*1.5) $
        fractalSphere
          defaultNoiseSettings
          -- Ranges for scaling
          (minx,maxx) -- x Range. (min,max).
          (miny,maxy) -- y Range. (min,max).
          (minz,maxz) -- z Range. (min,max).
          iterations
          frequency
          s)
```

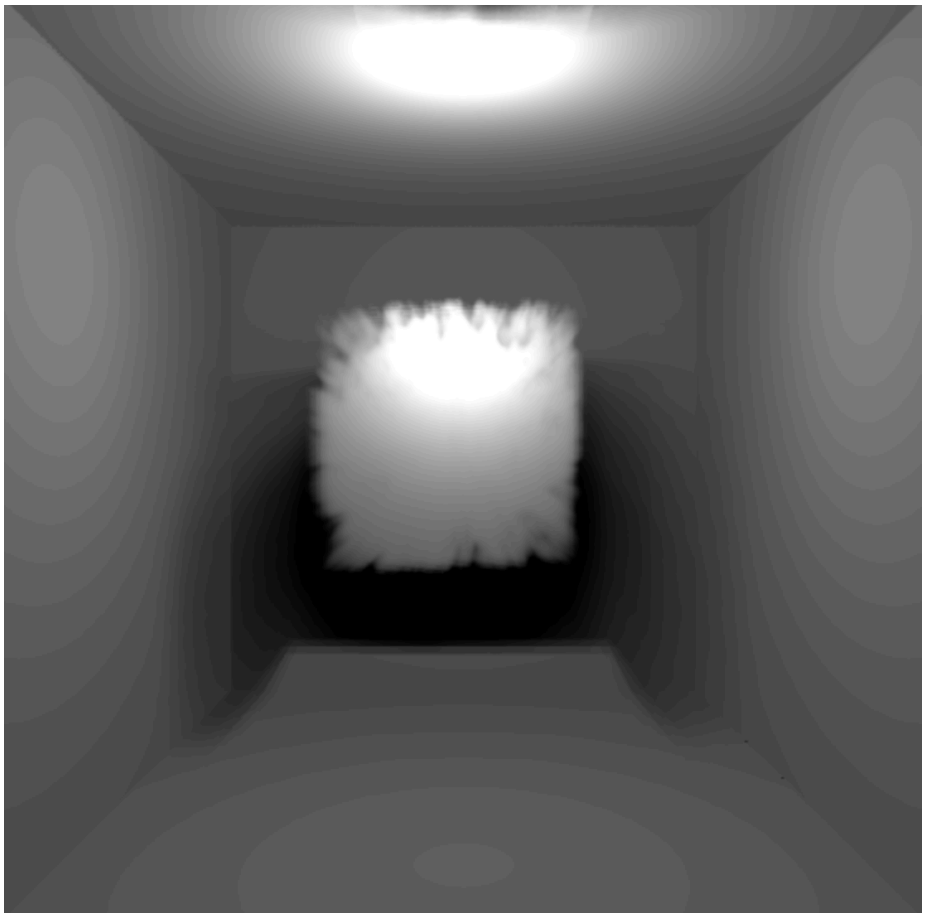


Figure 4.4: Eroded Cube

4.4.3 Noisy Sphere

The noisy sphere (Figure 4.5) was created by modulating each candidate point by standard Perlin Noise before asking the characteristic function of the `softSphere` scaled by `s` for the density value:

```
noisySphere
  = modulatePoint
    (addNoise
      defaultNoiseSettings
      -- Ranges for scaling
      (minx,maxx) -- x Range. (min,max).
      (miny,maxy) -- y Range. (min,max).
      (minz,maxz) -- z Range. (min,max).
      frequency
      amplitude)
    (scale s $ softSphere)
```

The user also has control of the frequency and amplitude of the noise. The frequency value scales up the point before the noise value is calculated, and the amplitude value scales the noise up once it has been calculated.

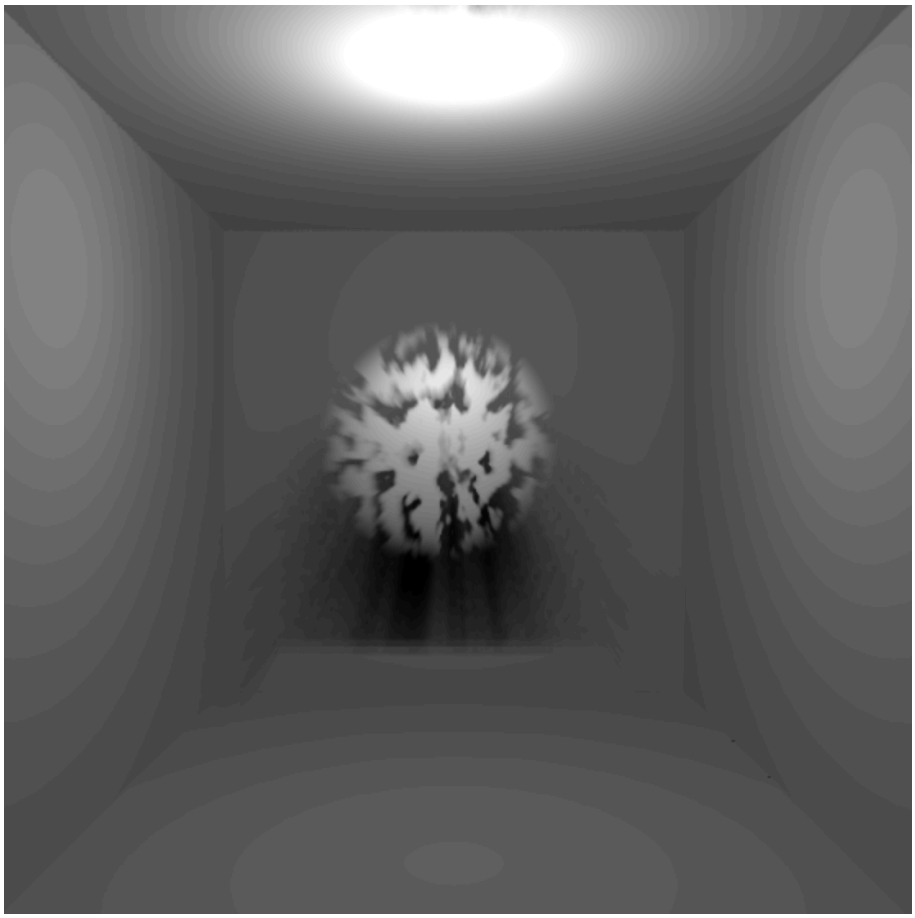


Figure 4.5: Noisy Sphere

4.4.4 Dripping Sphere

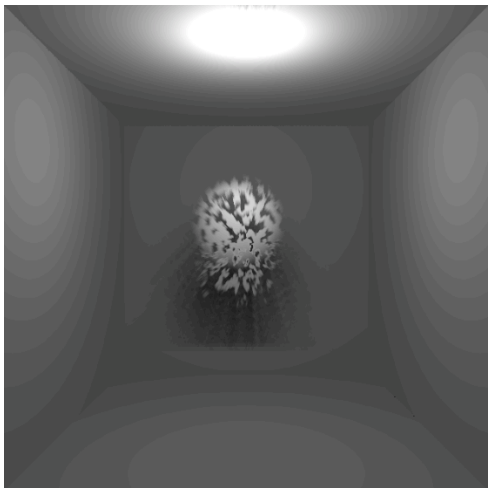
The drip effect is created by only adding noise to one component of the point. If noise is added to the x value then the object will drip towards the x axis. How much it drips can be controlled by how far away the object is from the origin. To create the normal coloured dripping sphere (Figure 4.6) the following code is used:

```
drippingSphereNormal
  = translate (s*d,0,0) $
      modulatePoint
        (dripX
          defaultNoiseSettings
          -- Ranges for scaling
          (minx,maxx) -- x Range. (min,max).
          (miny,maxy) -- y Range. (min,max).
          (minz,maxz) -- z Range. (min,max).
          frequency
          amplitude)
        (scale s (translate (-d,0,0) softSphere))
```

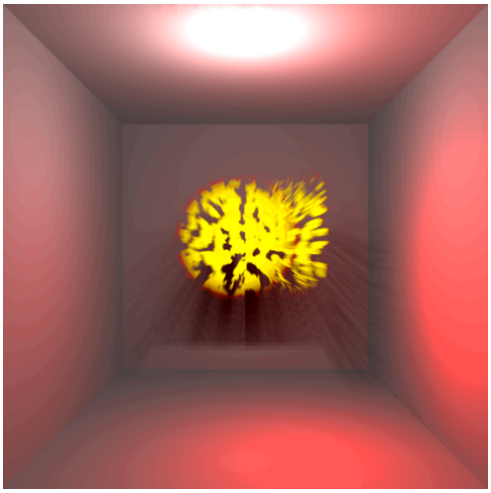
The `softSphere` is moved above the x axis to scale the drip by `d`, then it is scaled by `s`, next each point's x value is modulated by noise, before the now dripping sphere is returned to be centred on the origin.

```
drippingSphereFireColours
  = translate (0,s*d,0) $
      modulatePoint
        (dripY
          defaultNoiseSettings
          -- Ranges for scaling
          (minx,maxx) -- x Range. (min,max).
          (miny,maxy) -- y Range. (min,max).
          (minz,maxz) -- z Range. (min,max).
          frequency
          amplitude)
        (scale s (translate (0,-d,0) softSphere))
```

If the axis is switched to Y it almost looks like a flying fireball, an appearance that is amplified by making it glow and using the fireball colours. One could even apply this drip to a fireball instead of a `softSphere` to get something really exciting.



Normal



Fire Colours

Figure 4.6: Dripping Spheres

4.4.5 Fireball

To create the fireball (Figure 4.7), the default noise settings just will not do. To make the best fireball the `squiggliness` of the `NoiseSettings` needs to be much higher:

```
fireNoiseSettings
  = NoiseSettings { gradients = perm
                  , steepness = 1
                  , squiggliness = 8
                  , inverted  = True
                  }

fireball
  = modulatePoint
      (addTurbulence
        fireNoiseSettings
        -- Ranges for scaling
        (minx,maxx) -- x Range. (min,max).
        (miny,maxy) -- y Range. (min,max).
        (minz,maxz) -- z Range. (min,max).
        iterations)
      (scale s $ softSphere)
```

The fireball can then be created by modulating the incoming points with repeated noise, called turbulence.

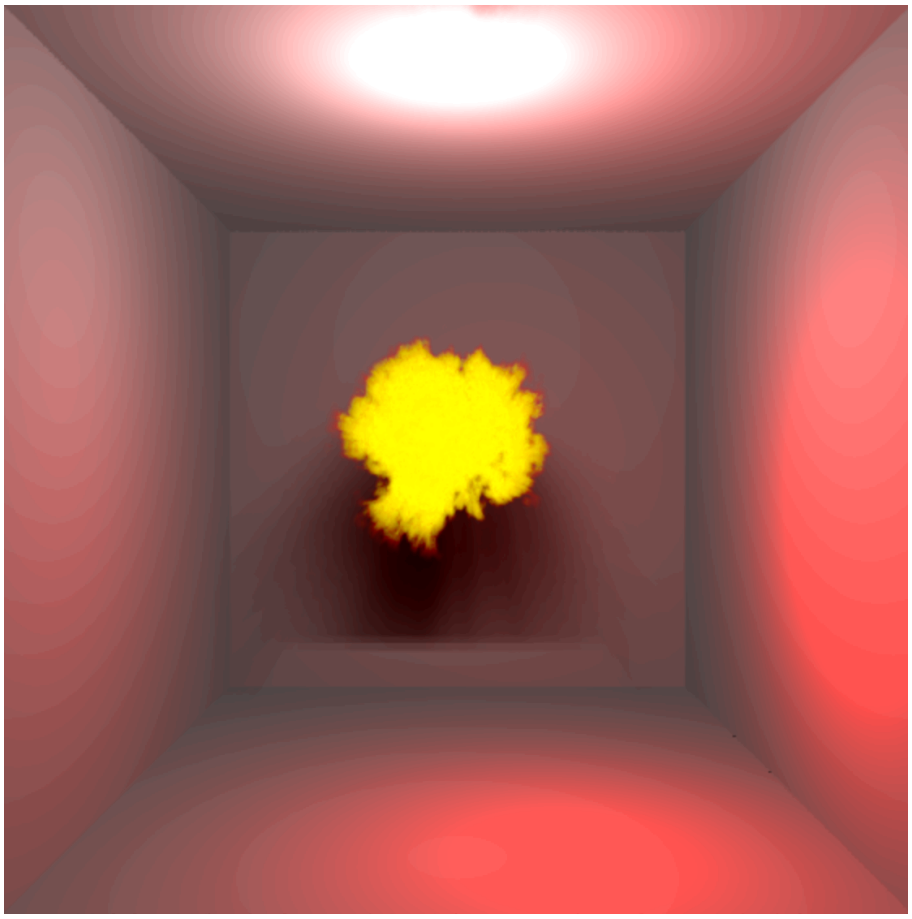


Figure 4.7: Fireball

4.4.6 Furry Donut

Unfortunately, as will be explored in 5.3, the implementation is slow. This means that an image for hair could not be created. However, the code is still worth a look:

```
furryDonut
  = modulateGeometry
      (hair
        hairSettings -- with increased squiggliness
        -- Ranges for scaling
        (minx,maxx) -- x Range. (min,max).
        (miny,maxy) -- y Range. (min,max).
        (minz,maxz) -- z Range. (min,max).
        points -- points to consider for projection
        frequency)
      (scale s $ softHalo torus)
```

To create the `furryDonut`, `modulateGeometry` is needed. For each point, `hair` works by using the provided ODF to find the nearest hard point (with a density of one) from `points`, the noise value of this point is then calculated and used. The function `softHalo` adds a "halo" of soft region around a shape so hair can be added in that halo.

Chapter Summary

This chapter explored the implementation this thesis. It gave a deep embedding, providing a proof of concept and evidence that the design of the language was successful in assisting an implementer. Two example semantics were also applied to the language highlighting the customisability and reliability of the language. It can adjust well to different domains, and the implementer is well guided towards a correct solution. Additionally, the method of rendering hypertexture for this thesis was explained, and the language demonstrated. What has been learned from these implementations will be summarised in Chapter 5 and shaped into achievable improvements in Chapter 6.

Chapter 5

Evaluation

This section summarises the evaluations provided throughout to assess the success of the thesis holistically with respect to the success criteria detail in 1.3. It will recap the goals of the thesis and then detail how features of the thesis meet these goals. Areas for improvement in the thesis will then be introduced to be further explored in Chapter 6.

5.1 Goals

The aim of this thesis as to shift the focus of hypertexture away from the result towards the implementation by creating a hypertexture language that:

- Clearly specifies behaviour.
- Is mathematically based.
- Describes hypertexture.
- Makes correct implementations easy to create.
- Makes hypertexture easier to control.

5.2 Meeting Criteria

The implementation is the feature. This whole thesis is about implementations for hypertexture. It explores in great detail through Chapter 3 how best they should be created and presented, culminating in the hypertexture language. Then in Chapter 4 different possible implementation domains are explored from Maya 4.2.1 to Fuzzy sets 4.2.2 . The fuzzy set instance does not even directly produce images - the results of this instance are the meaning of the language, because that is the focus. Then when the images produced by the raytracer are shown in 4.4 the emphasis is not on them, but the language that produced them. How they were achieved. The implementation.

Behaviour is clearly specified. In Chapter 3 a lot of effort is put into explaining the language and how it should behave. Its structure and mathematical models are detailed in 3.2. Using the language of mathematics makes the meaning of each section clear because mathematics is well studied and explored. The semantics are given as denotational semantics in 3.3 accompanied by English explanations and images (A and B) to be as clear as possible what each key word means. Chapter 4 demonstrates how the clear specification aids implementation. For example, the fuzzy set instance (4.2.2) was extremely easy to create, especially in Haskell, since it merely involved typing out the denotational semantics with Haskell syntax.

The language is mathematically based. The language has three mathematical models described in 3.2. Using multiple semantics has allowed the language to better meet the previous goal and specify behaviour through a semantics that is best suited. The benefits of having this mathematical basis is clear when properties of the language were shown mathematically in the Language chapter (3.4.1) and numerically in the Implementation chapter (4.2.2) through tests.

The language describes hypertexture. The hypertexture language produced certainly describes all phenomena of hypertexture from the "Hypertexture" [13] paper as demonstrated in 4.4 where images produced by the raytracer (4.3) are showcased. It also allows for attributes of the phenomena to be easily edited. For example, they can be made bigger using `scale`, and the `NoiseSettings` data type really helps change the desired elements of choice.

Implementations are easy to create. The quality of the hypertexture language that it clearly specifies behaviour greatly simplifies the creation of implementations. Clearly the denotational design method (detailed in 2.2.3) was an excellent choice. Equally, using the language design tools presented in 2.1.6 also aids the creation of an implementation as demonstrated in 2.1.7. Applying multiple semantics or implementations can be as easy as writing a new type class instance. Additionally, the ability to test properties and prove laws on implementations as seen in 4.2.2 further assists the creation of a correct implementation.

The modularity and choice of a DSL brings simplification and clarity making the hypertexture language highly customisable. Not only are implementations easy to create, but they are also more specialised and efficient, truly encapsulating the idea of a DSL by avoiding unnecessary clutter.

Moreover, the language is highly adaptable to specific domains. It can be stripped back to map to AutoDesk Maya as shown in 4.2.1, or adapt to renderers with different specialised features as discussed in 4.3. The implementations provided exhibit this feature.

Hypertexture is easier to control. By creating a language, this thesis has alleviated the problem that hypertexture is unintuitive and difficult to reason about, because now there exists a language within which it can be reasoned about. Having a mathematical basis (3.2) also allows for the language to be reasoned as shown in 3.4.1. The quality of the language that it is easy to embed in a renderer (discussed in 4.3.4) also opens the door for a real time render. This would provide instant feedback to the user of the language allowing them to gain intuition for how to control the language. Additionally, the `NoiseSettings` introduced in 4.4 helps customise the different phenomena, as illustrated by the way fire is made more intense.

5.3 Areas for Improvement

Overall this thesis has been very successful, meeting all of the goals. However, there are two main areas for improvement with this thesis: the implementations provided are very slow and areas of the language are too powerful. The implementation of the deep embedding could also be minorly improved using subtyping [15].

Slow Since the implementations were only to form a proof of concept, none of the code is optimised. Noise calculations for hypertexture phenomena such as fire are time consuming, and hypertexture like hair takes so long that images from the raytracer could not be produced. Speeding these up could open the way for a real time renderer, really improving users' ability to play with hypertexture and gain intuition on how it works.

Powerful As detailed in 3.4.2 the implicits and the modulate functions are too powerful. One function should not be able to achieve everything that can be done in the language.

Subtyping Currently to avoid the inconvenience of the Ls and Rs from the coproduct functor, smart constructors have been used. This works but is not very maintainable: a single change to the coproduct means that all smart constructors need to be changed. They are also not readable to read or write making mistakes very likely. A much more elegant way to achieve this would be to use subtyping as demonstrated in "Data types à la carte" [15].

Chapter Summary

This chapter summarised the goals of the thesis and gave evidence as to how they were met. Areas for improvement were then introduced, leading onto Chapter 6 where ways of tackling each of these problems is presented.

Chapter 6

Future Work

This section will suggest how the three areas for improvement introduced in 5.3 could be tackled. There are also a couple of other interesting avenues that could be explored. The adding of hypertexture to non-closed objects could be added using "Hypertexturing complex volume objects" [14], or as will be further explored in 6.4 the hypertexture language could be integrated with ReIncarnate [9].

6.1 Optimisation

When it comes to optimising the code produced, either the Haskell code can be optimised, or optimisation could be achieved through writing the language in the same language that the target domain is written in. For example, if the goal is to produce renders of the hypertexture and a C++ rasteriser was to be used, the language could be implemented in C++ to complement the rasteriser.

The optimisation process itself could also prove easier to verify with the clear specification of a correct implementation. It would be obvious what the required properties of the optimised code should be, and these can be tested against to ensure that changes to make the code faster do not break it.

6.2 OBJ input

Problem One of the modules of the code that is too powerful is the three dimensional Implicits that takes any function of x, y, and z to describe a hypertexture. Theoretically the user could use a really complicated function in here and do the job of the rest of the hypertexture language. For example, there is nothing stopping them using this function to soften something instead of using the Soften operators.

Solution The reason for this operator was to avoid limiting the user to the primitive shapes provided. An alternative to this would be providing the input in a more restricted form: OBJ files. OBJ files are an ASCII file format for describing shapes as a shell of triangles. If this was provided as part of the language coproduct, the user could import any valid OBJ file. This is a compelling prospect as there are many online OBJ stores providing a wide range of premade shapes (e.g. free3d.com).

How To achieve this import a parser for OBJ files would be needed that can extract the vertices and faces of triangles described in the file. Then a list of triangles that form the shape can be created using a function like the one below:

```
-- | Function to take an obj file and get triangles.
getTriangles
 :: String -- ^ .obj file
 -> [Triangle]
getTriangles s =
  let
    vertices = fromJust $ parseMaybe vertexParser s
    faces    = fromJust $ parseMaybe faceParser s
    makeTriangle :: [Point] -> (Int,Int,Int) -> Triangle
    makeTriangle vs (v0,v1,v2) = (vs!!(v0-1),vs!!(v1-1),vs!!(v2-1))
  in map (makeTriangle (vertices)) faces
```

Then this list of triangles needs to be converted from the shell representation of a three dimensional shape to one suitable for hypertexture, for example a point to density function:

```
-- | Function that works out if a point is in a triangle shell and
--   returns a density of 1 if it is, 0 if not. Works by counting the
--   amount of triangles point intersects with, if it is odd then the
--   point is inside.
--
-- Clear intersection:
-- >>> fromTriangles [((1,2,0),(1,-2,-1),(1,-2,1))] (0,0,0)
-- 1.0
--
-- Tilted intersection:
-- >>> fromTriangles [((0,2,0),(1,-2,-1),(1,-2,1))] (0,0,0)
-- 1.0
--
fromTriangles
  :: [Triangle] -- ^ Shell.
  -> Point      -- ^ Point.
  -> Double
fromTriangles ts p
  | odd $ length $ filter (intersect p) ts = 1
  | otherwise                              = 0
```

6.3 Subtyping

Subtyping is a method of making smart constructors more general [15]. Instead of manually writing all the Ins and Ls and Rs, as has been done in this thesis, the injections can be inferred. Using this method would greatly improve the maintainability of the deep embedding.

6.4 ReIncarnate Integration

The ReIncarnate [9] software provides the potential for another way of importing three dimensional shapes from another common file format, except this time the import can come in the form of the hypertexture language. ReIncarnate can take a .stl file and produce .cad3 code, ReIncarnate's internal language for describing three dimensional shapes in terms of its primitives and any booleans or transformations involved. If a way to translate between .cad3 and the hypertexture language was produced, then hypertexture code could be produced from .stl files, making it even easier to add hypertexture to different shapes. The hypertexture language has been designed with this potential translation in mind, including constructors that exactly match that of .cad3.

Summary

This chapter detailed plans to further improve the work of this thesis. It explored the idea of optimising the implementations, and proposed another module of the language that provides another way of importing shapes into the language, and gave the alternative of subtyping. Finally, it introduced an exciting compatibility potential with ReIncarnate [9]. The chapter highlights how, despite the thesis being very successful, there are still areas to explore and improve upon.

Chapter 7

Summary

The area within which hypertexture lies is an exciting one, including fireballs and Academy Awards. Being part of the computer graphics domain gives hypertexture relatability and tangibility. Combining this with the field of programming languages allowed the presentation of an interesting new slant on not just hypertexture but also the way computer graphics is viewed.

This section gives a brief summary of what was explored in this thesis.

Introduction Hypertexture is an alternative shape representation, describing a three dimensional shape as points in three dimensional space with density values rather than a three dimensional shell. It has relevance in the real world through its use in CGI. The goal of this thesis was to produce a domain specific language for describing hypertexture.

Background This thesis builds directly on the work of Ken Perlin who introduced the notion of hypertexture as a way of describing shapes [13]. The language for describing hypertexture was decided to be a domain specific language (DSL). A DSL is a language with a specific purpose, only containing necessary features. This has the advantages of improving efficiency and specificity of the language created. The language was created in Haskell because Haskell is well suited to hosting a DSL.

The design process of creating the hypertexture language followed a method from "Denotational design with type class morphisms " [4]. It involves selecting a mathematical model for what is to be described and using that mathematical model to guide semantic choices. The end result is a combination of Haskell type classes and denotational semantics: functions, their names, their types, and a mathematical description of how they should behave.

The mathematical models chosen for the hypertexture language are: implicit functions, sets, and fuzzy sets. Implicit functions are typically used to describe shapes as a function of x, y, and z. If an implicit function values to zero when given a point, that indicates that the point lies on the edge of the shape. Sets are a collection of items defined by binary membership. An object is either an element or it is not. Fuzzy sets are similar, but their membership is a continuous value between 0 and 1 [16][6]. An object can be half a member of a fuzzy set.

Language The most difficult part of designing the language was deciding what the mathematical model should be. The language went through many iterations: starting with the two potential models of point to density functions and fuzzy sets, experimenting with the point to density model, before settling on the fuzzy set model and making the language more modular.

The final language has three semantic models. Fuzzy sets form the outer most layer with normal sets and implicit functions describing subsets of the language. Implicit functions are used for the basic shapes. Normal sets are used for combining shapes together. Fuzzy sets are used to add the hypertexture, where the membership of a point represents its density.

Overall the language is successful in meeting the goals of the thesis because it:

- Clearly specifies behaviour.
- Is mathematically based.
- Describes hypertexture.
- Makes correct implementations easy to create.

- Makes hypertexture easier to control.

The main drawback of the language is that some components of the language give the user too much power.

Implementation Not only does this thesis provide a Haskell deep embedding of the proposed language, but also two example implementations. These implementations demonstrate different strengths of the language. One implementation uses the language as an interface to AutoDesk Maya and illustrates how the modular nature of the language makes it easy to adapt to different domains. The fuzzy set instance implements the language as its semantic meaning, demonstrating how the design method gave the language fuzzy set properties for free.

The fuzzy set implementation is also used by a custom made raytracer and rasteriser to create renderings of the hypertexture. All hypertextures mentioned in the "Hypertexture" [13] paper can be described using the language as can be seen in 4.4. Code was also made to produce cross sections of hypertexture.

Evaluation Overall this thesis has been a success, since the language produced achieves all the goals. The main areas that could be improved on is the speed of the example implementations, since they run slowly, and the over powered nature of some of the components.

Future Work In future this work could be improved upon. For example, instead of having implicit three dimensional functions as a way of getting custom shapes into the language an OBJ import could be used. This still provides the user with more freedom than just the primitive shapes, but also limits them to valid OBJ files. Finally, the work could also be improved upon by combining it with ReIncarnate [9].

In conclusion, this thesis successfully presents a uniform interface for hypertexture implementations in the form of a domain specific language accompanied with denotational semantics.

Bibliography

[1] Oscar winners data base entry. Accessed: 2019-02-04.

[2] Understanding perlin noise. Accessed: 2019-05-09.

[3] Session details: Course 6: Anyone can cook: Inside ratatouille's kitchen. In Apurva Shah, editor, *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages –, New York, NY, USA, 2007. ACM.

[4] C. Elliott. Denotational design with type class morphisms (extended version). 2016.

[5] G. Gilet and J.M. Dischler. A framework for interactive hypertexture modelling. *Computer Graphics Forum*, 28(8):2229–2243, 2009.

[6] S. Gottwald. An early approach toward graded identity and graded membership in set theory. *Fuzzy Sets and Systems*, 161(18):2369 – 2379, 2010.

[7] John C. Hart. Implicit representations of rough surfaces. *Computer Graphics Forum*, 16(2):91–99, 1997.

[8] A. Lagae, S. Lefebvre, J.P.Lewis, K.Perlin, M.Zwicker, and et al. State of the art in procedural noise functions, 2010.

[9] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.*, 2(ICFP):99:1–99:31, July 2018.

[10] K. Perlin. Improved noise reference implementation.

[11] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[12] K. Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002.

[13] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, July 1989.

[14] Richard Satherley and Mark W. Jones. Hypertexturing complex volume objects. *The Visual Computer*, 18(4):226–235, Jun 2002.

[15] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

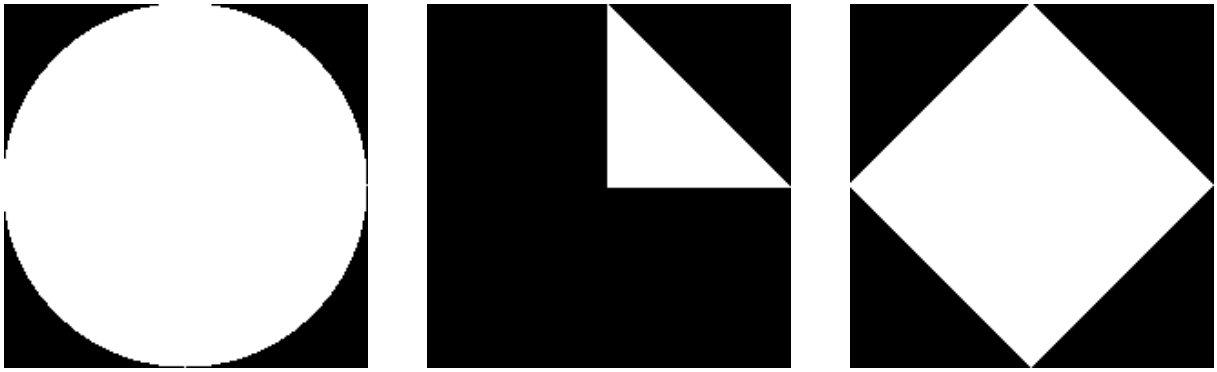
[16] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965.

Appendix A

Images for Two Dimensional Primitives

A.1 2D Primitive Cross Sections

Created using 4.2.2.

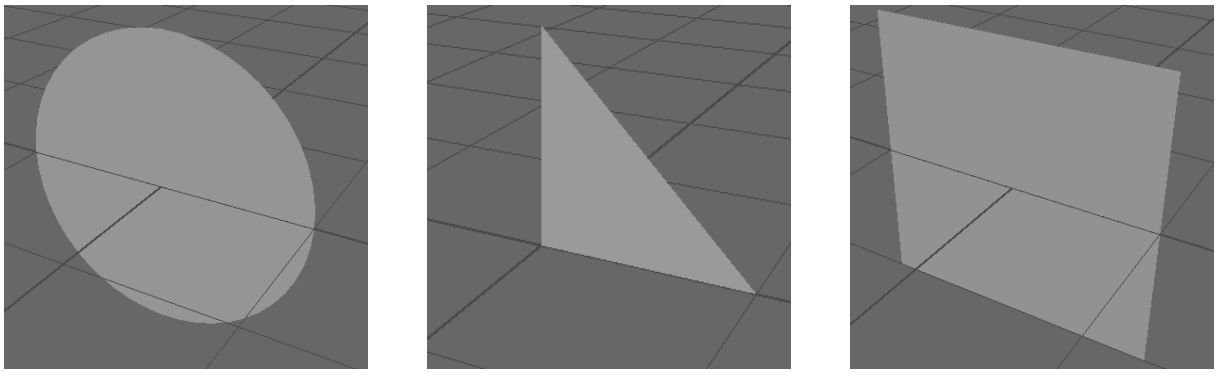


Circle Triangle Square

Figure A.1: 2D Primitive Cross Sections.

A.2 2D Primitives in 3D Space

Created using MELLang from 4.2.1.



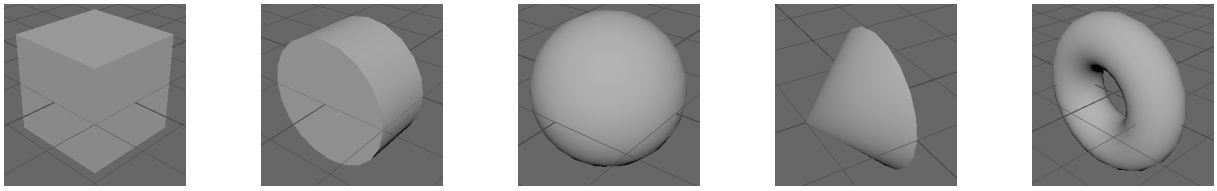
Circle Triangle Square

Figure A.2: 2D Primitives in 3D Space.

Appendix B

Images for Three Dimensional Primitives

Created using MELLang from 4.2.1.



Cube Cylinder Sphere Cone Torus

Figure B.1: 3D Primitives.