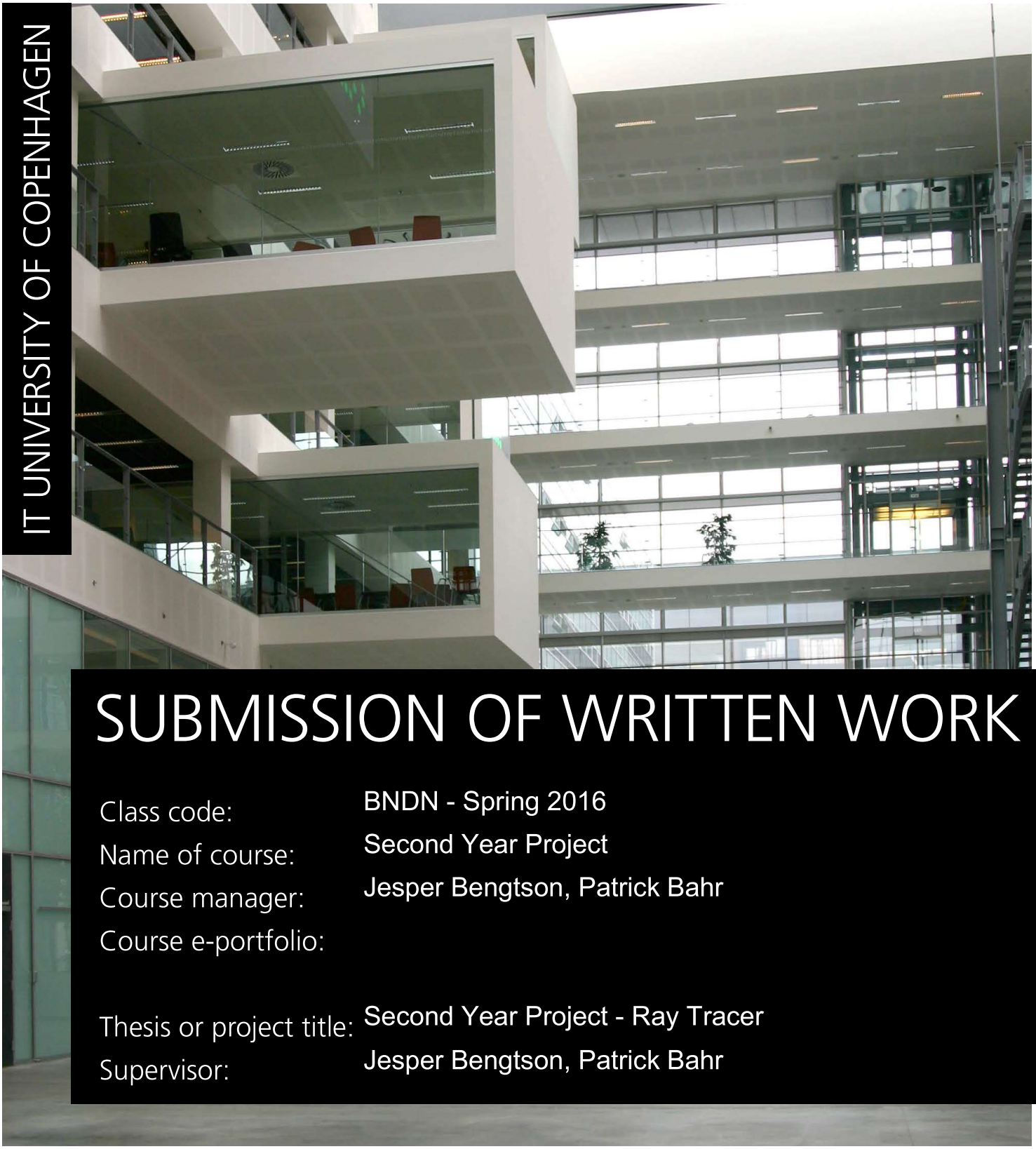
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	BNDN - Spring 2016
Name of course:	Second Year Project
Course manager:	Jesper Bengtson, Patrick Bahr
Course e-portfolio:	
Thesis or project title:	Second Year Project - Ray Tracer
Supervisor:	Jesper Bengtson, Patrick Bahr

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Daniel Nicklas Rosenberg Hansen	10/04-1994	daro@itu.dk
2.	Dennis Thinh Tan Nguyen	01/04-1993	dttn@itu.dk
3.	Maria Weybøl Techt	12/09-1993	mwey@itu.dk
4.	Nicoline Scheel	22/02-1995	nisch@itu.dk
5.	Emil Refsgaard Middelboe	02/09-1994	erem@itu.dk
6.	Thor Aakjær Valentin Olesen Nielsen	14/02-1995	tvao@itu.dk
7.			@itu.dk

Contents

1 Preface and Introduction

This report has been written in May 2016 for the Second Year Project at the IT University of Copenhagen under the supervision of Jesper Bengtson and Patrick Bahr.

Throughout the project, a ray tracer program has been developed for rendering objects in a 3-dimensional space to be displayed on a screen. It has been developed using the functional paradigm through F#, and the purpose was to implement an API provided by the supervisors.

2 Background and Description of Problem

Ray tracing is a technique for rendering objects visually on a screen. It is used for 3D modeling purposes and mathematical illustrations such as 3D plots. This domain has been widely explored and has resulted in detailed programs like Maya and 3D Studio Max. However, the requirements for this project are less ambitious and need not contain the advanced functionality of the ray mentioned above tracers.

A fundamental requirement was that the project should be developed using the functional paradigm through F#, which is a whole new solution domain for the entire group.

Initially, the team was provided with a set of fundamental requirements concerning triangle meshes, object modifications, object composition, color, light, texture, reflections, implicit surfaces, and optimization. The success criteria for the project was to be able to render properly the test suite provided by the supervisors.

3 Problem Analysis

3.1 Internal Structure - Thor

This section describes some of the different solutions that may be considered to describe the shape abstraction used to define shapes and their hit functions.

3.1.1 One Discriminated Union

A type is needed to represent all shapes required in the ray tracer while still distinguishing them on their unique internal structure. It is also needed to be able to differentiate between the different shapes and their properties. For this purpose, a "discriminated union" type may be used where each component type (called a union case) must be tagged with a label (called a case identifier or tag) so that they can be told apart or "discriminated".[1]

The following example shows how the shape abstraction might be implemented where each shape has its own label along with a component type describing their internal structure.

```
type Shape =
       | Box of Point*Point*Texture
       | Sphere of ...
       | Triangle of ...
```

While this solution groups the different types of shapes together and gives a name to the properties of the shapes, some drawbacks are to be considered. This solution may lead to problems with maintaining the code. Specifically, if it is required to add a new shape, remove one or change one, then any part of the code that uses these shapes will have to be refactored. Because of this, using an interface to represent the shape abstraction may be considered instead.

[1] [fsharpforfunandprofit, 2016]

This allows to only expose functions used to create new shapes that satisfy this interface. As a result, it is no longer bound to expose the internal structure of each shape to the client while making sure that future changes do not influence other parts of the code.

3.1.2 Singleton Discriminated Union

For further improvements, small unions (singletons) can be created with just one constructor for each shape where each shape implements the interface. As a result, it is no longer needed to refactor different areas of code where shapes are used. By way of example, it is no longer needed to modify functions using pattern matching on shapes and their hit functions. This leaves the solution more modular.

```
type Box =
       | Box of Point*Point*Texture
       interface IShape with
           member this.HitFunc (ray : Ray) = ...
```

The benefit of this approach is that you can easily obtain the properties of each shape. The downside is based on the fact that transformations are applied to these shapes, leaving their properties exposed to change and thus being unreliable. For this project, all shape modifications should be handled via transformations, making it unnecessary to store the object properties. Instead, one can settle for creation methods that take the relevant arguments and return a shape with its related hit function. This is an approach that conforms well with the functional paradigm, and the only reason to stray from it would be performance.

3.1.3 Object Expression

Lastly, because the shapes are subject to regular changes through transformations, it is unnecessary to store shapes and their properties. Instead, we only want to expose functions inside the **Shape.fsi** file. For this purpose, object expressions have proved the most advantageous to create immutable shape objects with a modular approach.

```
let new Box(lowerCorner : Point, upperCorner : Point, texture :
   Texture) =
        { new IShape with
            member this.HitFunc (ray : Ray) = ...
        }
```

This approach may be used to create objects inline without having to use class declarations. The object expressions allow one to create shapes. Transformations may then be applied on top of these shapes and return them as new shapes with modified hit functions. Altogether, the object expressions allow the creation of new shapes dynamically based on an existing base shape type and interface. Another benefit of this is that one no longer has to build many shapes to handle specific situations (e.g. transformations). Instead, one simply uses an object expression that customizes an existing shape type or provides a modified implementation of the shape interface and its hit function. As a result, the number of types created in the program is reduced, and one avoids an excessive proliferation of types.

3.2 Transformations - Daniel, Thor

3.2.1 The Idea

The core idea behind transformations is the possibility to have a single shape and inexpensively create different versions of it dynamically by applying different transformations to it. In this regard, the **Transformation** module needs a function that applies a transformation to any shape. This transformation should not transform the shape but rather the ray that is passed on to the hit function of the shape. Thus, the function has to take a function `transformHitFunction` with two parameters, a shape and a transformation, which should return a new hit function for the shape. On top of this, another `transform` function with the same two parameters should use the `transformHitFunction` and return a new shape with the modified hit function. The second function takes the first function as an argument to solve the general problem of creating a function that transforms one hit function into another.

3.2.2 Matrix Representation

Matrices are used to represent transformations in a consistent format suitable for computation. This allows transformations to be concatenated easily through multiplication. The matrices may be implemented in various ways, either manually or through the use of external math libraries. Firstly, one might consider using a library like **F# PowerPack Math**.[2] This library provides a predefined matrix representation with built-in support for certain matrix operations. However, it is not compatible with all operations required in this particular project, of which some has to be implemented manually. For this, one can use two-dimensional arrays to contain the transformations.

3.2.3 Operations

Some of the hit functions for shapes become a lot more difficult if one cannot assume that the shape is centered at the origin (or has a radius of one, for instance). By way of example, the hit function for a sphere is quite simple. However, it becomes increasingly hard with more complex shapes. In this regard, simple hit functions are used for all shapes and only modified using transformations to move them in certain ways.

3.2.4 Translation

Regarding translations, both the vector and the point need to be transformed using the inverse matrix. However, vectors are transformed differently compared to points. When transforming vectors, the additional fourth coordinate of the vector is 0 (as opposed to 1 for points). Consequently, the translation transformation has no effect on a vector.

3.3 Color, Lighting and Reflection - Daniel, Maria

The implementation of color and lighting was outlined throughout the lectures, and this was the practice followed.

Regarding the shadow ray tracing, there are two opportunities that can be used: forward and backward tracing. Forward tracing assumes that the lights in the scene spread light themselves, while backward tracing depends on the camera to know how to apply light. Forward tracing is smart to apply for baked lighting in game worlds, but when rendering live, backward tracing is the most efficient.

[2] [Microsoft, 2016]

In the real world, when lighting strikes a point, the reflections and light are usually spread in many directions. This is something that ray tracers which try to make lifelike pictures or physics simulators strive to reflect in their implementation. However, in this project, this is simply too complex, and a simpler solution was followed. Specifically, the ray tracer is based on the assumption that light only travels in one direction.

Moreover, reflection could be handled in a separate function, which would maintain a good overview of the program. However, the problem with this is that the reflections on a shape are not reflected a second time, as this function only takes the original color of the shape. This implies that color, lighting and reflections need to be handled at the same time.

3.4 Ply Parser - Thor

The PLY file represents a polygon file format used to store a collection of polygons to be rendered graphically in the ray tracer. The PLY format describes an object as a collection of vertices, faces and other elements along with the possible appearance of different properties.

As the primary goal was to parse triangle meshes, failure handling for parsing of other shapes was deemed less important.

3.4.1 The FParsec Parser Library

The parser library "FParsec" was considered to be used for parsing PLY files based on a recommendation from the supervisors. However, the "FParsec" library was deemed too complex and would thus require an unnecessary amount of time. Instead, one may focus on the specification of the ply parser in this particular project that is heavily focused on parsing vertices, faces and properties used to construct shapes composed of triangle meshes. This has been done using simple regular expressions, active patterns, mutable values, folding and a map structure to dynamically find properties.

3.4.2 The F# Idiomatic Alternative

The benefits and drawbacks from using a purely F# idiomatic approach as opposed to an imperative approach should be considered. Firstly, an imperative solution puts emphasis on simplicity and is the prevalent technique used within software development. However, it may give rise to side effects, duplicated code and require regular checks on mutable objects. Assuming the performance is hindered from an imperative design choice, immutable objects should be reconsidered to allow safe and clean concurrent programming. The mutable objects can make it notoriously hard to find bugs caused by mutable states shared between threads. On the other hand, an imperative approach is just as valid, if the performance is reasonable already and does not require the use of parallel or async programming. Besides, creating new object copies for every change can be just as costly when using a functional approach. On the other hand, the functional paradigm is self-contained and stateless making it possible to do parallel programming. In terms of the ply parser, it may either use F# data types or mutable values to store the results from parsing elements and their properties. In this regard, the notion of "count" plays a crucial role since the ply parser needs to parse elements based on the amount of elements that occur in a given PLY file. Mutable values can be used as counters within the imperative approach while the notion of count within functional programming is heavily inspired by the concept of recursion. All together, a trade off must be made between the two paradigms in practice.

3.5 Implicit Surfaces - Emil

3.5.1 Solving Polynomials

The goal of the implicit surface sub-part of the program is to handle an arbitrary polynomial and display it in a vector space. To accomplish this goal, the following polynomials must be solved. For polynomials of degree four and less there are specific ways to solve the polynomial, but for polynomials of a degree over 4, there is no specific way to solve the polynomial. To solve high degree polynomials, a number of numeric approaches are applicable.

One approach that could be used is binary partitioning, which is "searching" the polynomial for values given input on a range until a solution within a given error margin is found. This approach, while simple to implement, is quite time consuming. As the polynomials change for each ray, time is of the essence. Therefore, this approach is not the one chosen.

The approach recommended by the supervisors is Sturm's theorem in conjunction with Newton Raphson method. One of the advantages of this approach is based on the way that the range in which to look for the correct solution using a numerical approach is cut down drastically. A lower range to converge on will reduce the time of the search.

3.5.2 Abstract Expression Tree

A representation of arithmetic expressions is needed to handle arbitrary expressions with unknown variables and manipulate them. The alternative of handling them purely with primitive types may be extremely slow given that strings should be parsed and interpreted for each interaction.

The chosen approach has different levels of abstraction, where the highest is the atom type. The atom type consists of either a float or an exponent (a variable to the power of 1). The atoms are grouped in atom groups and atom groups are clustered in simple expressions. Each atom in an atom group is implicitly multiplied with the atoms next to it. The atom groups in a simple expression are implicitly added to the atom groups next to it.

This high level of abstraction is used in the poly type, which is a map from integers to a simple expression. The simple expression is implicitly multiplied with a variable to the power of its key. This allows for easy manipulation with the joints of a polynomial.

The lowest level of abstraction is found in FExpressions, which is a lot more accurate. It supports some arithmetic functions as well as primitive types. Adding this level of abstraction allows for more complex operations on expressions, which are needed to solve polynomials. Working on the highest level of abstraction reduces simplicity and introduces more code to cover some of the same bases. The lower abstraction adds the needed flexibility and heightened understandability, which made it the preferred level of abstraction to work with for most of the given tasks.

3.6 Constructive Solid Geometry - Daniel, Maria

The project needed to support constructive solid geometry, where shapes could be manipulated by union, intersection and subtraction.

It seemed evident that the result of these commands should result in a new shape with its own hit function.

As the program is three dimensional, it is only the solid shapes that support these operations (solid cylinder, sphere and box).

In order to decide on the implementation details, the following drawings were made:

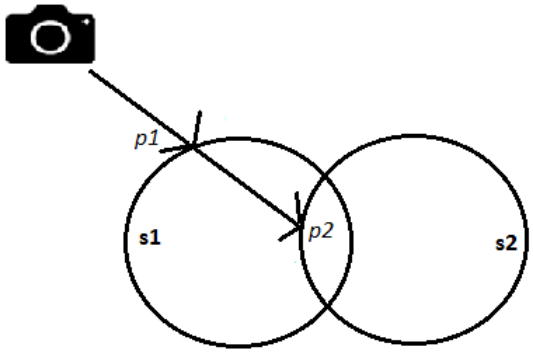


Figure 1: Intersection and union

When traversing the pixels in the scene, it is necessary to know whether a point is inside a shape. This means that the three solid shapes of this ray tracer need a definition of when a point is inside it. A generic `IsInside` function was not possible due to the difference of the shapes.

The implementation was defined by following the logic outlaid in the above figure. The hit functions have to be called on both shapes, and then the appropriate checks have to be made: for the union, it just takes the first shape that it hits as it should be treated as one shape, and for the intersection, it only draws the points that are within both shapes. However, for subtraction, only the hit function for s2 should be called (in the case where s1 is subtracted from s2), as this is the only shape we want to draw, without the points where it intersects with s1.

New bounding boxes had to be constructed for the new shapes. This is straight forward for union and intersection. However, finding the bounding box for subtraction has certain challenges. Due to the difference of the shapes (boxes, cylinders, spheres), it is very cumbersome to define a new bounding box that correctly encompasses this new shape that may ultimately take on many forms. All points would have to be considered when making this new bounding box, resulting in a loss of performance. This is the reason for the choice of keeping the entire bounding box for the shape that is being subtracted from.

3.7 KD Trees and Triangle Meshes - Nicoline, Dennis

Since triangle meshes are used for representing complex shapes, it is a requirement that the program under discussion has to render triangle meshes. However, triangle meshes can consist of a large number of triangles, making the hit function of the triangle mesh quite expensive. With this many shapes, a naive linear approach for storing triangles is out of question if a reasonable runtime is desired.

For this particular project, a KD-tree has been selected for storing shapes. KD-trees organize points in a k-dimensional space and allow for traversing with a multidimensional search key with a runtime of O(log N).[3] Furthermore, complex triangle meshes will inevitably contain clustered areas of triangles in which certain implementations of KD-trees adapt well to clustered data. Specifically, shapes can be equally distributed within the tree by splitting on the mean of the points within the tree, as opposed to simply splitting on the median of the axis-aligned bounding box of the root.

The KD-tree uses a heuristic as a guide to partition the k-dimensional space. Choosing a heuristic involves defining when to cut off empty space as well as when to stop the recursively splitting of nodes. The choice of heuristic can both speed up and slow down the construction

[3][Wikipedia, 2016]

of the tree. As an example, choosing a very critical empty space heuristic, which cuts off insignificant empty space, may result in larger memory consumption and slower building speed. On the other hand, choosing a heuristic which tolerates too much empty space will slow down the rendering of the shapes since many more rays will be generated and the intersection will be checked unnecessarily for irrelevant shapes. In summary, the structure of the KD-tree directly affects the complexity of the traversal and how many intersections have to be verified with the shapes within the tree. Some of these intersection checks can be avoided if empty space is separated from the start. Finding a balance between the two extremities is to be desired.

Steps can be taken to improve the empty space heuristic in itself. At higher levels of the tree, the empty space will be more insignificant than at a lower level of the tree. By adjusting the tolerated percentage of empty space depending on the recursion level and therefore the depth of the tree, time and memory can be saved. As an example, the empty space heuristic may be set to initially cut off empty space with fifteen percent of the enclosing bounding box. Thereafter, the next recursive call might only cut ten percent followed by seven percent and so on, reducing the risk that insignificant empty space is handled.

3.7.1 Other Spatial Data Structures

Other spatial data structures could be used instead of the KD-tree. Another example is the R-tree. Many different types of R-trees have been suggested to accommodate various applications, such as R+-trees. R+-trees are based on a compromise between R-trees and KD-trees. R+-trees and KD-trees both rely on the notion of space partitioning on axis-aligned regions. The main difference between the two data structures is that nodes in KD-trees are represented as split values that seperate planes on a given axis, while nodes are represented as bounding boxes in R+-trees. R+-trees do not cover the whole data space. This means that the R+-tree utilizes space more efficiently, whereas KD-trees do cover the entire data space. For the KD-tree, it is therefore necessary to implement a function to cut off empty spaces in order to utilize space efficiently. Another difference is that R+-trees is a dynamic index structure, meaning that operations like insertion and removal can be combined with searching and traversal of the tree during runtime. As a result, R+-trees are more efficient if the data changes dynamically during runtime. In contrast, the KD-tree is static, so once built, modifying or rebalancing a built KD-tree will require the tree to be rebuilt again. Therefore, KD-trees are not suited for dynamic ray tracing.

This ray tracer will only load bulks of data once and does not need to support runtime insertion of shapes. Thus, a KD-tree has been chosen to achieve efficient rendering of triangle meshes and shapes. The decision of choosing a KD-tree is substantiated by the fact that it is less complex to implement compared to the R+-tree, and still sufficient for efficient rendering of complex shapes. This is provided that the system is only required to load the data once and that the data sets do not change dynamically, requiring a complete reconstruction of the tree. In addition, the KD-tree was chosen over an octree. This is due to the fact that the KD-tree has a better worst-case complexity for search than the octree, which has an unbounded search complexity or O(N) in the worst case for reasonable input in the octree. Since the data for triangle meshes tends to be clustered, the complexity when searching using an octree could possibly deteriorate as the tree becomes unbalanced.

4 User guide

Since the system is developed as a back-end system, no user interface has been created. However, an API was given beforehand by the supervisors and is linked to the implementation under

discussion. To run an example scene of the program, it is necessary to run the executable file. For further information on the back-end API, please refer to the full user guide located in the appendix section 8.1.

5 Technical Description

5.1 Internal Structure - Thor

The **IShape** interface has been used to represent the abstraction of objects on the scene. The interface is implemented by all shapes to create a modular structure that is open for extension but closed for modification. The shapes rely on an abstraction rather than a concrete type implementation.

```
type IShape =
    abstract member HitFunc: Ray -> float Option*Point Option*
       Vector Option*(unit -> Material)
    abstract member HitBbox: Ray -> bool
    abstract member Bbox: BoundingBox
    abstract member IsInside: Point -> bool
```

Previously, one big discriminated Shape union was used to contain all basic shapes. However, it was soon realized that this gave rise to many internal modifications each time the shape type was extended. Specifically, one had to refactor all functions that pattern matched on shapes and the program was thus violating the open-closed solid principle making the code harder to maintain.

Instead, the implementation uses object expressions that allow the creation of immutable shape objects independent of each other. There was no need for mutable shapes where one can change e.g. the center of a circle. Instead, all shape modifications are handled with transformations. Consequently, there is no need to store the shape properties (e.g. center of a sphere). Instead, one uses a make function that takes the relevant arguments and returns a shape with the relevant hit function.

```
let new Box(lowerCorner : Point, upperCorner : Point, texture :
   Texture) =
        { new IShape with
            member this.HitFunc (ray : Ray) = ...
        }
let mkNewBox (lowerCorner : Point) (upperCorner : Point) (tex :
   Texture) = new Box(lowerCorner, upperCorner, tex)
```

As a result, a more solid functional way of representing objects without side effects has been achieved and the approach has helped establish common grounds about the contents of a shape (e.g. a sphere will always have a centre and a radius) without trading off the ability to easily modify shapes and their hit functions dynamically.

5.2 Transformations - Thor, Daniel

5.2.1 Matrix representation

Arrays of arrays are used to represent matrices and the identity matrix used for all transformation calculations:

```
let init () : matrix =  let arrayOfArrays =
                            [|[| 1.0; 0.0; 0.0; 0.0|];
                            [|0.0; 1.0; 0.0; 0.0|];
                            [|0.0; 0.0; 1.0; 0.0|];
                            [|0.0; 0.0; 0.0; 1.0|]|]
                        Array2D.init 4 4 (fun i j -> arrayOfArrays
                           .[i].[j])
```

The column and row values are then changed based on the operation type and math described in the theory:

```
let translate (x : float) (y : float) (z : float) : Transformation
    =
    let matrix = init()
    matrix.[3,0] <- x
    matrix.[3,1] <- y
    matrix.[3,2] <- z
    let invmatrix = init()
    invmatrix.[3,0] <- -x
    invmatrix.[3,1] <- -y
    invmatrix.[3,2] <- -z
    (matrix,invmatrix)
```

5.2.2 Operations

The matrix operations have been implemented according to the theory provided throughout the lectures. Thus, the operations for translating (moving), rotating, shearing and scaling have been implemented in a similar way to the above translation example.

5.2.3 Shape Transformation

Two functions are required to transform one hit function into another. For this purpose, a `transformHit` function that takes a hit and transformation and returns a modified hit function which can be implemented as illustrated:

```
let transformHit hit trans =
        let (m,inv) = trans
        match hit with
            |(dist,Some hit,Some normal,mat) ->
                        let normal = Vector.normalise (
                           transformVector normal (m))
                        (dist,Some (transformPoint hit m),Some
                           normal,mat)
            |_ -> hit
```

Regarding translations (movements), both the vector and the point need to be transformed using the inverse matrix. However, vectors are transformed differently compared to points.The transformations are applied differently using the help functions `transformPoint` and `transformVector`.

Finally, the `transformHit` function is used to inexpensively modify the hit function of shapes and return a new `transform` shape with a modified hit function based on the old shape's hit function and the transformation:

```
let transform (sh : Shape) (tr : Transformation) : Shape =
      {
          new Shape with
              member this.HitFunc (ray : Ray) =
                      let p = getOriginPoint ray
                      let v = getDirection ray
                      let (m,inv) = tr
                      transformHit (sh.HitFunc (mkRay (
                         transformPoint p inv) (transformVector
                         v inv))) (m,inv)
                  ...
      }
```

5.3 Lighting - Daniel, Maria

The camera fires a ray for each pixel in the image. When a ray is fired, it iterates over a list of shapes that are in the scene. Some of these shapes have no bounding box, such as implicit shapes and planes, and also a KD-tree containing all other shapes. The KD-tree uses its implementation of a hit function to traverse and find the nearest hit.

When a hit is returned from the ray that was fired, the amount of light that this point is exposed to is calculated. This is calculated by folding over the list of lights in the scene, and checking if any other shape is in the way. If no shapes are in the way of the light, we use the following equation to calculate the amount of light the point gets from the given angle and distance. N is the normal vector; L is the vector from the hit to the light. We use the `saturate` function to avoid negative values.

```
saturate (Vector.dotProduct N L) * getLightIntensity light *
   getLightColor light
```

These sub-equations are summed for each light that affects the point. Also, the ambient light, which affects all the shapes in the scene no matter their position.

Lastly, the result of the lighting is multiplied with the color of the material on the point, resulting in the equation:

```
color * (light1 + light2 + light3 + ambient)
```

5.4 Reflections - Daniel, Maria

The same approach is applied for reflections in a recursive function. Initially, a new ray is fired from the hit point using the reflection vector. The vector is calculated with the normal vector from the hit function. Based on this, one may derive the angle of incidence, and the angle of reflection is the same, so given this fact, the angle of reflection is calculated with the following equation, where v is the vector of incidence and n is the normal:

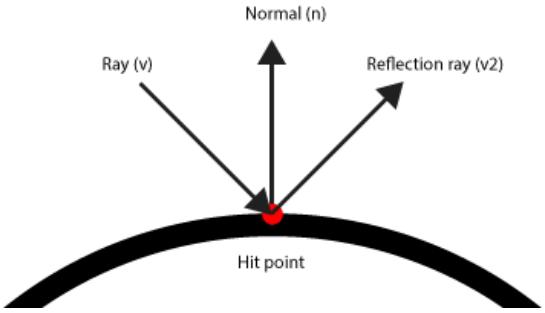


Figure 2: Reflection ray

```
V2 = v-(2.0*(Vector.dotProduct v n)*n)
```

With all the colors derived from each reflection, the color of the point with reflections is calculated. After that, the color of the reflected shape is multiplied with the reflectivity of the current. Finally, the color result is added with the color of the current object multiplied with one minus the reflectivity, because that is the amount of color not reflected. Because of the recursion, the reflection function is then called on the reflected shape, which then becomes current and so on.

5.5 Ply Parser - Thor

5.5.1 Implementation

The information described in the header is used to determine where the different pieces of information reside in the file. Specifically, the PLY parser should parse information required to build triangle meshes including vertices, triangle faces and the possible appearance of properties like normal vectors and UV coordinates. The header is parsed first to determine which lines contain these elements. This is done using a parameterized active pattern, `ParseRegex`, that uses regular expressions to match the content of different strings in the ply file:

```
let (|ParseRegex|_|) regex input =
    let m = Regex(regex).Match(input)
    if m.Success
    then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None
```

By way of example, some vertices and faces are used to determine the number of lines to be parsed for these elements and the appearance of properties is used to predetermine the normal vectors and UV coordinates for each triangle face element. The active pattern is used to pattern match on these elements as shown below:

```
let parseHeader (header: seq<string>) =
  let counter = ref 0
  for line in header do
  match line with
    | ParseRegex "element vertex (\d+)" [amount] ->
      vertexCount <- int (amount)
    | ParseRegex "element face (\d+)" [amount] ->
```

```
    faceCount <- int (amount)
  | ParseRegex "property ([A-Za-z0-9]+) ([A-Za-z0-9]+)" [
     datatype; name] ->
  if not (vertexPropertyMap.ContainsKey(name)) then do
    vertexPropertyMap.Add(name, counter.Value)
    counter := counter.Value + 1
  | _ -> ()
```

5.5.2 Mutable Values

The ply parser implementation is partially imperative due to the mutable values used to keep track of how many vertices and faces that occur in the file during parsing. After the header has been parsed, the vertices and faces are parsed according to the count of the mutable values and the possible appearance of properties in the map. This is ultimately used to predetermine the size allocated in each array containing vertices and faces parsed in the **parseVertices** and **parseFaces** functions respectively:

```
let parseVertices (vertices: seq<string>) =
       let counter = ref 0
       let vertexarr = Seq.fold (fun array a ->
                                    (Array.set array counter.Value
                                        (stringToVertex a))
                                    counter := counter.Value + 1
                                    array)
                                    (Array.zeroCreate vertexCount)
                                    vertices
       vertexarr
```

5.5.3 Properties

The counter used in the `parserHeader` function makes it possible to store properties based on their appearance and actual order within the file. As a result, the appearance and order of properties like normal vector and UV coordinates are not hard-coded and do not depend on the format of the PLY file. Instead, they are stored using a map that allows for instant look up times as opposed to iterating through the complete list of properties sequentially. In practice, this change made the parsing twice as fast. The previous ply parser version was implemented with the assumption that properties would always appear and have a certain order. Consequently, this would conflict with files where the content was not set in a particular way and create malformed elements.

The old approach has been replaced with a more dynamic approach using the previously described map to contain properties based on their possible appearance in the file independent of their position in the file:

```
let stringToVertex (s: string) : Vertex =
       let properties = s.Split[|' '|]
       let x = getProperty (getPropertyMapPosition "x"
          vertexPropertyMap)
       ...
       let nx = getProperty (getPropertyMapPosition "nx"
          vertexPropertyMap)
```

```
...
let normal = Vector.mkVector nx ny nz
mkVertex (Point.mkPoint x y z) normal v u
```

5.5.4 Parsing Comparisons

The choice of an imperative ply parser solution has given rise to a collection of benefits and drawbacks. Firstly, the performance was not affected by the mutable values when compared to a completely functional solution. Specifically, it was tried to replace the mutable values and counters with F# types and recursive calls to compare the running time, which was different within a 0,1 second margin. Thus, performance was not deemed an issue with the imperative approach. On the whole, the current ply parser conforms well to the files used in the test suite and does not comprise any performance overhead despite the partial application of concepts within both the imperative and functional paradigm from F#. However, the side effects did affect the parsing results when adding properties to the map and parsing multiple files sequentially. By way of example, the team experienced a bug when running the test suite because of the mutable values that were not reinitialized between the parsing of each file. As a result, the tests reused the properties parsed from the first file resulting in erroneous test results. Also, the mutable counter values are only assigned once in `parseHeader`, which could there easily have been replaced with the use of a tuple.

5.5.5 Failure Handling

In terms of failure handling, a custom "ParseException" is raised when certain lines cannot be parsed. This might be considered bad practice in functional programming because it is not represented by the signature of the parse function. Thus, it is not obvious what went wrong when the parser raises the exception. Alternatively, it should have been replaced with a F# type containing parser results (as described in the problem analysis) based on the outcome, success or failure. This would make it possible to see what is parsed and which lines are currently not supported by the ply parser. In this way, it becomes easier to narrow down failures, detect errors and extend the ply parser.

5.6 Implicit Surfaces - Emil

5.6.1 Polynomial Solving

The implementation solves first-degree and second-degree polynomials with specific formulas. Given time constraints, the 3rd and 4th-degree polynomials, which has specific formulas for solving, is attempted to be solved with the Newton Raphson method. The Newton Raphson method is not as efficient as the specific formulas, but it would be a sufficient initial implementation. Unfortunately, the abstract expression tree inflicted quite a bit of complication regarding run time. The building, substituting, simplifying, converting and traversing the expression tree are not always cheap operations. As the manipulation of the expression tree is necessary for every single ray fired, the rendering of implicit surfaces is quite slow.

The former implementation made use of the poly abstraction level in the hit function, and as a result of this, converted the expression to a poly and in some functions, such as polynomial long division and back again. This resulted in a lot of unnecessary operations all for the sake of using an abstraction level. The current implementation relies mostly on the expression abstraction level, and bypasses the excessive need for conversion operations. This resulted in rendering times 2-3 times faster for the implicit surfaces than before.

The Newton Raphson method is unfortunately not functioning in the current solution as the complexities from the expression tree added with the time constraints proved too difficult to overcome. As a result of this, the solution can only solve first and second-degree polynomials, but has a draft of an implementation for high degree polynomials based on Sturm's theorem and the Newton Raphson method.

5.6.2 Abstract Expression Tree

The implementation of the abstract expression tree operates primarily within the expression level. This makes handling division and roots doable without changing the atom type, which does not support division in the implementation's current state. The implementation relies heavily on the method `cleanExpr`, which isolates and rewrites divisions and root expressions.

The `simplify` method simplifies an expression to a higher abstraction level (simple expression) while collapsing additions and multiplications where possible. `cleanExpr` takes an expression tuple, which correlates to the left and right side of an equation. Given that the expressions to be solved are polynomials, it can beset equals to zero, which will allow solving the equation for the possible hit. To isolate root or division expressions, the expression tree is traversed, matching the expression types and checking whether an expression contains either a division or a root. The process of solving equations by hand is mimicked: for example, an addition with FAdd(e1, e2) where e1 is the division and e2 is not, e2 is subtracted from the right side of the expression tuple.

This is done by traversing the expression tree recursively until a division is reached. At this point, the function `cleanDivMults` is used to make sure nonsensical divisions such as '2/2' is present. The `cleanExpr` function calls itself again if there is any division or root left in the result. This is done since divisions or roots are sometimes introduced to isolate given expressions. The approach to handle roots is generally the same, but `cleanRootExp` is used to clear n roots to the power of n.

On the simple expression abstraction level, the `simplifySE` function is implemented to collapse simple expressions with the same variable. For instance, '2x-2x' should be evaluated to 0. Arguably, this problem would be solved when the substitution of values happen in the hit function. However, the fact that the simplification of simple expressions does not collapse these kinds of expressions result in some problems, especially when doing polynomial long division because the check of the given polynomials degree proved troublesome when the expressions were not simplified fully.

5.7 Constructive Solid Geometry - Daniel, Maria

The first thing necessary for implementing constructive solid geometry is to define an `IsInside` function for the solid shapes within the **ConstructiveSolidGeometry** module.

For the sphere, the distance between the point and the center needs to be within the radius:

```
(Vector.magnitude (Point.distance p centre))<=radius
```

For the cylinder, we assume that it is placed in the center. Therefore, we can check that the y coordinate is within the height of the cylinder:

```
(py<=(height/2.0)) && (py>=(-height/2.0))
```

Afterwards, a check as to the x coordinate is made, similar to that of the sphere:

```
(Vector.magnitude (Point.distance p (Point.mkPoint 0.0 py 0.0)))<=
   radius
```

Regarding the box, the upper and lower corners are given, so the only thing left to do is check that the point is within these coordinates:

```
x1<=px && px<=x2 && y1 <= py && py <=y2 && z1 <= pz && pz <= z2
```

The union is one shape that holds the two shapes it consists of. It calls the hit function for both shapes, but only returns the one with the shortest distance to the camera. It also ignores all hits within the combined shape. The bounding box for this new shape is likewise the union of the bounding boxes of the two shapes.

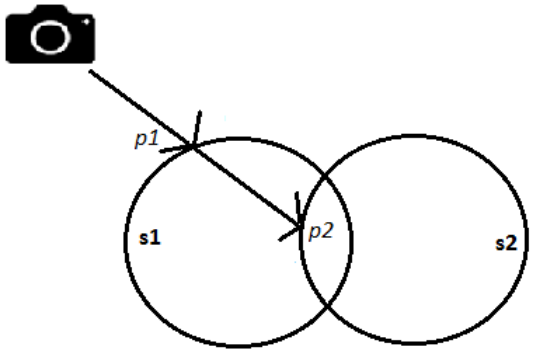


Figure 3: Constructive solid geometry

The intersection function checks whether the hit point is in the shape that it does not belong to. For instance, in the above figure, p1 is not inside s2 and should not be colored. p2 is the hit point for s2 but is also inside s1. This means that the shapes intersect here, and it should be colored. The bounding box is calculated by taking the two middle values of the highest and lowest points of the former bounding boxes and using these to make the point for the new bounding box.

In subtraction, only the hit function of the first shape is called. Suppose that we want to subtract s1 from s2 in the above figure - the only p2 would be in the subtraction function, not p1 as we only want to draw s2 and subtract the points where it intersects with s1. The only thing left to do is then to check whether this hit is inside s1 - if it is, it should not be drawn.

5.8 Optimization - Daniel

For each pixel that the camera has to render, there is a task to be performed for calculating the hits, light and reflection. These tasks can be done independently -therefore, they are carried out asynchronously. All pixels from (1,1) to (width,height) is stored as an array with width*height tasks that have to be carried out. To convert this array back to coordinated we use (i % width,i / width) where i is the index.

5.9 KD Tree - Nicoline, Dennis

5.9.1 Heuristic

When building the KD-tree, two heuristics are used. The first heuristic is used when cutting off empty spaces and the other is used to stop splitting a given node further. The latter is based on the percentage of shapes that overlap the intended split value. The split value is generated according to the distribution of shapes within the bounds corresponding to the node. The empty space cutting heuristic is defined as a percentage. The empty space is the distance

between a point and another given point, such as the mean of all shapes or an outer bounding box depending on whether an empty outer space or inner empty space is cut off. The percentage that the empty space composes of the outer bounding box is computed and compared to the heuristic. If the percentage of empty space is larger than the allowed percentage of the heuristic, the empty space is cut away. (See figure 4)

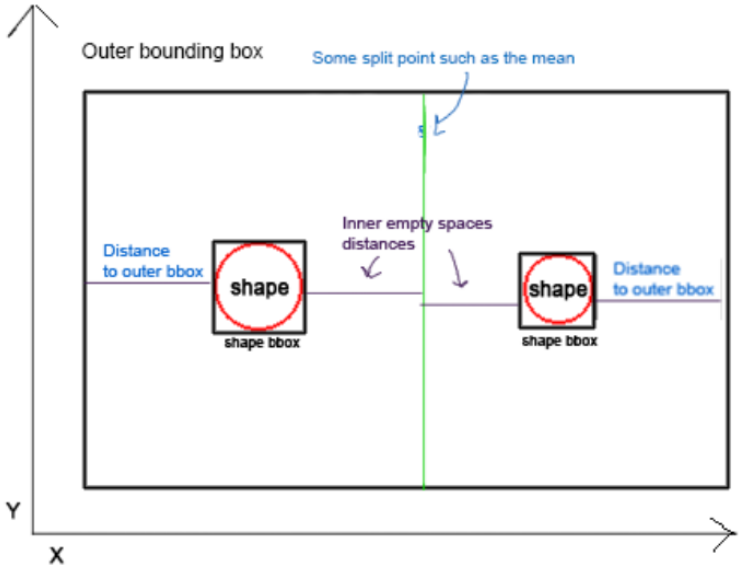


Figure 4: Illustration on inner- and outer empty spaces

It has been a design choice that empty spaces are only cut off if the empty space percentage is greater or equal to ten percent. This percentage is derived from an experiment where several heuristics (from one to fifty percent) were put up against each other. The triangle mesh used for testing contained 10474 polygons. Based on several runs, the time was measured for ray tracing and building. Considering the results (see figure 16), ten percent heuristic had the lowest average ray tracing time. The building time did not differ significantly across the different empty space heuristic values.

Since the building time of the KD-tree is minimal, the deviations between the build times had an insignificant impact on the overall time compared to the ray tracing time where the deviations were up to several seconds. Thus, ray tracing time was deemed the key factor when selecting an optimal heuristic for the system. Ten percent empty space heuristic had the best average ray tracing time while still being able to trim off unnecessary space.

Taken the extremities into consideration, one percent heuristic did have a good building and tracing time but may result in many irrelevant cuts that can result in a very deep tree structure. A heuristic of fifty percent had the slowest ray tracing time due to large empty spaces not being cut off and thus, the system had to perform additional ray tracing. Hence, the slow tracing time.

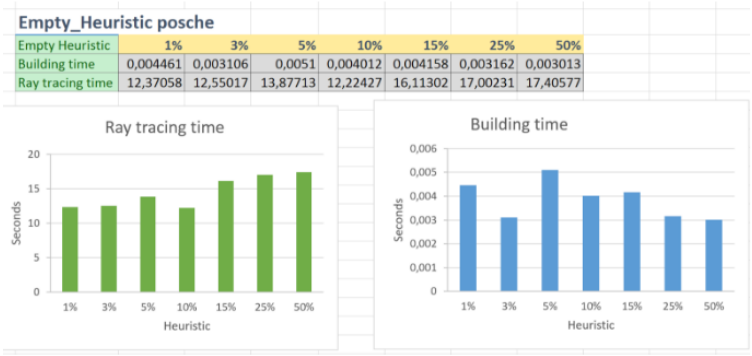


Figure 5: Rendering and build time based on heuristic chart

5.9.2 Leaf Creation Heuristic

Another heuristic being used is the heuristic that determines when the KD-tree should stop splitting and create a leaf instead. The given heuristic is set to sixty percent and was provided during the lectures as an optimal heuristic for splitting. The heuristic prevents dividing planes that contain a large number of overlapping shapes and will, therefore, return a leaf if a given ratio of overlapping shapes exceeds the heuristic.

Even if the heuristic was provided, it is still critical to understand the difference between a heuristic that is too low and a heuristic that is too high. A heuristic that is too high would result in few splits with leaves containing an enormous number of shapes that will have an impact on the rendering time. A heuristic that is too low may result in cutting on areas with few overlaps and also a tree that is unnecessarily deep which can lead to longer traversing times.

Depending on the data, no heuristic is the best heuristic as it is all about finding the sweet spot that can accommodate the data that the system handles as efficiently as possible. It may be very difficult or even impossible to find the perfect heuristic that will accommodate any situation.

A full explanation of how the tree is built can be found in the appendix section 8.3.

5.9.3 Memory Usage

Every leaf holds shapes that lie within the bounds corresponding to it. The shapes of the leaves are contained within lists. Lists are used because the number of shapes on either side of the mean within a given dimension is unknown. Since the list grows larger as more elements are added to the collection, this is not a concern. Further, the functionality of the system only demands that the collection can be iterated over, and specific elements are never accessed, leaving the list collection as a reasonable choice. It should be noted that the memory usage could be decreased by keeping shapes in an array and referring to the shapes of the leaves using their index position in the initial array. However, this would not change the fact that a list collection is necessary to hold the references to index positions. Furthermore, this would require more array accesses once the tree is traversed. The team chose to prioritize fast search time over small memory consumption.

While the memory consumption was deprioritized in the case mentioned above, other aspects of the implementation take memory consumption into account. This involved the use of functional concepts like tail-recursion, continuations and accumulating parameters. Firstly, continuations are used to compose the tree when recursively computing the left and right tree in the functions `SplitOnMean` and `SplitOnDimension`. The following example shows the continuation, which is given as a parameter to the next recursive call in `SplitOnDimension`.

The continuation is composed of two continuations which are used to build the left and right tree respectively:

```
fun( rightThree -> splitOnDimension upperBbox upperShapes
     (fun leftThree
      ->
        c (Node(largestDimension, closestMean, rightThree,
           leftThree))))
```

Using continuations and accumulating parameters ensures that the recursive call is in tail position, meaning that the expression built in the recursive steps does not need to be reduced afterwards. This allows the compiler to use tail calls, which will keep the stack space used by the recursive call constant. The memory consumption is therefore reduced. The continuation-based declaration is preferred over that of the accumulating parameter because it can handle larger lists. This is preferred over the building speed that you gain using accumulating parameters, because the KD-tree is only built once.[4]

5.10 Triangle Mesh Representation - Nicoline, Dennis

A triangle mesh can be perceived as a single shape with a bounding box, a hit function and a placement, or as a collection of many individual polygons with the same parameters. The system uses the following abstraction when creating a triangle mesh: Mesh Triangle, Triangle Mesh and Mesh Builder.

5.10.1 Mesh Triangles

The mesh triangle is the atomic level of a triangle mesh. It is an abstraction based on the IShape interface explained in section 5.1, and together with a specific number of mesh triangles, forms a triangle mesh. The mesh triangle is comparable to the simple triangle shape but differs in the implementation. Instead of storing points like ordinary triangles, mesh triangles store vertices. A vertex is another abstraction that contains a point, normal for smooth shading and u, v coordinates for texturing. Furthermore, mesh triangles accommodate enabling of smooth shading. If smooth shading is enabled, the hit function uses the calculated normals, which is the average normal of surrounding triangles, to interpolate the hit. If smooth shading is disabled, the hit function will simply calculate the normal of the current triangle and use that to flat shade the triangle.

5.10.2 Triangle Mesh

The triangle mesh consists of many mesh triangles, and all these triangles are stored within a KD-tree. Its bounding box is defined by finding the largest and smallest corner coordinates based on all of the polygons in the triangle mesh. Whenever a ray hits the triangle mesh, the KD-tree is traversed until some hit is returned.

5.10.3 Mesh Builder

Since a triangle mesh is a complex shape that consists of many sub-shapes, a **MeshBuilder** abstraction was introduced into the program. The mesh builder contains functions used to build a triangle mesh based on a given PLY file and whether smooth shading is enabled. The mesh builder utilizes the PLY parser and receives a collection of faces and vertices. If smooth

[4] [Hansen and Rischel, p.212-214]

shading is enabled, the mesh builder will then calculate. On the other hand, if none exists the average normals for every vertex is calculated. Based on the face collection that refers to index references in the vertex collection, the mesh builder constructs mesh triangles by using each vertex reference of a given face. All of the mesh triangles are added to a KD-tree and wrapped in the triangle mesh abstraction and then returned as a single shape.

By building and representing a triangle mesh as a shape, one can store triangle meshes in KD-trees with other shapes and transform them using the same functions as for ordinary shapes.

6 Test

6.1 Unit Testing

To make sure that the basic functionality was working, unit tests were made for the Vector, Point, ExprParse and ExprToPoly modules as well as the secondDegree function.

6.2 Manual Testing

Throughout the project, the functionality has mostly been tested manually through visual rendering to the screen and files. As this is a very visual project, that was the best way to make the testing.

Regarding the more mathematical concepts of the project, the interactive mode in Visual Studio was used with such as **ExprToPoly** and **ExprParse**.

6.3 Test Suite

The supervisors of the course provided a test suite that the program had to be tested against.

The test suite was split into 7 categories which tested different functionality: implicit surfaces, shapes, light, affine transformations, meshes, textures and constructive solid geometry.

Within each category, there is code for setting up different scenarios, each accompanied by a reference to a picture. In this picture, a visual representation of the scene is shown, rendered in the reference ray tracer. This serves as a base for a visual comparison between this project and a reference ray tracer.

The test suite consists of 63 single tests, none of which were passed in the beginning. This was due to bugs different places in the code, as well as bad merges that had overwritten fixed code.

Thus, whenever it was discovered that our rendition of the scene did not match the picture from the test, we would debug until it matched.

6.3.1 Deviations from the Test Suite

In general, the colors of our pictures are darker than those of the reference pictures. This is probably due to a different gamma correction. A comparison between our picture and that of the reference can be found in the appendix in Section 8.5.

As described in the technical description, cases of polynomials greater than 2 degrees are not rendered.

The placement of the camera is somewhat skewed, which can also be seen in the appendix.

The tests are running slowly compared to when rendered in our own program file. The Stanford Bunny takes 16 seconds to render in our own program file, but 25,1 seconds in the test suite. This may be due to the plane and reflections.

7 Conclusion

In conclusion, the implementation has been continuously revised based on concepts from both the imperative and functional paradigm.

The internal structure is represented with the use of an interface that has served as a modular extension of the program allowing to create shapes without exposing their internal structure. This conforms well with the use of transformations that allows to create different versions of shapes inexpensively by applying various transformations on shapes and returning them as new shapes with modified hit functions based on the implemented shape abstraction.

The camera, light and reflections work well in showing the effects of the shapes on the scene. It looks somewhat different from that of the reference ray tracer, but was kept as the team preferred these colors.

A ply parser was implemented in order to parse polygon data from a PLY file used to render complex shapes composed of triangle meshes. The benefits of using a purely functional approach did not outweigh the performance of the current solution despite its partial application of mutable values. For the future, one could improve on the failure handling of the parser. Moreover, if performance were to become an impediment, one could reconsider the trade off of using mutable objects to parse polygons that might ultimately require the use of parallel programming, leaving the current solution undesirable.

On top of this, it was crucial to store the triangle meshes in an efficient way for the purpose of rendering pictures smoothly. In this regard, a KD-tree implementation was used to partition the space in a way which allowed for efficient search and enabled rendering of complex triangle meshes. The ray tracing time was improved significantly by the KD-tree, but the heuristic leaves space for improvement. In addition, the program does not currently support other polygons than triangle meshes, which was the primary type to be used in this particular project. However, extending the program to support other polygon types could be considered.

The implicit surfaces were partially implemented, making it possible to render spheres, planes and handle expressions with roots and division. For the future, one might consider improving the expression and polynomial library to support solving n-degree equations with the Newton Raphson and Sturm methods.

Finally, the ray tracing program allows creation of complex objects using the constructive solid geometry technique to combine objects.

Altogether, the ray tracer performs well according to the general requirements outlaid throughout the course and provided in the test suite, yet leaves areas like implicit surfaces and optimization for further improvement in the future.

Bibliography

fsharpforfunandprofit. Discriminated Union Theory. https://fsharpforfunandprofit.com/posts/discriminated-unions/, 2016. [Online; accessed 25-May-2016].

Michael R. Hansen and Hans Rischel. *Functional Programming Using F#*. ISBN 978.

MSDN Microsoft. F# PowerPack Math Library. https://msdn.microsoft.com/en-us/library/hh304365(v=vs.100).aspx, 2016. [Online; accessed 25-May-2016].

Wikipedia. k-d tree. https://en.wikipedia.org/wiki/K-d_tree, 2016. [Online; accessed 26-May-2016].

8 Appendix

8.1 User guide and Examples

Since the system acts as a back-end for a given ray tracer, the system will only provide an API, which any user interface can call to. In this sense, no user interface has been made for this project and thus no manual are created to accommodate for this. Despite not having a UI, this section will cover the usage of the API and how one can render various scenes with the created ray tracer.

8.1.1 Setting up mandatory settings

Before one can render anything, one must define a camera and the ambient light for the scene.

The camera can be created by using the function

```
let camera = mkCamera (Point.mkPoint 0.0 0.0 0.0) (Point.mkPoint
    0.0 0.0 0.0) (Vector.mkVector 0.0 1.0 0.0)
```

The parameters can be explained:

- First parameter = position of x, y and z
- Second parameter = look at point of x, y and z
- Third parameter = Up-vector

The ambient light is defined as:

```
let ambientLight = mkAmbientLight(mkColour 1.0 1.0 1.0) 0.1
```

The parameters can be explained:

- First parameter = is a colour that takes an R, B and G colour value
- Second parameter = is the intensity of the ambient light

8.1.2 Loading Textures

Before a shape can be created, a texture must be defined. A texture can be a simple color or a picture.

Picture Texture
To use a picture as a texture

```
let texture = loadTexture ("texturePath.png")
```

The parameters can be explained:

- First parameter = is the path to the given.

Non-picture texture
To use a solid color as a texture one can use the function:

```
let texture = mkMatTexture (mkMaterial (mkColour 0.0 0.0 0.0) 0.5
    )
```

The parameters can be explained:

- First parameter = is the material that takes a colour and a reflectivity value
- Second inner parameter = is a colour that takes R, B and G values

8.2 Creating light sources

To create light sources for the scene, one can use the following function

```
let light = mkLight (Point.mkPoint 0.0 0.0 0.0) (mkColour 1.0 1.0
    1.0) 0.5q
```

The parameters can be explained:

- First parameter = is the position of the light source
- Second parameter = is a colour that takes R, B and G values
- Third parameter = Is the light intensity

8.2.1 Creating simple shapes

To create simple shapes, such as the sphere, plane, triangle and etc. the following function can be used. Please notice that the rest of the shapes can be found in the shapes.fsi. All shapes has the "mk" prefix in front of the function that constructs them.

Spheres example:

```
let shape = mkSphere (Point.mkPoint 0.0 0.0 0.0) 10.0 texture
```

The parameters can be explained:

- First parameter = center point of sphere x, y and z
- Second parameter = Radius of sphere
- Third parameter = texture for the sphere

8.2.2 Creating Triangle mesh

To create a triangle mesh the following function can be applied:

```
let shape = buildTriangleMesh texture isSmoothShade "filename.ply"
```

The parameters can be explained:

- First parameter = is the texture of the triangle mesh
- Second parameter = a boolean value whether smooth shading should be enabled
- Third parameter = path to PLY file

Please note that one may be required to transform the loaded triangle mesh before it has a desirable appearance.

8.2.3 Transformation

If desired, a shape can be transformed in various ways. The following transformations be applied. The API for transformation can be found in Transformation.fsi

- rotate
- translate
- scale
- shearing
- mirror

Example of scaling To scale a shape, one must define the scale transformation and its values. The values will scale on the x, y or z axis. To use the transformation, one must use the "transform" function that takes a shape and a transformation, such as scale.

```
let scale = scale 1.0 1.0 1.0
let transformedShape =  transform shape scale
```

Merging transformations If one is required to perform many transformation, one can merge the transformations before applying it to a shape. the function mergeTransformation will take a list of all transformation and return a merged transformation that can be used to transform a shape.

```
let Mergedtrans = mergeTransformations [(scale 2.0 2.0 2.0) ; (
   translate  0.0 10.0 0.0)]

let transformedShape = transform shape mergedTrans
```

8.2.4 Creating and rendering the scene

When the camera has been placed, ambient lights added and all shapes have been built one can construct a scene and render it by using the following function

```
let scene  = mkScene [shape1 ; shape2 ; shapeN ]
                [light1 ; light 2; lightN] ambient camera 0

renderToScreen scene \\renders the scene
```

The parameters of mkScene can be explained:

- First parameter = is the list of all shapes that are to be rendered
- Second parameter = is the list of all light sources that are to be rendered.
- Third parameter = is the ambient light
- Fourth parameter = is the number of reflection the system must render.

8.3 Building the KD-Tree

The tree is given a predefined set of shapes as well as its current bounding box. That is the bounding box that contains the whole tree. (See figure 6)

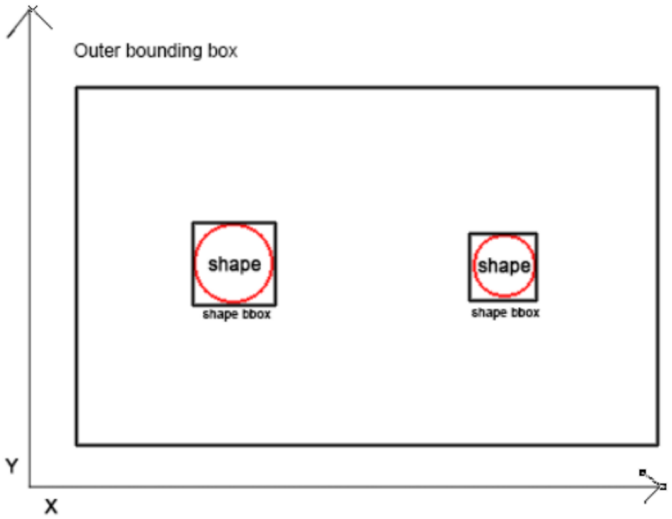


Figure 6: Illustration the initial scene with two shapes.

Based on the bounding box, the largest dimension, which is the length of x, y and z are found. The acquired dimension is used to define the operations which are to be called when splitting and trimming empty space. (See figure 7)

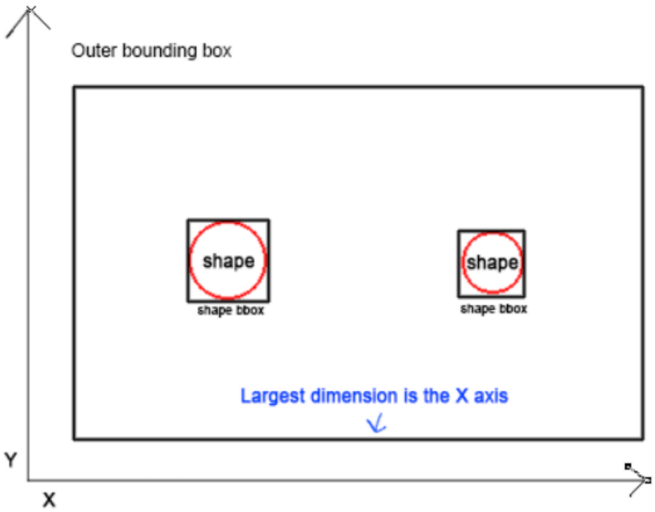


Figure 7: Finding the largest dimension. X is the largest dimension here.

8.3.1 Cutting off empty space

When the desired dimension is found, the next step is to cut of empty spaces in this case cutting the empty outer space. The system will find the two outermost coordinates and calculate their

ratio based on distance to the outer bounding box wall. The ratio is compared with an empty split heuristic and if the ratio, either the upper or lower ratio, is greater or equal to the heuristic a cut will occur and two new child bounding box is created based on the point with the greatest ratio. Note that the system will do one cut at a time on each recursive call. It prioritizes the lower ratio first, then the above ratio and lastly if no outer ratios are greater or equal to the heuristic the system will continue to trim inner empty spaces or split based on that.

The scenarios are illustrated as following:

- Lower ratio >= upper ratio && empty space ratio ratio.
  - Lowest point is selected as split point and empty space is cut off (See figure 8 and figure 9)

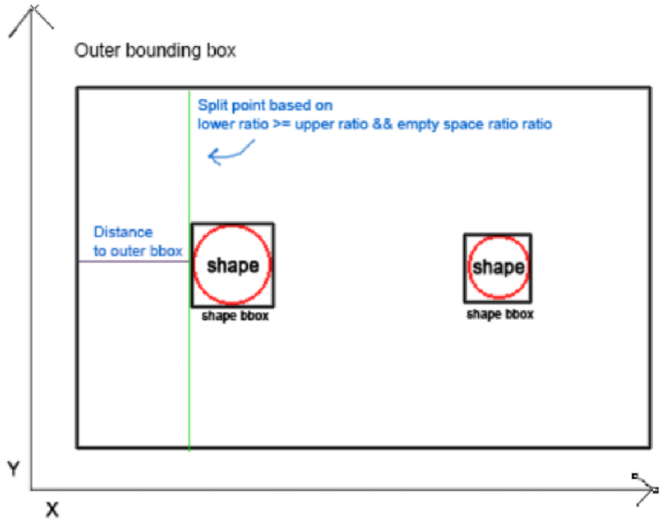


Figure 8: Finding outer space distance of lower and calculate trimming point

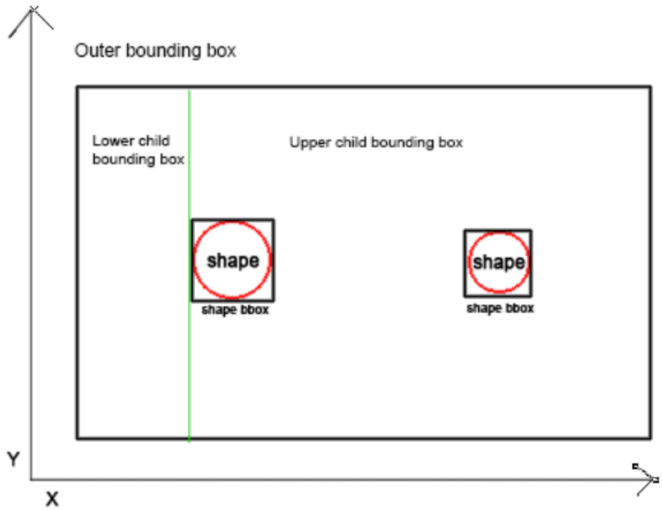


Figure 9: Lower empty space cut off and child bounding boxes is created.

- upper ratio >= empty space ratio
  - upper point is selected as split point and empty space is cut off (See figure 10)

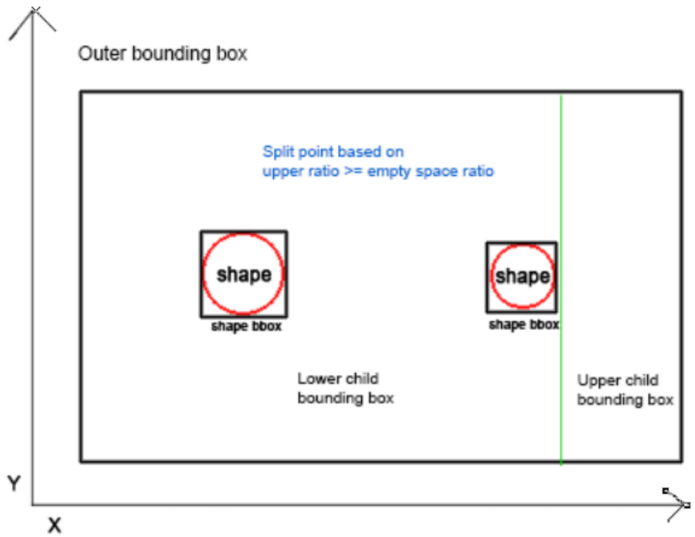


Figure 10: Cutting of upper empty space.

8.4 Finding the mean and determine shapes location

After every possible outer empty spaces are cut off, the system will proceed to split based on the shapes. for all shapes, a mean is calculated. The mean is the average coordinate based on the current dimension and every coordinate of the shapes (See figure 11). This mean is the middle coordinate of all the shapes and the system will use the mean to distribute and calculate where each shape is located, that is if they are above, between or below the mean, according to the shapes coordinates.

- shape upper point < mean –> place shape in lower
- shape lower point > mean –> place shape in upper
- shape upper point > mean && shape lower point < mean –> shape is in between and is added to upper and lower (a counter is increased for each shape that is in between. This counter is used to calculated a ratio)

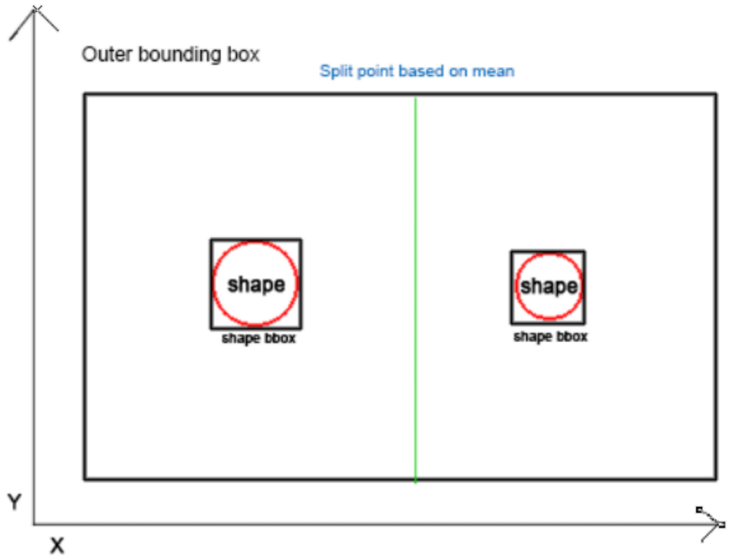


Figure 11: Mean point with one shape above and one shape below the mean

8.4.1 Splitting further or returning leaf

After the distribution of the shapes, the system will either divide the three further or return the leaves containing their respective shapes. This decision is determined based on a split heuristic and the ratio of how many shapes are overlapping the mean.

8.4.2 Returning a leaf

If either ratio of upper and lower is greater or equal to the split heuristic, no split will occur and a leaf is returned. (See figure 12)

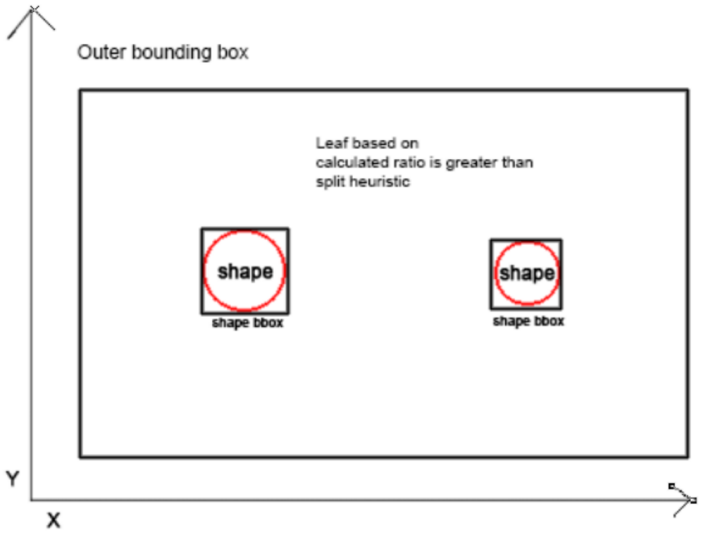


Figure 12: Leaf is returned. No additional splits will occur on the given bounding box.

8.4.3 Splitting the bounding box and calculating inner empty space

However if the ratios are less than the heuristic, the system will proceed to trim the inner spaces. This process is similar to trimming the outer space but instead of comparing with the distance from two outermost shapes with the outer bounding; two innermost shapes are found and their distance to the mean is calculated and ratios are acquired. (See figure 13) Based on the empty space heuristic, if either ratio is greater than the heuristic a split will occur (see figure 14) and two child bounding boxes are constructed with their respective shapes.re (See figure 15) Again, the system will prioritize the lower side first before the upper side and the whole process restarts until a leaf is returned.

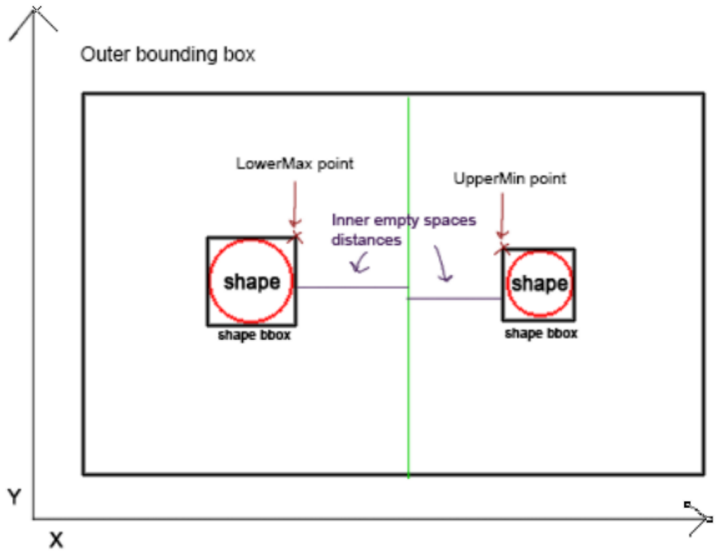


Figure 13: Calculating inner distances to mean to use as ratio.

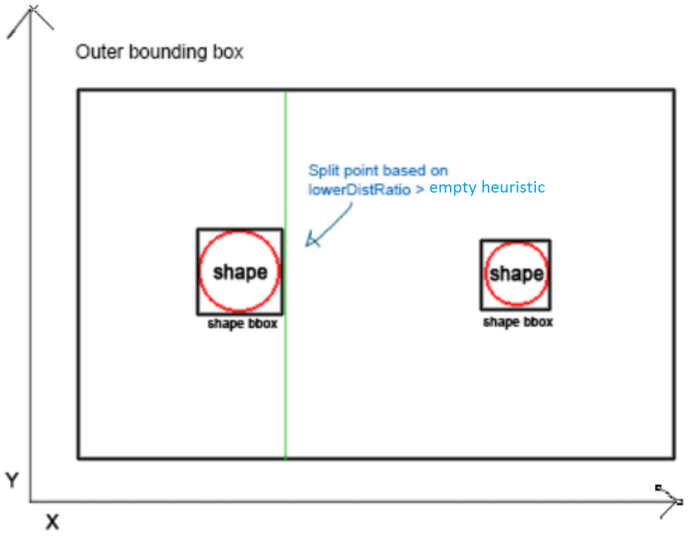


Figure 14: Example when lower ratio is greater than empty heuristic. The mean/split point is moved to lower max point.

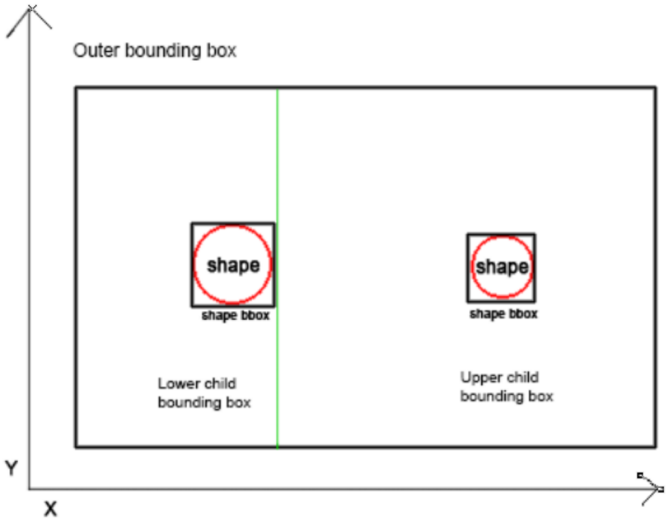


Figure 15: Split is done based on figure 14 and two new child bounding boxes are created.

8.5 Pictures

Figure 16: This is the final scene rendered with the ray tracer!

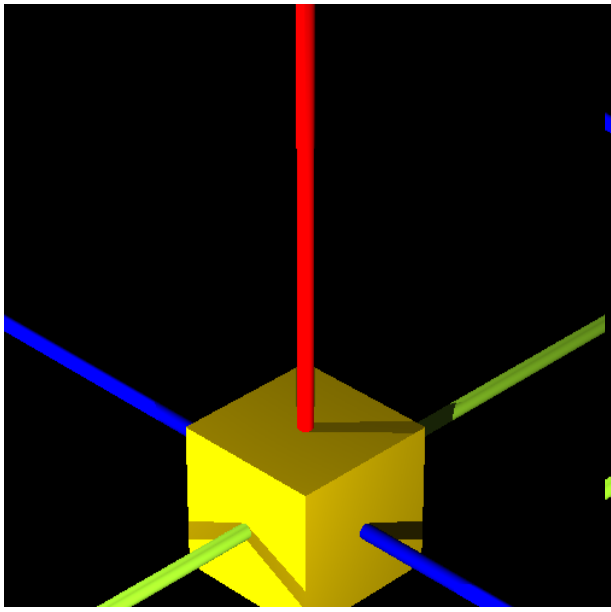


Figure 17: Result

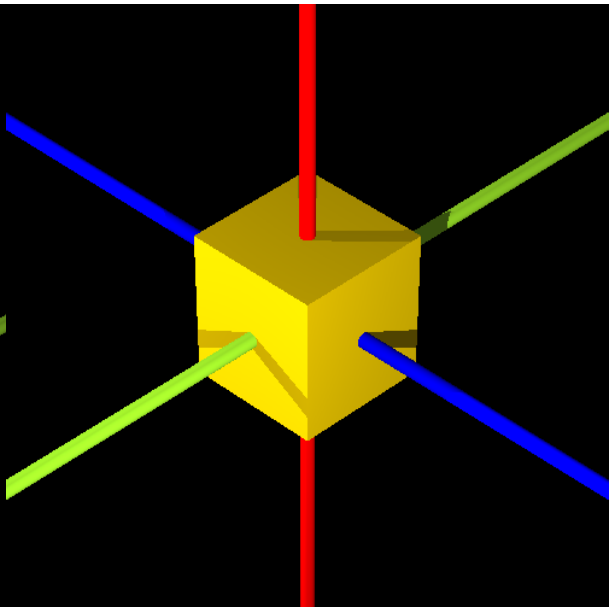


Figure 18: Reference

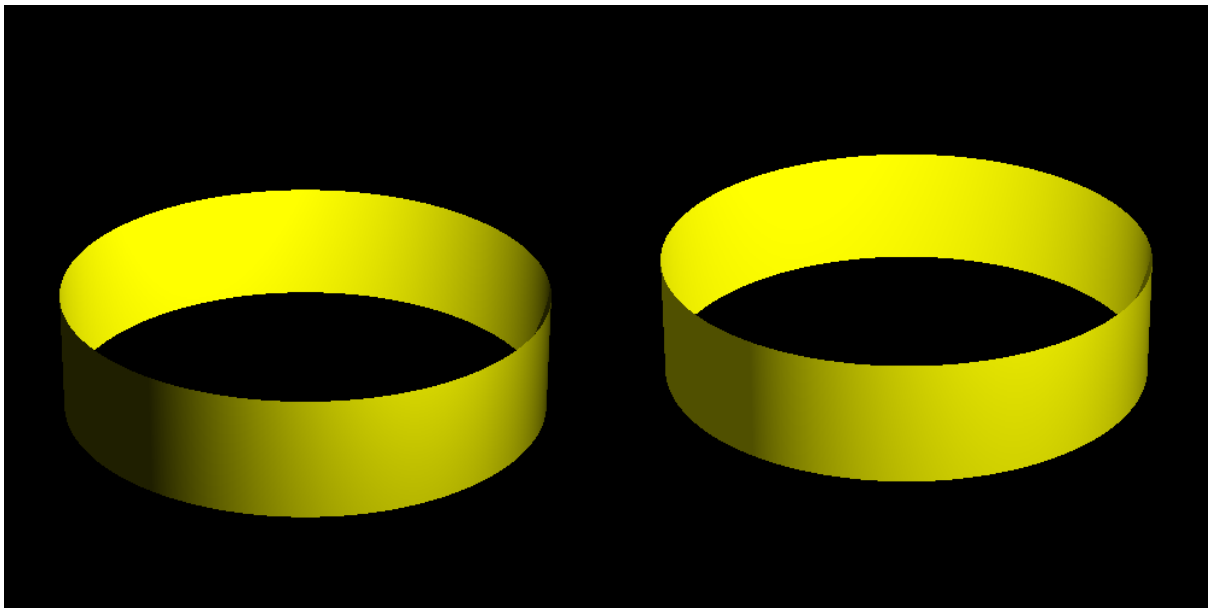


Figure 19: Result

Figure 20: Reference

8.6 Code

8.6.1 API.fsi

```
namespace API

module API =
  type vector = Vector.Vector
  type point = Point.Point
  type colour = RayTracer.SceneAssets.Colour
```

```
type material = RayTracer.SceneAssets.Material
type shape = RayTracer.Shape.IShape
type texture = RayTracer.SceneAssets.Texture
type baseShape = (texture -> shape)
type camera = RayTracer.Camera.Camera
type scene = RayTracer.Scene.Scene
type light = RayTracer.SceneAssets.LightType
type ambientLight = RayTracer.SceneAssets.LightType
type transformation = RayTracer.Transformation.Transformation

val mkVector : x:float -> y:float -> z:float -> vector
val mkPoint : x:float -> y:float -> c:float -> point
val fromColor : c : System.Drawing.Color -> colour
val mkColour : r:float -> g:float -> b:float -> colour

/// Construct a material with the given colour and reflectivity.
    Reflectivity
/// is the ratio of how much the material reflects (1.0 means it
    is a perfect
/// mirror and 0.0 means that the material does not reflect
   anything).
val mkMaterial : colour -> reflectivity : float -> material
/// Textures are functions that take x-y coordinates and produce
    a material.
/// The x-y coordinates range over the texture space specified
   for the
/// individual basic shapes (mkSphere, mkPlane etc.).
val mkTexture : (float -> float -> material) -> texture
/// Construct a texture with a constant material for each point.
val mkMatTexture : material -> texture


/// Construct a textured shape from a base shape.
/// Basic shapes are textured according to the texture space
   given.
val mkShape : baseShape -> texture -> shape

/// Construct a sphere.
/// texture coordinates: [0,1] X [0,1]
val mkSphere : center : point -> radius : float -> texture ->
   shape
/// Construct a rectangle.
/// texture coordinates: [0,1] X [0,1]
val mkRectangle : corner : point -> width : float -> height :
   float -> texture -> shape
/// Constructe a triangle.
val mkTriangle : a:point -> b:point -> c:point -> texture ->
   shape
/// Construct a plane with the equation z = 0,
```

```
/// i.e. the x-y plane
/// texture coordinates: R X R
val mkPlane : texture -> shape
/// Construct an implicit surface.
/// texture coordinates: {(0,0)}, i.e. has only a single
   material
/// The gramar for valid expressions are the following and you
   will need to create a parser for it
/// x := string
/// n := integer
/// f := float
/// e := e + e (addition)
///      e - e (subtraction)
///      -e    (negation)
///      e * e (multiplication)
///      e / e (division)
///      e^n   (exponent)
///      e_n   (root)
///      (e)   (parenthesis)
///      x     (variable)
///      n     (integer number)
///      f     (floating point number)
///
/// Note that the expression can contain both floats and
   integers. The operators bind in the expected order
/// (note that negation binds the hardest (-x^2) is (-x)^2 and
   not -(x^2)
val mkImplicit : string -> baseShape
/// Load a triangle mesh from a PLY file.
/// texture coordinates: [0,1] X [0,1]
val mkPLY : filename : string -> smoothShading : bool ->
   baseShape
/// construct a hollow cylinder (i.e. open on both ends)
/// texture coordinates: [0,1] X [0,1]
val mkHollowCylinder : center : point -> radius : float ->
   height : float -> texture -> shape
/// construct a solid cylinder (i.e. closed on either end by a
   disk)
/// texture space: hollow cylinder part is textured like
   mkHollowCylinder;
///                top and bottom disk are textured like mkDisk
val mkSolidCylinder : center : point -> radius : float -> height
    : float ->
                      cylinder: texture -> bottom : texture ->
                         top : texture -> shape
/// construct a disk at point p in the plane parallel
/// to the x-y plane
/// texture coordinates: [0,1] X [0,1]
val mkDisc : p : point -> radius : float -> texture -> shape
```

```
/// construct an axis-aligned box with lower corner low
/// and higher corner high
/// textures: the six faces of the box are textured like
   mkRectangle
val mkBox : low : point -> high : point -> front : texture ->
   back : texture ->
            top : texture -> bottom : texture -> left : texture
               -> right : texture  -> shape

/// union produces the union of s1 and s2 in such a way that all
    internal edges are deleted.
val union : s1: shape -> s2 : shape -> shape
val intersection : shape -> shape -> shape
/// subtracts s2 from s1 in such a way that s2's texture is
   maintained in the places where s1 is cut.
val subtraction : s1 : shape -> s2 : shape -> shape
/// group works like union, but does not remove internal edges
val group : shape -> shape -> shape

/// Construct a camera at 'position' pointed at 'lookat'. The '
   up' vector describes which way is up.
/// The viewplane is has dimensions 'unitWidth' and 'unitHeight
   ', has a pixel resolution of 'pixelWidth'
/// times 'pixelHeight', and is 'distance' units in front of the
    camera.
val mkCamera : position : point -> lookat : point -> up : vector
    -> distance : float ->
  unitWidth : float -> unitHeight : float -> pixelWidth : int ->
      pixelHeight : int -> camera

val mkLight : position : point -> colour : colour -> intensity :
    float -> light

val mkAmbientLight : colour : colour -> intensity : float ->
   ambientLight

val mkScene : shapes : shape list -> lights : light list ->
   ambientLight -> camera -> max_reflect : int -> scene
val renderToScreen : scene -> unit
val renderToFile : scene -> filename : string -> unit


/// For rotations all angles are in radians. Note that angles
   greater than 2*pi and less than zero are possible.
val rotateX : angle : float -> transformation
val rotateY : angle : float -> transformation
val rotateZ : angle : float -> transformation
val sheareXY : distance : float -> transformation
val sheareXZ : distance : float -> transformation
```

```
    val sheareYX : distance : float -> transformation
    val sheareYZ : distance : float -> transformation
    val sheareZX : distance : float -> transformation
    val sheareZY : distance : float -> transformation
    val scale : x : float -> y : float -> z : float ->
       transformation
    val translate : x : float -> y : float -> z : float ->
       transformation
    val mirrorX : transformation
    val mirrorY : transformation
    val mirrorZ : transformation
    /// Merge the given list of transformations into one, such that
       the resulting
    /// transformation is equivalent to applying the individual
       transformations
    /// from left to right (i.e. starting with the first element in
       the list).
    val mergeTransformations : transformation list -> transformation
    val transform : shape -> transformation -> shape
```

8.6.2 API.fs

```
namespace API
open RayTracer

module API =
  type vector = Vector.Vector
  type point = Point.Point
  type colour = SceneAssets.Colour
  type material = SceneAssets.Material
  type shape = Shape.IShape
  type texture = SceneAssets.Texture

  type camera = Camera.Camera

  type scene = Scene.Scene
  type light = SceneAssets.LightType
  type ambientLight = SceneAssets.LightType
  type transformation = Transformation.Transformation
  type baseShape = (texture -> shape)

  let mkVector (x : float) (y : float) (z : float) : vector =
     Vector.mkVector x y z
  let mkPoint (x : float) (y : float) (z : float) : point = Point.
     mkPoint x y z
  let fromColor (c : System.Drawing.Color) : colour = SceneAssets.
     fromColor c
```

```
let mkColour (r : float) (g : float) (b : float) : colour =
   SceneAssets.mkColour r g b

let mkMaterial (c : colour) (r : float) : material = SceneAssets
   .mkMaterial c r
let mkTexture (f : float -> float -> material) : texture =
   SceneAssets.mkTexture f
let mkMatTexture (m : material) : texture = SceneAssets.
   mkMatTexture m

let mkShape (b : baseShape) (t : texture) : shape = b t
let mkSphere (p : point) (r : float) (t : texture) : shape =
   Shape.mkSphere p r t
let mkTriangle (a:point) (b:point) (c:point) (t : texture) :
   shape = Shape.mkTriangle a b c t
let mkRectangle (corner : point) (width : float) (height : float
   ) (t : texture) : shape = Shape.mkRectangle corner width
   height t
let mkPlane (t : texture) : shape = Shape.mkPlane t
let mkImplicit (s : string) : baseShape = (fun tex ->
   ImplicitSurface.mkImplicit s tex)
let mkPLY (filename : string) (smooth : bool) : baseShape = (fun
    tex -> MeshBuilder.buildTriangleMesh tex smooth filename)


let mkDisc (c : point) (r : float) (t : texture) : shape = Shape
   .mkDisc c r t
let mkBox (low : point) (high : point) (front : texture) (back :
    texture) (top : texture) (bottom : texture) (left : texture)
    (right : texture) : shape = Shape.mkBox low high front back
   top bottom left right

let group (s1 : shape) (s2 : shape) : shape = Shape.group s1 s2
let union (s1 : shape) (s2 : shape) : shape =
   ConstructiveGeometry.union s1 s2
let intersection (s1 : shape) (s2 : shape) : shape =
   ConstructiveGeometry.intersection s1 s2
let subtraction (s1 : shape) (s2 : shape) : shape =
   ConstructiveGeometry.subtraction s1 s2

let mkCamera (pos : point) (look : point) (up : vector) (zoom :
   float) (width : float)
  (height : float) (pwidth : int) (pheight : int) : camera =
     Camera.mkCamera pos look up zoom width height pwidth
     pheight
let mkLight (p : point) (c : colour) (i : float) : light =
   SceneAssets.mkLight p c i
let mkAmbientLight (c : colour) (i : float) : ambientLight =
   SceneAssets.mkAmbientLight c i
```

```
let mkScene (s : shape list) (l : light list) (a : ambientLight)
    (c : camera) (m : int) : scene = Scene.mkScene s l a c m
let renderToScreen (sc : scene) : unit = Scene.renderToScreen sc
let renderToFile (sc : scene) (path : string) : unit = Scene.
   renderToFile sc path

let translate (x : float) (y : float) (z : float) :
   transformation = Transformation.translate x y z
let rotateX (angle : float) : transformation = Transformation.
   rotateX angle
let rotateY (angle : float) : transformation = Transformation.
   rotateY angle
let rotateZ (angle : float) : transformation = Transformation.
   rotateZ angle
let sheareXY (distance : float) : transformation =
   Transformation.sheareXY distance
let sheareXZ (distance : float) : transformation =
   Transformation.sheareXZ distance
let sheareYX (distance : float) : transformation =
   Transformation.sheareYX distance
let sheareYZ (distance : float) : transformation =
   Transformation.sheareYZ distance
let sheareZX (distance : float) : transformation =
   Transformation.sheareZX distance
let sheareZY (distance : float) : transformation =
   Transformation.sheareZY distance
let scale (x : float) (y : float) (z : float) : transformation =
    Transformation.scale x y z
let mirrorX : transformation = Transformation.mirrorX
let mirrorY : transformation = Transformation.mirrorY
let mirrorZ : transformation = Transformation.mirrorZ
let mergeTransformations (ts : transformation list) :
   transformation = Transformation.mergeTransformations ts
let transform (sh : shape) (tr : transformation) : shape =
   Transformation.transform sh tr


let mkHollowCylinder (c : point) (r : float) (h : float) (t :
   texture) : shape =
      let x,y,z = Point.getCoord c
      transform (Shape.mkCylinder r h t) (translate x y z)
let mkSolidCylinder (c : point) (r : float) (h : float) (t :
   texture) (bottom : texture) (top : texture) : shape =
      let x,y,z = Point.getCoord c
      transform (Shape.mkSolidCylinder r h t top bottom) (
         translate x y z)
```

8.6.3 ExprParse.fsi

```
module ExprParse

type terminal = Add | Min | Mul | Div | Pwr | Root | Lpar | Rpar |
    Int of int | Float of float | Var of string
exception Scanerror
val scan: char seq -> terminal list
val insertMult: terminal list -> terminal list
//val arithmetic: terminal list -> float

type expr = FNum of float | FInt of int | FVar of string | FAdd of
    expr*expr | FMin of expr*expr | FMult of expr*expr | FDiv of
   expr*expr | FExponent of expr*int | FRoot of expr*int
exception Parseerror
val parse: terminal list -> expr
val parseStr: seq<char> -> expr
val dotAST: expr -> string
```

8.6.4 ExprParse.fs

```
module ExprParse

(* Grammar:

E    = T Eopt .
Eopt = "+" T Eopt | "-" T Eopt | e .
T    = F Topt .
Topt = "*" F Topt | "/" F Topt | e .
F    = App Fopt .
Fopt = "^" Int | e .
App  = "sqrt" P | P.
P    = Int [ Float | Var | "(" E ")" .

e is the empty sequence.
*)

type terminal =
  Add | Min | Mul | Div | Pwr | Root | Lpar | Rpar | Int of int |
     Float of float | Var of string

let isblank c = System.Char.IsWhiteSpace c
let isdigit c  = System.Char.IsDigit c
let isletter c = System.Char.IsLetter c
let isletterdigit c = System.Char.IsLetterOrDigit c

let explode s = [for c in s -> c]

let floatval (c:char) = float((int)c - (int)'0')
let intval(c:char) = (int)c - (int)'0'

exception Scanerror
```

```
let rec scnum (cs, value) =
  match cs with
    '.' :: c :: cr when isdigit c -> scfrac(c :: cr, (float)value,
        0.1)
  | c :: cr when isdigit c -> scnum(cr, 10* value + intval c)
  | _ -> (cs,Int value)     (* Number without fraction is an
     integer. *)
and scfrac (cs, value, wt) =
  match cs with
    c :: cr when isdigit c -> scfrac(cr, value+wt*floatval c, wt
       /10.0)
  | _ -> (cs, Float value)

let rec scname (cs, value) =
  match cs with
    c :: cr when isletterdigit c -> scname(cr, value + c.ToString
       ())
  | _ -> (cs, value)

let scan s =
  let rec sc cs =
    match cs with
      [] -> []
    | '+' :: cr -> Add :: sc cr
    | '-' :: cr -> Min :: sc cr
    | '*' :: cr -> Mul :: sc cr
    | '/' :: cr -> Div :: sc cr
    | '^' :: cr -> Pwr :: sc cr
    | '(' :: cr -> Lpar :: sc cr
    | ')' :: cr -> Rpar :: sc cr
    | '-' :: c :: cr when isdigit c -> let (cs1, t) = scnum(cr, -1
        * intval c)
                                          t :: sc cs1
    | '_' :: cr -> Root :: sc cr
    | c :: cr when isdigit c -> let (cs1, t) = scnum(cr, intval c)
                                t :: sc cs1
    | c :: cr when isblank c -> sc cr
    | c :: cr when isletter c -> let (cs1, n) = scname(cr, (string
       )c)
                                  Var n :: sc cs1
    | _ -> raise Scanerror
  sc (explode s)

exception WrongFormatError // when ints and floats are placed side
    by side, an error must have occurred

let rec insertMult = function
  Float r :: Var x :: ts -> Float r :: Mul :: insertMult (Var x::
```

```
     ts)
| Float r1 :: Float r2 :: ts -> raise WrongFormatError
| Float r :: Int i :: ts -> raise WrongFormatError
| Var x :: Float r :: ts -> Var x :: Mul :: insertMult (Float r::
   ts)
| Var x1 :: Var x2 :: ts -> Var x1 :: Mul :: insertMult (Var x2::
   ts)
| Var x :: Int i :: ts -> Var x :: Mul :: insertMult (Int i::ts)
| Int i :: Float r :: ts -> raise WrongFormatError
| Int i :: Var x :: ts -> Int i :: Mul :: insertMult (Var x::ts)
| Int i1 :: Int i2 :: ts -> raise WrongFormatError
| Float r :: Lpar :: ts -> Float r :: Mul :: insertMult (Lpar::ts)
| Var x :: Lpar :: ts -> Var x :: Mul :: insertMult (Lpar::ts)
| Int i :: Lpar :: ts -> Int i :: Mul :: insertMult (Lpar::ts)
| t :: ts -> t :: insertMult ts
| [] -> []

type expr =
  | FNum of float
  | FInt of int
  | FVar of string
  | FAdd of expr * expr
  | FMin of expr * expr
  | FMult of expr * expr
  | FDiv of expr * expr
  | FExponent of expr * int
  | FRoot of expr * int


exception Parseerror
let rec E (ts:terminal list) = (T >> Eopt) ts
and Eopt (ts, inval) =
    match ts with
        |Add :: tr ->   let (x,y) = T tr
                        Eopt (x,FAdd(inval,y))
        |Min :: tr ->   let (x,y) = T tr
                        Eopt (x,FMin(inval,y))
        |_ -> (ts,inval)

and T ts = (F >> Topt) ts
and Topt (ts, inval) =
    match ts with
    |Mul :: tr -> let (x,y) = F tr
                  Topt (x,FMult(inval,y))
    |Div :: tr -> let (x,y) = F tr
                  Topt (x,FDiv(inval,y))
    |_ -> (ts,inval)
and F ts = (P >> Fopt) ts
and Fopt (ts, inval)=
```

```
    match ts with
    |Pwr :: tr -> match tr with
                    Int x::tp -> (tp,FExponent(inval, x))
                    | _ -> raise Parseerror
    |Root :: tr -> match tr with
                    |Int x::tp -> (tp, FRoot(inval, x))
                    | _ -> raise Parseerror
    |_ -> (ts,inval)
and P ts =
    match ts with
         Int i:: tr -> (tr,FNum(float(i)))
        |Float f:: tr -> (tr,FNum(f))
        |Var s :: tr -> (tr,FVar(s))
        |Lpar :: tr ->
                            let (rest,exp) = E tr
                            match rest with
                                Rpar::r ->  (r,exp)
                                |_ -> raise Parseerror
        |_ -> match ts with
              | Min :: tr -> let (rest, expr) = P tr
                             (rest, FMin(FInt 0, expr))
              | _ -> raise Parseerror

let parse ts =
  match E ts with
    ([], result) -> result
  | _ -> raise Parseerror

let parseStr s = parse (scan s)
let dotAST ast =
  let fixStr (s:string) = s.Replace ("\"", "\\\"")
  let genDot s n e = "digraph G {\nlabel=\"" + (fixStr s) + "\"\n"
      + n + e + "\n}"
  // i is unique label such that nodes and edges are unique in
     DiGraph.
  let genNodeStr i l = "Node"+(string i)+" [label=\""+l+"\"];\n"
  let genEdgeStr i1 i2 = "Node"+(string i1)+" -> " + "Node"+(
     string i2)+";\n"
  // Edges are unique and stored in a set.
  // Nodes are not unique and stored in a map, i.e., node with "+"
      may happen several times.
  // This is only for building a graphical representation of a
     syntax tree - not necessary
  let rec genNE (i,nmap,eset) = function
    FNum r -> (i,Map.add i (genNodeStr i ((string)r)) nmap,eset)
                  // Add node with number
  | FVar x -> (i,Map.add i (genNodeStr i x) nmap,eset)
                           // Add node with variable
  | FAdd (e1,e2) -> let (i1,nmap1,eset1) = genNE (i+1,nmap,eset)
```

```
   e1          // Generate nodes and edges for e1 and e2
                let (i2,nmap2,eset2) = genNE (i1+1,nmap1,eset1
                   ) e2
                (i2+1,Map.add (i2+1) (genNodeStr (i2+1) "+")
                   nmap2,                        // Add node for
                    "+"
                 Set.add (genEdgeStr (i2+1) i2) (Set.add (
                    genEdgeStr (i2+1) i1) eset2))  // Add edge
                     for "+"->e1 and "+"->e2
| FMin (e1,e2) -> let (i1,nmap1,eset1) = genNE (i+1,nmap,eset)
   e1          // Generate nodes and edges for e1 and e2
                let (i2,nmap2,eset2) = genNE (i1+1,nmap1,eset1
                   ) e2
                (i2+1,Map.add (i2+1) (genNodeStr (i2+1) "-")
                   nmap2,                        // Add node for
                    "+"
                 Set.add (genEdgeStr (i2+1) i2) (Set.add (
                    genEdgeStr (i2+1) i1) eset2))  // Add edge
                     for "+"->e1 and "+"->e2

| FMult (e1,e2) -> let (i1,nmap1,eset1) = genNE (i+1,nmap,eset)
   e1         // Generate nodes and edges for e1 and e2
                 let (i2,nmap2,eset2) = genNE (i1+1,nmap1,
                    eset1) e2
                 (i2+1,Map.add (i2+1) (genNodeStr (i2+1) "*")
                    nmap2,                         // Add node
                    for "*"
                  Set.add (genEdgeStr (i2+1) i2) (Set.add (
                     genEdgeStr (i2+1) i1) eset2))  // Add
                     edge for "*"->e1 and "*"->e2
| FDiv (e1,e2) ->  let (i1,nmap1,eset1) = genNE (i+1,nmap,eset)
   e1         // Generate nodes and edges for e1 and e2
                 let (i2,nmap2,eset2) = genNE (i1+1,nmap1,
                    eset1) e2
                 (i2+1,Map.add (i2+1) (genNodeStr (i2+1) "/")
                    nmap2,                         // Add node
                    for "*"
                  Set.add (genEdgeStr (i2+1) i2) (Set.add (
                     genEdgeStr (i2+1) i1) eset2))  // Add
                     edge for "*"->e1 and "*"->e2

| FExponent (e1,ie) -> let (i1,nmap1,eset1) = genNE (i+1,nmap,
   eset) e1                                  // Generate nodes and
    edges for e1
                       let (i2,nmap2) = (i1+1,Map.add (i1+1) (
                          genNodeStr (i1+1) ((string)ie)) nmap1)
                            // Add node for integer (exponent)
                       (i2+1,Map.add (i2+1) (genNodeStr (i2+1) "
                          ^") nmap2,
```

```
                                // Add node for "^"
                               Set.add (genEdgeStr (i2+1) i2) (Set.add
                                  (genEdgeStr (i2+1) i1) eset1))
                                         // Add edges for "^"->e1 and
                                  "^"->ie
  | _ -> failwith "Not supported"

  let (_,nmap,eset) = genNE (0,Map.empty,Set.empty) ast  //
     Generate map for nodes and set for edges
  genDot (sprintf "%A\n" ast) (Map.fold (fun acc _ s -> acc + s) "
     " nmap) (Set.fold (fun acc s -> acc + s) "" eset)  //
     Generate big string with dot-code.
```

8.6.5 ExprToPoly.fsi

```
module ExprToPoly

type expr = ExprParse.expr
val subst: expr -> (string * expr) -> expr

type atom = ANum of float | AExponent of string * int
type atomGroup = atom list
type simpleExpr = SE of atomGroup list
val ppSimpleExpr: simpleExpr -> string
val exprToSimpleExpr: expr -> simpleExpr
val simplifyAtomGroup: atomGroup -> atomGroup
val simplifySimpleExpr: simpleExpr -> simpleExpr
val simplify: expr -> atom list list
val simpleExprToExpr: simpleExpr -> expr
val simplifyToExpr: expr -> expr
val getExprDegree: expr -> int -> int

type poly = P of Map<int,simpleExpr>
val getPolyMap: poly -> Map<int, simpleExpr>
val ppPoly: string -> poly -> string
val simpleExprToPoly: simpleExpr -> string -> poly
val exprToPoly: expr -> string -> poly
val polyToExpr: poly -> string -> expr
val isolateDiv: expr*expr -> expr
val cleanExpr: expr -> expr
val simplifySE: simpleExpr -> simpleExpr -> simpleExpr
val cleanDivMults: expr -> expr
val getExprValue: expr -> float
```

8.6.6 ExprToPoly.fs

```
module ExprToPoly

(*#load "ExprParse.fs"*)
```

```
type expr = ExprParse.expr
open ExprParse

let rec ppExpr = function
  | FNum c -> string(c)
  | FVar s -> s
  | FAdd(e1,e2) -> "(" + (ppExpr e1) + " + " + (ppExpr e2) + ")"
  | FMin(e1,e2) -> "(" + (ppExpr e1) + " - " + (ppExpr e2) + ")"
  | FMult(e1,e2) -> (ppExpr e1) + " * " + (ppExpr e2)
  | FDiv(e1,e2) -> (ppExpr e1) + " / " + (ppExpr e2)
  | FExponent(e,n) -> "(" + (ppExpr e) + ")^" + string(n)
  //| FRoot(e1,e2) -> "( Root " + (ppExpr e1) + "_" + (ppExpr e2)
     + ")"

let rec subst e (x,ex) =
  match e with
  | FInt i -> FInt i
  | FNum c -> FNum c
  | FVar s -> if(s=x) then ex else FVar s
  | FAdd(e1,e2) -> FAdd(subst e1 (x,ex), subst e2 (x,ex))
  | FMin(e1,e2) -> FMin(subst e1 (x,ex), subst e2 (x,ex))
  | FMult(e1,e2) -> FMult(subst e1 (x,ex), subst e2 (x,ex))
  | FDiv(e1,e2) -> FDiv(subst e1 (x,ex), subst e2 (x,ex))
  | FExponent(e1,n) -> FExponent(subst e1 (x,ex),n)
  | FRoot(e1,e2) -> FRoot(subst e1 (x,ex), e2)

type atom = ANum of float | AExponent of string * int
type atomGroup = atom list
type simpleExpr = SE of atomGroup list
let isSimpleExprEmpty (SE ags) = ags = [] || ags = [[]]

let simpleExprToExpr (SE ags) =
    let rec agsToExpr _ags =
        match _ags with
        |ANum f::[] -> FNum f
        |AExponent(s,i)::[] -> FExponent(FVar s, i)
        |ANum 0.0::_ -> FNum 0.0
        |ANum f::zs -> FMult(FNum f, agsToExpr zs)
        |AExponent(s,i)::zs -> FMult(FExponent(FVar s, i),
           agsToExpr zs)
        |[] -> FInt 0

    let rec atomGrpToExpr atGrp =
        match atGrp with
        |x::[] -> agsToExpr x
        |x::xs -> FAdd(agsToExpr x, atomGrpToExpr xs)
        |[] -> FNum 0.0
    atomGrpToExpr ags
```

```
//Checks whether expression contains a FDiv.
let rec containsDiv = function
    | FAdd(e1, e2) -> containsDiv e1 || containsDiv e2
    | FMin(e1, e2) -> containsDiv e1 || containsDiv e2
    | FMult(e1, e2) -> containsDiv e1 || containsDiv e2
    | FRoot(e1, _) -> containsDiv e1
    | FExponent(e1, _) -> containsDiv e1
    | FVar _ | FNum _ | FInt _ -> false
    | FDiv(_, _) -> true

//Checks whether expression contains a FDiv.
let rec containsRoot = function
    | FAdd(e1, e2) -> containsRoot e1 || containsRoot e2
    | FMin(e1, e2) -> containsRoot e1 || containsRoot e2
    | FMult(e1, e2) -> containsRoot e1 || containsRoot e2
    | FDiv(e1, e2) -> containsRoot e1 || containsRoot e2
    | FRoot(e1, _) -> true
    | FExponent(e1, _) -> containsRoot e1
    | FVar _ | FNum _ | FInt _ -> false

//Checks whether expression contains a FRoot.
//let rec containsRoot f = function
//    | FAdd(e1, e2) -> containsRoot e1 (fun e1 -> containsRoot e2
    (fun e2 -> e2 = FDiv (x,y))
//    | FMin(e1, e2) -> containsRoot e1 (fun e1 -> containsRoot e2
    (fun e2 -> e2 = FDiv (x,y))
//    | FDiv(e1, e2) -> containsRoot e1 (fun e1 -> containsRoot e2
    (fun e2 -> e2 = FDiv (x,y))
//    | FMult(e1, e2) -> containsRoot e1 (fun e1 -> containsRoot
   e2 (fun e2 -> e2 = FDiv (x,y))
//    | FExponent(e1, _) -> containsRoot e1 (fun e1 ->
   containsRoot e1 (fun e -> )
//    | FRoot(e1, _) -> true
//    | FVar _ | FNum _ | FInt _ -> false

let ppAtom = function
  | ANum c -> string(c)
  | AExponent(s,1) -> s
  | AExponent(s,n) -> s+"^"+(string(n))
let ppAtomGroup ag = String.concat "*" (List.map ppAtom ag)
let ppSimpleExpr (SE ags) = String.concat "+" (List.map
   ppAtomGroup ags)

let rec combine xss = function
  | [] -> []
  | ys::yss -> List.map ((@) ys) xss @ combine xss yss

//let rec getMultipliers (e:expr) c =
//    let checkExponent (ex:expr) =
```

```
//        match ex with
//        | FExponent(_, 0) -> c (FInt 1)
//        | FExponent(ex1, 1) -> c ex1
//        | FExponent(ex1, n) -> getMultipliers (FMult(ex1,
   FExponent(ex1, n-1))) c
//        | _ -> c ex
//
//    match e with
//    | FAdd(e1, e2) -> getMultipliers e1 (fun leftEx ->
   getMultipliers e2 (fun rightEx -> [leftEx;rightEx]))
//    | FMin(e1, e2) -> getMultipliers e1 (fun leftEx ->
   getMultipliers e2 (fun rightEx -> [leftEx;rightEx]))
//    | FDiv(e1, e2) -> getMultipliers e1 (fun leftEx ->
   getMultipliers e2 (fun rightEx -> [leftEx;rightEx]))
//    | FMult(e1, e2) -> checkExponent e1 @ checkExponent e2
//    | FExponent(e1, 0) -> c (FInt 1)
//    | FExponent(e1, 1) -> c e1
//    | FExponent(e1, n) -> c (FMult(e1, FExponent(e1, n-1)))
//    | FVar s -> c (FVar s)
//    | FNum n -> c (FNum n)
//    | FInt i -> c (FInt i)

let rec getMultipliers (e:expr) (acc: expr list)=
    let checkExponent (ex:expr) =
        match ex with
        | FExponent(_, 0) -> [FInt 1]
        | FExponent(ex1, 1) -> [ex1]
        | FExponent(ex1, n) -> getMultipliers (FMult(ex1,
           FExponent(ex1, n-1))) acc
        | _ -> [ex]

    match e with
    | FAdd(e1, e2) -> getMultipliers e1 acc @ getMultipliers e2
       acc
    | FMin(e1, e2) -> getMultipliers e1 acc @ getMultipliers e2
       acc
    | FDiv(e1, e2) -> getMultipliers e1 acc @ getMultipliers e2
       acc
    | FMult(e1, e2) -> checkExponent e1 @ checkExponent e2 @ acc
    | FExponent(e1, 0) -> acc
    | FExponent(e1, 1) -> e1 :: acc
    | FExponent(e1, n) -> getMultipliers (FMult(e1, FExponent(e1,
       n-1))) acc
    | _ -> acc

let rec removeMultiplier (ex:expr) (mul:expr) =
    match ex with
    | FAdd(e1, e2) -> FAdd(removeMultiplier ex e1,
       removeMultiplier ex e2)
```

```
    | FMin(e1, e2) -> FMin(removeMultiplier ex e1,
       removeMultiplier ex e2)
    | FMult(e1, e2) -> if mul = e1 then e2
                       elif mul = e2 then e1
                       else FMult(removeMultiplier ex e1,
                          removeMultiplier ex e2)
    | FExponent(e1, n) -> if mul = e1 then FExponent(e1, n-1)
                          else FExponent(e1, n)
    //| FRoot(e1, n) -> FRoot(removeMultiplier ex e1, n)
    | FDiv(e1, e2) -> FDiv(removeMultiplier ex e1,
       removeMultiplier ex e2)
    | FVar s -> FVar s
    | FNum n -> FNum n
    | FInt i -> FInt i

//let rec removeMultiplier (ex:expr) (mul:expr) c =
//    match ex with
//    | FAdd(e1, e2) -> removeMultiplier e1 mul (fun leftEx ->
   removeMultiplier e2 mul (fun rightEx -> FAdd(leftEx, rightEx)))
//    | FMin(e1, e2) -> removeMultiplier e1 mul (fun leftEx ->
   removeMultiplier e2 mul (fun rightEx -> FMin(leftEx, rightEx)))
//    | FMult(e1, e2) -> if mul = e1 then c e2
//                       elif mul = e2 then c e1
//                       else removeMultiplier e1 mul (fun leftEx
   -> removeMultiplier e2 mul (fun rightEx -> FMult(leftEx,
   rightEx)))
//    | FExponent(e1, n) -> if mul = e1 then c (FExponent(e1, n-1)
   )
//                          else c (FExponent(e1, n))
//    //| FRoot(e1, n) -> FRoot(removeMultiplier ex e1, n)
//    | FDiv(e1, e2) -> removeMultiplier e1 mul (fun leftEx ->
   removeMultiplier e2 mul (fun rightEx -> FDiv(leftEx, rightEx)))
//    | FVar s -> c (FVar s)
//    | FNum n -> c (FNum n)
//    | FInt i -> c (FInt i)

//True of param e is a member of param l1. else false.
let rec mem (l1: expr list) (e: expr) =
    match l1 with
    | x::xs -> if x = e
               then true
               else mem xs e
    | _ -> false

//Tailrecursive intersection of lists
let rec listIntersect (l1: expr list) (l2: expr list) (acc:expr
   list) =
    match l1 with
    | x::xs -> match mem l2 x with
```

```
                | true -> listIntersect xs l2 (x::acc)
                | false -> listIntersect xs l2 acc
    | [] -> acc

let rec getExprValue = function
    | FNum n -> n
    | FInt i -> float(i)
    | FAdd(e1, e2) -> getExprValue e1 + getExprValue e2
    | FMin(e1, e2) -> getExprValue e1 - getExprValue e2
    | FMult(e1, e2) -> getExprValue e1 * getExprValue e2
    | FDiv(e1,e2) -> getExprValue e1 / getExprValue e2
    | FExponent(e1, n) -> System.Math.Pow(getExprValue e1, float(n
       ))
    | FVar s -> 1.0

//Removes identical multiplication from division. (4*x*x / 4*x ->
   x)
let rec cleanDivMults (div:expr) =
    match div with
    | FDiv(e1, e2) -> let upper = getMultipliers e1 []
                      let lower = getMultipliers e2 []
                      let intersect = listIntersect upper lower []

                      let cleanLower = List.fold (fun acc ex -> (
                         removeMultiplier acc ex)) e2 intersect
                      let cleanUpper = List.fold (fun acc ex -> (
                         removeMultiplier acc ex)) e1 intersect
                      FDiv(cleanUpper, cleanLower)
    | FAdd(e1, e2) -> FAdd(cleanDivMults e1, cleanDivMults e2)
    | FMin(e1, e2) -> FMin(cleanDivMults e1, cleanDivMults e2)
    | FMult(e1, e2) -> FMult(cleanDivMults e1, cleanDivMults e2)
    | FExponent(e1, n) -> FExponent(cleanDivMults e1, n)
    | FRoot(e1, n) -> FRoot(cleanDivMults e1, n)
    | ex -> failwith "not a div"

let rec multDiv (e1:expr) (e2:expr) =
    match e1 with
    | FDiv(ex1, ex2) -> FDiv(FMult(ex1, e2), ex2)
    | FAdd(ex1, ex2) -> FAdd(multDiv ex1 e2, multDiv ex2 e2)
    | FMin(ex1, ex2) -> FMin(multDiv ex1 e2, multDiv ex2 e2)
    | FMult(ex1, ex2) ->
                        match containsDiv ex1 with
                             | true -> multDiv (multDiv ex1 ex2)
                                 e2
                            | false -> multDiv (multDiv ex2 ex1)
                                 e2
    | FExponent(ex1, 0) -> FInt 1
    | FExponent(ex1, 1) -> ex1
    | FExponent(ex1, n) -> multDiv (FMult(FExponent(ex1, n-1), ex1
```

```
        )) e2
    | _ -> FMult(e1, e2)

let cleanRootExp (e:expr) (n:int) =
    match e with
      | FRoot(e1, n1) when n = n1 -> e1
      | FAdd(e1, e2) -> FAdd(FAdd(FExponent(e1, n), FExponent(e1,
         n)), FMult(FInt 2, FMult(e1, e2)))
      | FMin(e1, e2) -> FMin(FAdd(FExponent(e1, n), FExponent(e1,
         n)), FMult(FInt 2, FMult(e1, e2)))
      | FMult(e1, e2) -> FMult(FExponent(e1, n), FExponent(e2, n))
      | ex -> ex

//Takes expr = 0 and isolates division expressions in order to
   express them
//without using division.
let rec isolateDiv ((leftEx:expr), rightEx:expr) =
    match (leftEx, rightEx) with
    | (FAdd(e1, e2), resEx)     ->
        match (containsDiv e1, containsDiv e2) with
            | (true, false) -> isolateDiv (e1, FMin(resEx, e2))
            | (false, true) -> isolateDiv (e2, FMin(resEx, e1))
            | (true, true) -> isolateDiv (e1, FMin(resEx, e2))
            //isolateDiv (FAdd((isolateDiv(e1, resEx)), (
               isolateDiv(e2, resEx))), resEx)
            | (false, false) -> failwith "No division found."
    | (FMin(e1, e2), resEx)     ->
        match (containsDiv e1, containsDiv e2) with
            | (true, false) -> isolateDiv (e1, FAdd(resEx, e2))
            | (false, true) -> isolateDiv (FMult(FInt -1, e2),
               FMin(resEx, e1))
            | (true, true) -> isolateDiv (e1, FAdd(resEx, e2))
            //isolateDiv (FMin((isolateDiv(e1, resEx)), (
               isolateDiv(e2, resEx))), resEx)
            | (false, false) -> failwith "No division found."
    | (FMult(e1, e2), resEx)    ->
        match (containsDiv e1, containsDiv e2) with
            | (true, false) -> isolateDiv (e1, FDiv(resEx, e2))
            //((cleanDivMults (multDiv e1 e2)), resEx)
            | (false, true) -> isolateDiv (e2, FDiv(resEx, e1))
            //((cleanDivMults (multDiv e2 e1)), resEx)
            | (true, true) -> isolateDiv (e1, FDiv(resEx, e2))
            //((cleanDivMults (multDiv e1 e2)), resEx)
            //isolateDiv (FMult((isolateDiv(e1, resEx)), (
               isolateDiv(e2, resEx))), resEx)
            | (false, false) -> failwith "No division found."
    | (FExponent(e1, n), resEx) -> match (containsDiv e1) with
            | true -> isolateDiv (e1, FRoot(resEx, n))
            | false -> failwith "No division found."
```

```
    | (FRoot(e1, n), resEx)      -> match (containsDiv e1) with
          | true -> isolateDiv (e1, FExponent(resEx, n))
                    | false -> failwith "No division found."
    | (FDiv(e1, e2), resEx)      ->
      match (containsDiv e1, containsDiv e2) with
          | (true, false) -> FMin(FMult(resEx, e2), e1)
          | (false, true) -> FMin(FMult(resEx, e2), e1)
          | (true, true) -> FMin(FMult(resEx, e2), e1)
          //isolateDiv (FDiv((isolateDiv(e1, resEx)), (
             isolateDiv(e2, resEx))), resEx)
          | (false, false) -> match cleanDivMults (FDiv(e1, e2))
              with
          | FDiv(ex1, ex2) -> match (ex1, ex2) with //supports
             simple division.
          |(FInt i, FInt i2) -> FInt (i/i2)
          |(FInt i, FNum n) -> FNum (float(i)/n)
          |(FNum n, FInt i) -> FNum (n/float(i))
          |(FNum n, FNum n2) -> FNum (n/n2)
          |_ -> FMin(FMult(resEx, ex2), ex1)
          | _ -> failwith "cleanDivMults did not return a div."
    | (_, resEx) -> resEx

//Takes expr = 0 and isolates the root in order to express it
   without
//using roots.
let rec isolateRoot ((leftEx:expr), rightEx:expr) =
    match (leftEx, rightEx) with
    | (FAdd(e1, e2), resEx)      ->
       match (containsRoot e1, containsRoot e2) with
          | (true, false) -> isolateRoot (e1, FMin(resEx, e2))
          | (false, true) -> isolateRoot (e2, FMin(resEx, e1))
          | (true, true)  -> isolateRoot (FAdd((isolateRoot(e1,
             resEx)), (isolateRoot(e2, resEx))), resEx)
          | (false, false) -> FMin(FMin(resEx, e1), e2)
    | (FMin(e1, e2), resEx)      ->
      match (containsRoot e1, containsRoot e2) with
          | (true, false) -> isolateRoot (e1, FAdd(resEx, e2))
          | (false, true) -> isolateRoot (e2, FMin(resEx, e1))
          | (true, true)  -> isolateRoot (FMin((isolateRoot(e1,
             resEx)), (isolateRoot(e2, resEx))), resEx)
          | (false, false) -> FMin(FAdd(resEx, e2), e1)
    | (FMult(e1, e2), resEx)     ->
      match (containsRoot e1, containsRoot e2) with
          | (true, false) -> isolateRoot (e1, FDiv(resEx, e2))
          | (false, true) -> isolateRoot (e2, FDiv(resEx, e1))
          | (true, true) -> isolateRoot (FMult((isolateRoot(e1,
             resEx)), (isolateRoot(e2, resEx))), resEx)
          | (false, false) -> failwith "No root found"
    | (FExponent(e1, n), resEx) ->
```

```
        match (containsRoot e1) with
            | true -> isolateRoot (e1, resEx) //(cleanRootExp e1 n
               , resEx)
            | false -> failwith "No root found"
    | (FRoot(e1, n), resEx)  ->
        match (containsRoot e1) with
            | true -> isolateRoot (e1, FExponent(resEx, n))
            | false -> FMin(FExponent(resEx, n), e1)
    | (FDiv(e1, e2), resEx)     ->
        match (containsRoot e1, containsRoot e2) with
            | (true, false) -> isolateRoot (e1, FMult(resEx, e2))
            | (false, true) -> isolateRoot (e2, FMult(resEx, FDiv(
               FInt 1, e1)))
            | (true, true) -> isolateRoot (FDiv((isolateRoot(e1,
               resEx)), (isolateRoot(e2, resEx))), resEx)
            | (false, false) -> failwith "No root found"
    | (_, resEx) -> resEx

////Checks whether param e contains a division or root.
////If it does, the expr is rewritten expressing it without using
   the root or the division.
////Else the param e is returned.
let rec cleanExpr (e:expr) =
    match (containsDiv e, containsRoot e) with
    | (true, false) -> isolateDiv (e, (FInt 0)) |> cleanExpr
    | (false, true) -> isolateRoot (e, (FInt 0)) |> cleanExpr
    | (true, true) -> isolateRoot ((isolateDiv (e, (FInt 0))),
       FInt 0) |> cleanExpr
    | (false, false) -> e

let rec simplify (ex:expr) =
    match cleanExpr ex with
    | FNum c -> [[ANum c]]
    | FInt c -> [[ANum (float(c))]]
    | FVar s -> [[AExponent(s,1)]]
    | FAdd(e1,e2) -> simplify e1 @ simplify e2
    | FMin(e1,e2) -> simplify e1 @ simplify (FMult(FNum -1.0, e2))
    | FMult(e1,e2) -> (combine (simplify e1) (simplify e2))
    | FExponent(e1, 0) -> [[ANum 1.0]]
    | FExponent(e1, 1) -> simplify e1
    | FExponent(e1, n) -> simplify (FMult(e1,FExponent(e1,n-1)))
    | FRoot(_, _) -> failwith "checkExpr did not catch FRoot."
    | FDiv(_, _) -> failwith "checkExpr did not catch FDiv."

//Returns the degree of param al. Assuming there is only one
   unknown.
let rec simExDegree (al: atom list) (acc:int)=
    match al with
    | AExponent(s, i)::xs -> simExDegree xs (acc+i)
```

```
    | ANum c::xs -> simExDegree xs acc
    | _ -> acc

//Returns the product of the constants of param al.
let rec getSimExCons (al:atom list) (acc:float) =
    match al with
    | AExponent(s, i)::xs -> getSimExCons xs acc
    | ANum c::xs -> getSimExCons xs (acc*c)
    | _ -> acc

//Compares two atom lists, returns 0 if they are same degree and
   the sum of
//their constans is zero. else return param al2.
let simplifyAgs (al1: atom list) (al2: atom list) =
    match simExDegree al1 0 = simExDegree al2 0 with
    | true  -> match (getSimExCons al1 1.0) + (getSimExCons al2
       1.0) with
               | 0.0 -> [ANum 0.0]
               | _   -> al2 //[AExponent("x", (simExDegree al1 0))
                  ;(ANum ((getSimExCons al1 1.0) + (getSimExCons
                  al2 1.0)))]
    | false -> al2

//Compares param al to the atom lists in ags.
let rec collapseSE (SE ags) (al:atom list) =
    match ags with
    |x::xs -> simplifyAgs al x :: collapseSE (SE(xs)) al
    |_     -> []

//Cleans min of exponents. Ie. 2x - 2x should be 0.
let rec simplifySE (SE ags) (SE ags1) =
    match ags1 with
    |x::xs -> simplifySE (SE(collapseSE (SE(ags)) x)) (SE(xs))
    |_ -> SE (ags)

let simplifyAtomGroup ag =
        let exponents =
            List.fold (fun map x ->
             match x with
                |AExponent(s,i) ->
                    let v = Map.tryFind s map
                        if  v <> None  then Map.add s (i+v.Value)
                           map
                        else Map.add s i map
                |_ -> map)Map.empty ag

        let product = List.fold (fun prod x ->
            match x with
                |ANum i -> prod*i
```

```
                |_ -> prod)1.0 ag
        let list = Map.foldBack (fun key v list -> AExponent(key,
           v) :: list) exponents []
        if product <> 1.0
            then (ANum product) :: list
        else if list = []
            then [ANum product]
        else list

let simplifySimpleExpr (SE ags) =
  let ags' = List.map simplifyAtomGroup ags
  // Add atom groups with only constants together
  let addedGroups =
    List.fold (fun sum x ->
        match x with
            |[ANum i] -> sum+i
            |_ -> sum)
        0.0 ags'
  //Last task is to group similar atomGroups into one group.
  let exponents =
    List.fold (fun map x ->
        match x with
            | [ANum i] -> map
            | c -> let v = Map.tryFind c map
                if v <> None then Map.add c (v.Value+1.0) map
                else Map.add c 1.0 map) Map.empty ags'
  let list = Map.foldBack (fun key v l -> if v <> 1.0 then ((ANum
     v :: key) :: l) else  key :: l) exponents []
  let ags'' = if addedGroups <> 0.0 then [ANum addedGroups] ::
     list else list
  simplifySE (SE(ags'')) (SE(ags''))

let exprToSimpleExpr e = simplifySimpleExpr (SE (simplify e))

type poly = P of Map<int,simpleExpr>

let getPolyMap (p:poly) =
    match p with
    P map -> map

let ppPoly v (P p) =
  let pp (d,ags) =
    let prefix = if d=0 then "" else ppAtom (AExponent(v,d))
    let postfix = if isSimpleExprEmpty ags then "" else "(" + (
       ppSimpleExpr ags) + ")"
    prefix + postfix
  String.concat "+" (List.map pp (Map.toList p))

(* Collect atom groups into groups with respect to one variable v
```

```
    *)
let splitAG v m = function
  | [] -> m
  | ag ->
    let eqV = function AExponent(v',_) -> v = v' | _ -> false
    let addMap d ag m =
        match Map.tryFind(d) m with
            | None -> m.Add(d,SE([ag]))
            | Some (SE(s)) -> m.Add(d,SE(ag::s))

    match List.tryFind eqV ag with
      | Some (AExponent(_,d)) ->
        let ag' = List.filter (not << eqV) ag
        addMap d ag' m
      | Some _ -> failwith "splitAG: Must never come here! - ANum
         will not match eqV"
      | None -> addMap 0 ag m

let simpleExprToPoly (SE ags) v =
  P (List.fold (splitAG v) Map.empty ags)

let exprToPoly e v = (exprToSimpleExpr >> simplifySimpleExpr >>
   simpleExprToPoly) e v

let highestVal (i1:int) (i2:int) =
    match i1 > i2 with
    | true  -> i1
    | false -> i2

let rec getExprDegree (e:expr) (acc:int) =
    match e with
    | FAdd(e1, e2) -> highestVal (getExprDegree e1 acc) (
       getExprDegree e2 acc)
    | FMin(e1, e2) -> highestVal (getExprDegree e1 acc) (
       getExprDegree e2 acc)
    | FMult(e1, e2) -> highestVal (getExprDegree e1 acc) (
       getExprDegree e2 acc)
    | FExponent(e1, n) -> highestVal (getExprDegree e1 acc) n
    | _ -> acc

let simplifyToExpr (e:expr) = (exprToSimpleExpr >>
   simplifySimpleExpr >> simpleExprToExpr) e

let polyToExpr (polynomial:poly) (s:string) = (ppPoly s polynomial
   ) |> scan |> insertMult |> parse
```

8.6.7 Point.fsi

```
module Point
```

```
type Vector = Vector.Vector

type Point
val zero : Point
val mkPoint : float -> float -> float -> Point
val getX : Point -> float
val getY : Point -> float
val getZ : Point -> float
val getCoord : Point -> float * float * float
val move : Point -> Vector -> Point
val distance : Point -> Point -> Vector
val direction : Point -> Point -> Vector
```

8.6.8 Point.fs

```
module Point
open Vector;
open System;
type Vector = Vector.Vector
type Point =
  | P of float * float * float
  override p.ToString() =
    match p with
      P(x,y,z) -> "("+x.ToString()+","+y.ToString()+","+z.ToString
         ()+")"

let zero = P(0.0,0.0,0.0)
let mkPoint x y z = P(x,y,z)
let getX (P(x,_,_)) = x
let getY (P(_,y,_)) = y
let getZ (P(_,_,z)) = z
let getCoord (P(x,y,z)) = (x,y,z)
let move (P(x,y,z)) (v:Vector) = P((x+Vector.getX(v)),(y+Vector.
   getY(v)),(z+Vector.getZ(v)))


let distance (P(px,py,pz)) (P(qx,qy,qz)) = (Vector.mkVector (qx-px
   ) (qy-py) (qz-pz))
let direction p q = Vector.normalise (distance p q)
```

8.6.9 FloatHelper.fs

```
namespace RayTracer
(*
 The constant in this module is used for computations using float-
    values that might im
 precicion-troubles.
 We consider values the same if they differ by a small amount
    instead of exactly zero.
```

```
    The constant in this module will be used to handle floats-near
       zero when solving quadratic equations.
    *)

    module FloatHelper =

        // Constant
        let ZeroThreshold = 0.0000000001
```

8.6.10 SceneAssets.fsi

```
    namespace RayTracer

    module SceneAssets =
        type Point = Point.Point
        type Vector = Vector.Vector
        type Ray = {origin:Point; direction:Vector}
        type Vertex = {point:Point; normal:Vector; u:float; v:float}

        [<Sealed>]
        type BoundingBox =
            member lower : Point
            member higher : Point

        type Face = F of int * int * int

        [<Sealed>]
        type Colour =
            static member ( * ) : float * Colour -> Colour
            static member ( * ) : Colour * Colour -> Colour
            static member ( + ) : Colour * Colour -> Colour


        type LightType =
            Light of Point*Colour*float |AmbientLight of Colour*float
        type Texture
        type Material

        // Interface used by all shapes with common functions
        type IShape =
            abstract member HitFunc: Ray -> float Option*Point Option*
               Vector Option*(unit -> Material)
            abstract member IsInside: Point -> bool
            abstract member HitBbox: Ray -> bool
            abstract member Bbox: BoundingBox


        val mkRay : origin:Point -> direction:Vector -> Ray
        val getOriginPoint : ray:Ray -> Point
        val getDirection : ray:Ray -> Vector
```

```
    val mkBbox : Point -> Point -> BoundingBox
    val mkMaterial : colour : Colour -> float -> Material
    val black : unit -> Material
    val mkTexture : (float -> float -> Material) -> Texture

    val mkMatTexture : Material -> Texture

    val mkAmbientLight : colour : Colour -> intensity : float ->
       LightType

    val mkLight : position : Point -> colour : Colour -> intensity
        : float -> LightType

    val mkColour : r:float -> g:float -> b:float -> Colour

    val fromColor : c : System.Drawing.Color -> Colour

    val loadTexture : string -> Texture

    val getTextureColour: Texture -> u:float*v:float -> Material

    val getMaterialColour: Material -> Colour;

    val getMaterialReflexivity: Material -> float;

    val getColor : Colour -> System.Drawing.Color
    val getLightColour : LightType -> Colour
    val getLightIntensity : LightType -> float
    val getLightPosition : LightType -> Point

    //A vertex that also contains u and v coordinates and a normal
        vector used for triangle mesh
    val mkVertex : Point -> Vector -> float -> float -> Vertex
```

8.6.11 SceneAssets.fs

```
namespace RayTracer
open System.Drawing
open System

module SceneAssets =
    type Point = Point.Point
    type Vector = Vector.Vector
    type Colour = {red:float;green:float;blue:float}
    type LightType =
        Light of Point*Colour*float |AmbientLight of Colour*float
    type Material = {color:Colour;reflexivity:float}
    type Texture = {textureFunction:(float -> float -> Material)}
    type Ray = {origin:Point; direction:Vector}
    type BoundingBox = {L : Point; H : Point}
```

```
type BoundingBox with
    member this.lower = this.L
    member this.higher = this.H

// Shape interface implemented by all shapes with common
   features
type IShape =
    abstract member HitFunc: Ray -> float Option*Point Option*
       Vector Option*(unit -> Material)
    abstract member HitBbox: Ray -> bool
    abstract member Bbox: BoundingBox
    abstract member IsInside: Point -> bool

let mkBbox (lower : Point) (higher : Point) : BoundingBox =
    {L = lower; H = higher}

type Vertex = {point:Point; normal:Vector; u:float; v:float}
type Face = F of int * int * int


//Generates a ray based on an origin point and a direction
let mkRay p dir = {origin=p;direction=dir}
let getOriginPoint  (ray:Ray) = ray.origin
let getDirection (ray:Ray) = ray.direction

let mkTexture func = {textureFunction = func}

let mkMaterial col r = {color = col; reflexivity = r;}

//Gamma correction for colors
let Gamma = 0.95

let mkMatTexture mat = {textureFunction = (fun _ _ -> mat)}

let mkLight center color intensity = Light(center,color,
   intensity)

let mkColour  (r:float) (g:float) (b:float) =
    let gamma = Gamma
    {red = Math.Pow(r,gamma); green = Math.Pow(g,gamma); blue
       = Math.Pow(b,gamma)}

//Help function for points that need to be colored black, to
   make things easier
let black () = mkMaterial (mkColour 0.0 0.0 0.0) 0.0

let fromColor (c:System.Drawing.Color) = mkColour ((float) c.R
   /255.0) ((float) c.G/255.0) ((float) c.B/255.0)
```

```
let mkAmbientLight colour intensity = AmbientLight(colour,
   intensity)

let getTextureColour (t:Texture) (u:float,v:float) = t.
   textureFunction u v

let getColor (c:Colour) =
    let gamma = 1.0/Gamma
    Color.FromArgb(min ((int)(Math.Pow(c.red,gamma)*255.0))
       255,min ((int)(Math.Pow(c.green,gamma)*255.0)) 255, min
        ((int)(Math.Pow(c.blue,gamma)*255.0)) 255)

let getMaterialColour (m:Material) = m.color;

let getMaterialReflexivity (m:Material) = m.reflexivity;

let getLightColour (l:LightType) =
    match l with
    | AmbientLight(colour,_) -> colour
    | Light(_,colour,_) -> colour

let getLightIntensity (l:LightType) =
    match l with
    | AmbientLight(_,intensity) -> intensity
    | Light(_,_,intensity) -> intensity

let getLightPosition (l:LightType) =
    match l with
    | Light(position,_,_) -> position
    | _ -> failwith "This light has no position!"

type Colour with
    static member ( * ) (s:float, c:Colour) = (mkColour (c.red
       *s) (c.green*s) (c.blue*s))
    static member ( * ) (c1:Colour,c2:Colour) = (mkColour (c1.
       red*c2.red) (c1.green*c2.green) (c1.blue*c2.blue))
    static member ( + ) (c1:Colour,c2:Colour) = (mkColour (c1.
       red+c2.red) (c1.green+c2.green) (c1.blue+c2.blue))

let loadTexture (file : string): Texture =
    let img = new Bitmap(file)
    let widthf = float (img.Width - 1)
    let heightf = float (img.Height - 1)
    let texture (u : float) (v : float) =
        lock img (fun () ->
        let color = img.GetPixel
                        (int (widthf * u), int (heightf * (1.0
                            - v)))
```

```
            mkMaterial (fromColor color) 0.0
            )
        mkTexture(texture)

    //Vertex that also contains u and v coordinates
    let mkVertex (p:Point) (n:Vector) (u:float) (v:float) = {point
       =p;normal=n;u=u;v=v}
```

8.6.12 BoundingBox.fsi

```
namespace RayTracer
module BoundingBox =
    type Point = SceneAssets.Point
    type Vector = Vector.Vector
    type BoundingBox = SceneAssets.BoundingBox
    type Ray = SceneAssets.Ray

    val computeTriangleBbox : Point -> Point -> Point ->
       BoundingBox
    val computeSphereBbox : float -> BoundingBox
    val computeBoxBbox : Point -> Point -> BoundingBox
    val computeDiscBbox :  float -> BoundingBox
    val computeCylinderBbox  :  float -> float -> BoundingBox
    val computeRectangleBbox : float -> float -> BoundingBox
    val getBboxIntersection: bbox:BoundingBox -> ray:Ray -> bool*
       float*float
    val checkBboxIntersection: bbox:BoundingBox -> ray:Ray -> bool
    val computeSceneBbox: BoundingBox list -> BoundingBox
    val computeSceneBboxArr: BoundingBox[] -> BoundingBox
    val findLowestPoint : Point -> Point -> Point
    val findHighestPoint : Point -> Point -> Point
```

8.6.13 BoundingBox.fs

```
namespace RayTracer
open System
open Vector.Generic //Should this be generic?
module BoundingBox =
    type Point = SceneAssets.Point
    type Vector = Vector.Vector
    type BoundingBox = SceneAssets.BoundingBox
    type Ray = SceneAssets.Ray


    //======================Help functions=====================

    //Folds over a list of floats, continuously compares them and
       finds the minimum point starting at infinity
    let findMin (list : float list) = List.fold (fun acc f -> min
       acc f) ((float)Single.PositiveInfinity) list
```

```
//Folds over a list of floats, continuously compares the
   values and finds the maximum starting at 0
let findMax (list : float list) = List.fold (fun acc f -> max
   acc f) ((float) Single.NegativeInfinity) list


//epsilon represents a very small float point value
let epsilon = FloatHelper.ZeroThreshold


//======================Bounding box functions
   ======================

//--Triangle bounding box--
//This help function calculates the upper x,y,z point triangle
let findHigher (a : Point) (b : Point) (c : Point) =
    let Hx = findMax [Point.getX a; Point.getX b; Point.getX c
       ] + epsilon
    let Hy = findMax [Point.getY a; Point.getY b; Point.getY c
       ] + epsilon
    let Hz = findMax [Point.getZ a; Point.getZ b; Point.getZ c
       ] + epsilon
    Point.mkPoint Hx Hy Hz //Returns the point

//This help function calculates the lower x,y,z point of a
   triangle
let findLower (a : Point) (b : Point) (c : Point) =
    let Lx = findMin [Point.getX a; Point.getX b; Point.getX c
       ] - epsilon
    let Ly = findMin [Point.getY a; Point.getY b; Point.getY c
       ] - epsilon
    let Lz = findMin [Point.getZ a; Point.getZ b; Point.getZ c
       ] - epsilon
    Point.mkPoint Lx Ly Lz //Returns the point

//This function calculates the bounding box for a triangle
let computeTriangleBbox a b c : BoundingBox =
    let higher = findHigher a b c
    let lower = findLower a b c
    SceneAssets.mkBbox lower higher

//This function calculates the bounding box for a sphere
let computeSphereBbox (radius : float) : BoundingBox =
    let center = Point.mkPoint 0.0 0.0 0.0
    let lower = Point.mkPoint (Point.getX center - radius -
       epsilon) (Point.getY center - radius - epsilon) (Point.
       getZ center - radius - epsilon)
    let higher = Point.mkPoint (Point.getX center + radius +
```

```
        epsilon) (Point.getY center + radius + epsilon) (Point.
        getZ center + radius + epsilon)
    SceneAssets.mkBbox lower higher

//This function calculates the bounding box for a box
let computeBoxBbox (lower : Point) (higher : Point) =
    let lower = Point.mkPoint (Point.getX lower - epsilon) (
        Point.getY lower - epsilon) (Point.getZ lower - epsilon
        )
    let higher = Point.mkPoint (Point.getX higher + epsilon) (
        Point.getY higher + epsilon) (Point.getZ higher +
        epsilon)
    SceneAssets.mkBbox lower higher

//This function calculates the bounding box for a disc
let computeDiscBbox  (r : float) : BoundingBox =
    let c = Point.mkPoint 0.0 0.0 0.0
    let lower = Point.mkPoint (Point.getX c - r - epsilon) (
        Point.getY c - r - epsilon) (Point.getZ c - epsilon)
    let higher = Point.mkPoint (Point.getX c + r + epsilon) (
        Point.getY c + r + epsilon) (Point.getZ c + epsilon)
    SceneAssets.mkBbox lower higher

//This function calculates the bounding box for a cylinder
let computeCylinderBbox (r : float) (h : float) : BoundingBox
   =
    let c = Point.mkPoint 0.0 0.0 0.0
    let lower = Point.mkPoint (Point.getX c - r - epsilon) (
        Point.getY c - h/2.0 - epsilon) (Point.getZ c - r -
        epsilon)
    let higher = Point.mkPoint (Point.getX c + r + epsilon) (
        Point.getY c + h/2.0 + epsilon) (Point.getZ c + r +
        epsilon)
    SceneAssets.mkBbox lower higher


//This function calculates the bounding box for a rectangle
let computeRectangleBbox (width : float) (height : float) :
   BoundingBox =
    let bottomLeft = Point.mkPoint 0.0 0.0 0.0
    let lower = Point.mkPoint (Point.getX bottomLeft - epsilon
        ) (Point.getY bottomLeft - epsilon) (Point.getZ
        bottomLeft - epsilon)
    let higher = Point.mkPoint (Point.getX bottomLeft + width
        + epsilon) (Point.getY bottomLeft + height + epsilon) (
        Point.getZ bottomLeft + epsilon)
    SceneAssets.mkBbox lower higher
```

```
//Finds the lowest coordinates of two points
let findLowestPoint (p1 : Point) (p2 : Point) =
    let x = if Point.getX p1 > Point.getX p2 then Point.getX
       p2 else Point.getX p1
    let y = if Point.getY p1 > Point.getY p2 then Point.getY
       p2 else Point.getY p1
    let z = if Point.getZ p1 > Point.getZ p2 then Point.getZ
       p2 else Point.getZ p1
    Point.mkPoint x y z

//Finds the highest coordinates of two points
let findHighestPoint (p1 : Point) (p2 : Point) =
    let x = if Point.getX p1 > Point.getX p2 then Point.getX
       p1 else Point.getX p2
    let y = if Point.getY p1 > Point.getY p2 then Point.getY
       p1 else Point.getY p2
    let z = if Point.getZ p1 > Point.getZ p2 then Point.getZ
       p1 else Point.getZ p2
    Point.mkPoint x y z

//This function takes a list of BoundingBoxes and based on
   their coords,
//a bounding box of the entire scene is calculated.
let computeSceneBbox  (bboxes : BoundingBox list) :
   BoundingBox =
    let startPoint = (Point.mkPoint 0.0 0.0 0.0)
    let inf = (float) Single.PositiveInfinity
    let lowerPoint = List.fold (fun point (bbox:BoundingBox)
       -> findLowestPoint point bbox.lower) (Point.mkPoint inf
        inf inf) bboxes
    let higherPoint = List.fold (fun point (bbox:BoundingBox)
       -> findHighestPoint point bbox.higher) (Point.mkPoint
       0.0 0.0 0.0) bboxes
    SceneAssets.mkBbox lowerPoint higherPoint

let computeSceneBboxArr  (bboxes : BoundingBox[]) :
   BoundingBox =
    let startPoint = (Point.mkPoint 0.0 0.0 0.0)
    let inf = (float) Single.PositiveInfinity
    let lowerPoint = Array.fold (fun point (bbox:BoundingBox)
       -> findLowestPoint point bbox.lower) (Point.mkPoint inf
        inf inf) bboxes
    let higherPoint = Array.fold (fun point (bbox:BoundingBox)
        -> findHighestPoint point bbox.higher) (Point.mkPoint
       0.0 0.0 0.0) bboxes
    SceneAssets.mkBbox lowerPoint higherPoint

  //---- Ray intersection with bounding box ----
```

```
//These two function will find the min and max  value based on
    3 given values
let findMaxCoordinate (x : float) (y : float) (z : float) =
   Math.Max(Math.Max (x, y),z)
let findMinCoordinate (x : float) (y : float) (z : float) =
   Math.Min(Math.Min (x, y),z)

//This function is a generic function that is used to
   calculate tx, ty and tz values for intersection with bbox
   computation
let calculateTValue o H L d =
    if(d >= 0.0) then //Based on the direction of a ray.
        let t = (L - o)/d
        let t' = (H - o)/d
        (t,t')
    else
        let t = (H - o)/d
        let t' = (L - o)/d
        (t,t')

let getBboxIntersection (bbox : BoundingBox) (ray : Ray) =
    //Defining all the variables
    let ox = Point.getX ray.origin // o is the origin point of
        a ray
    let oy = Point.getY ray.origin
    let oz = Point.getZ ray.origin

    let (dx,dy,dz) = Vector.getCoord (ray.direction) // d is
       the direction of the ray

    let Hx = Point.getX bbox.higher //H is the higher point of
        a bounding box
    let Hy = Point.getY bbox.higher
    let Hz = Point.getZ bbox.higher

    let Lx = Point.getX bbox.lower //L is the lower point of a
        bounding box
    let Ly = Point.getY bbox.lower
    let Lz = Point.getZ bbox.lower

    //Calculates t values (used to check if there is a hit)
    let txvalues = calculateTValue ox Hx Lx dx
    let tx = fst(txvalues)
    let tx' = snd(txvalues)

    let tyvalues = calculateTValue oy Hy Ly dy
    let ty = fst(tyvalues)
    let ty' = snd(tyvalues)
```

```
            let tzvalues = calculateTValue oz Hz Lz dz
            let tz = fst(tzvalues)
            let tz' = snd(tzvalues)

            //Getting the smalles and largest t value
            let t = findMaxCoordinate tx ty tz
            let t' = findMinCoordinate tx' ty' tz'

            //Compare T values to check if there is a hit with the
               given bounding box
            let isHit = t < t' && t'> 0.0
            (isHit,t,t')

        //This function will check if a ray intersects with a given
           bounding box
        let checkBboxIntersection (bbox : BoundingBox) (ray : Ray) =
            let (isHit, t, t') = getBboxIntersection bbox ray
            isHit
```

8.6.14 Transformation.fsi

```
namespace RayTracer

module Transformation =

    type Shape = SceneAssets.IShape

    type matrix = float[,]
    type Transformation = matrix*matrix
    type Ray = SceneAssets.Ray

    val translate : x:float -> y:float -> z:float ->
       Transformation

    val rotateX : angle:float -> Transformation
    val rotateY : angle:float -> Transformation
    val rotateZ : angle:float -> Transformation

    val sheareXY : distance:float -> Transformation
    val sheareXZ : distance:float -> Transformation
    val sheareYX : distance:float -> Transformation
    val sheareYZ : distance:float -> Transformation
    val sheareZX : distance:float -> Transformation
    val sheareZY : distance:float -> Transformation

    val scale : x:float -> y:float -> z:float -> Transformation

    val mirrorX : Transformation
    val mirrorY : Transformation
    val mirrorZ : Transformation
```

```
    val transform : Shape -> Transformation -> Shape
    val mergeTransformations : Transformation list ->
       Transformation
    val transformBbox : BoundingBox.BoundingBox -> Transformation
       -> BoundingBox.BoundingBox
```

8.6.15 Transformation.fs

```
namespace RayTracer
(*
    This module is used to perform affine transformations on
       shapes.
*)
module Transformation =
    open RayTracer.SceneAssets
    open System
    type Point = Point.Point
    type Vector = Vector.Vector
    type Shape = SceneAssets.IShape
    type Ray = SceneAssets.Ray
    type matrix = float[,]
    type Transformation = matrix*matrix

    // The transpose of a matrix swaps all rows with columns and
       is used to     transform   the normal
    let transpose (m:matrix) : matrix =
        let arrayOfArrays =
        [|[| m.[0,0]; m.[0,1]; m.[0,2]; m.[0,3]|];
        [|m.[1,0]; m.[1,1]; m.[1,2]; m.[1,3]|];
        [|m.[2,0]; m.[2,1]; m.[2,2]; m.[2,3]|];
        [|m.[3,0]; m.[3,1]; m.[3,2]; m.[3,3]|]|]
        Array2D.init 4 4 (fun i j -> arrayOfArrays.[i].[j])

    let init () : matrix =  let arrayOfArrays =
        [|[| 1.0; 0.0; 0.0; 0.0|];
        [|0.0; 1.0; 0.0; 0.0|];
        [|0.0; 0.0; 1.0; 0.0|];
        [|0.0; 0.0; 0.0; 1.0|]|]
        Array2D.init 4 4 (fun i j -> arrayOfArrays.[i].[j])

    // Translates/moves shape
    let translate (x : float) (y : float) (z : float) :
       Transformation =
        let matrix = init()
        matrix.[3,0] <- x
        matrix.[3,1] <- y
        matrix.[3,2] <- z
        let invmatrix = init()
        invmatrix.[3,0] <- -x
```

```
        invmatrix.[3,1] <- -y
        invmatrix.[3,2] <- -z
        (matrix,invmatrix)

    // The below functions are used to rotate shapes around the (x
       ,y,z) axis using a radian angle

    // Rotate around the x-axis
    let rotateX (angle : float) : Transformation =
        let matrix = init()
        matrix.[1,1] <- Math.Cos(angle)
        matrix.[2,1] <- -Math.Sin(angle)
        matrix.[1,2] <- Math.Sin(angle)
        matrix.[2,2] <- Math.Cos(angle)
        let invmatrix = init()
        invmatrix.[1,1] <- Math.Cos(angle)
        invmatrix.[2,1] <- Math.Sin(angle)
        invmatrix.[1,2] <- -Math.Sin(angle)
        invmatrix.[2,2] <- Math.Cos(angle)
        (matrix,invmatrix)

    // Rotate around y-axis
    let rotateY (angle : float) : Transformation =
        let matrix = init()
        matrix.[0,0] <- Math.Cos(angle)
        matrix.[2,0] <- Math.Sin(angle)
        matrix.[0,2] <- -Math.Sin(angle)
        matrix.[2,2] <- Math.Cos(angle)
        let invmatrix = init()
        invmatrix.[0,0] <- Math.Cos(angle)
        invmatrix.[2,0] <- -Math.Sin(angle)
        invmatrix.[0,2] <- Math.Sin(angle)
        invmatrix.[2,2] <- Math.Cos(angle)
        (matrix,invmatrix)

    // Rotate around z-axis
    let rotateZ (angle : float) : Transformation =
        let matrix = init()
        matrix.[0,0] <- Math.Cos(angle)
        matrix.[1,0] <- -Math.Sin(angle)
        matrix.[0,1] <- Math.Sin(angle)
        matrix.[1,1] <- Math.Cos(angle)
        let invmatrix = init()
        invmatrix.[0,0] <- Math.Cos(angle)
        invmatrix.[1,0] <- Math.Sin(angle)
        invmatrix.[0,1] <- -Math.Sin(angle)
        invmatrix.[1,1] <- Math.Cos(angle)
        (matrix,invmatrix)
```

```
// The below functions are used to shear a shape in indicated
   direction

// Shear in XY direction (Sxy)
let sheareXY (distance : float) : Transformation =
    let matrix = init()
    matrix.[0,1] <- distance
    let invMatrix = init()
    invMatrix.[0,1] <- -distance
    (matrix, invMatrix)

// Shear in XZ direction (Szx)
let sheareXZ (distance : float) : Transformation =
    let matrix = init()
    matrix.[0,2] <- distance
    let invMatrix = init()
    invMatrix.[0,2] <- -distance
    (matrix, invMatrix)

// Shear in YX direction (Syx)
let sheareYX (distance : float) : Transformation =
    let matrix = init()
    matrix.[1,0] <- distance
    let invMatrix = init()
    invMatrix.[1,0] <- -distance
    (matrix, invMatrix)

// Shear in YZ direction (Syz)
let sheareYZ (distance : float) : Transformation =
    let matrix = init()
    matrix.[1,2] <- distance
    let invMatrix = init()
    invMatrix.[1,2] <- -distance
    (matrix, invMatrix)

// Shear in ZX direction (Szx)
let sheareZX (distance : float) : Transformation =
    let matrix = init()
    matrix.[2,0] <- distance
    let invMatrix = init()
    invMatrix.[2,0] <- -distance
    (matrix, invMatrix)

// Shear in ZY direction (Szy)
let sheareZY (distance : float) : Transformation =
    let matrix = init()
    matrix.[2,1] <- distance
    let invMatrix = init()
    invMatrix.[2,1] <- -distance
```

```
    (matrix, invMatrix)

// Scale shape in all axes using floating point numbers
// Numbers lower than one shrinks a shape and numbers greater
   than one grows it (e.g. 2.0 is double size)
let scale (x : float) (y : float) (z : float) : Transformation
    =
    let matrix = init() // Creates initially empty matrix of
       dimension 4x4
    matrix.[0,0] <- x
    matrix.[1,1] <- y
    matrix.[2,2] <- z
    let invmatrix = init() // Creates initially empty matrix
       of dimension 4x4
    invmatrix.[0,0] <- 1.0/x
    invmatrix.[1,1] <- 1.0/y
    invmatrix.[2,2] <- 1.0/z
    (matrix, invmatrix)

// The below functions are used to mirror a shape by inverting
    the sign of a point

// Mirror shape around x axis
let mirrorX : Transformation =
    let matrix = init()
    matrix.[0,0] <- -1.0 // Invert sign of a (x-axis) to - 1
    (matrix, matrix) // Mirror is its own inverse

// Mirror shape around y axis
let mirrorY : Transformation =
    let matrix = init()
    matrix.[1,1] <- -1.0 // Invert sign of b (y-axis) to - 1
    (matrix,matrix) // Mirror is its own inverse

// Mirror shape around z axis
let mirrorZ : Transformation =
    let matrix = init()
    matrix.[2,2] <- -1.0 // // Invert sign of c (z-axis) to -
       1
    (matrix,matrix) // Mirror is its own inverse

let transformPoint (p : Point.Point) (m:matrix) : Point.Point
   =
    let (x,y,z) = Point.getCoord p
    let m2 = [x;y;z;1.0]
    Point.mkPoint   (m2.[0]*m.[0,0]+m2.[1]*m.[1,0]+m2.[2]*m
       .[2,0]+m2.[3]*m.[3,0])
                    (m2.[0]*m.[0,1]+m2.[1]*m.[1,1]+m2.[2]*m
                       .[2,1]+m2.[3]*m.[3,1])
```

```
                    (m2.[0]*m.[0,2]+m2.[1]*m.[1,2]+m2.[2]*m
                       .[2,2]+m2.[3]*m.[3,2])

let transformVector v (m:matrix) : Vector =
    let (x,y,z) = Vector.getCoord v
    let m2 = [x;y;z;0.0]
    Vector.mkVector (m2.[0]*m.[0,0]+m2.[1]*m.[1,0]+m2.[2]*m
       .[2,0]+m2.[3]*m.[3,0])
                    (m2.[0]*m.[0,1]+m2.[1]*m.[1,1]+m2.[2]*m
                       .[2,1]+m2.[3]*m.[3,1])
                    (m2.[0]*m.[0,2]+m2.[1]*m.[1,2]+m2.[2]*m
                       .[2,2]+m2.[3]*m.[3,2])


let transformHit hit trans =
    let (m,inv) = trans
    match hit with
        |(dist,Some hit,Some normal,mat) ->
                    let normal = Vector.normalise (
                       transformVector normal (m))
                    (dist,Some (transformPoint hit m),Some
                       normal,mat)
        |_ -> hit

let transformBbox (bbox : BoundingBox.BoundingBox) (tr :
   Transformation) =
    let higher = bbox.higher
    let lower = bbox.lower
    let minHighFront = Point.mkPoint (Point.getX lower) (Point
       .getY higher) (Point.getZ higher)
    let maxHighFront = higher
    let minLowFront = Point.mkPoint (Point.getX lower) (Point.
       getY lower)(Point.getZ higher)
    let maxLowFront = Point.mkPoint (Point.getX higher) (Point
       .getY lower) (Point.getZ higher)
    let minHighBack = Point.mkPoint (Point.getX lower) (Point.
       getY higher) (Point.getZ lower)
    let maxHighBack = Point.mkPoint (Point.getX higher) (Point
       .getY higher) (Point.getZ lower)
    let minLowBack = lower
    let maxLowBack = Point.mkPoint (Point.getX higher) (Point.
       getY lower) (Point.getZ lower)
    let (matrix, inverse) = tr
    let tMinHighFront = transformPoint minHighFront matrix
    let tMaxHighFront = transformPoint maxHighFront matrix
    let tMinLowFront =  transformPoint minLowFront matrix
    let tMaxLowFront =  transformPoint maxLowFront matrix
    let tMinHighBack =  transformPoint minHighBack matrix
    let tMaxHighBack =  transformPoint maxHighBack matrix
```

```
        let tMinLowBack = transformPoint minLowBack matrix
        let tMaxLowBack = transformPoint maxLowBack matrix

        let posInf = (float) Single.PositiveInfinity
        let negInf = (float) Single.NegativeInfinity
        let (min, max) =
            List.fold (fun (currentMin, currentMax) point ->
                ((
                    BoundingBox.findLowestPoint currentMin point),
                    (BoundingBox.findHighestPoint currentMax point
                       )
                 ))
                ((Point.mkPoint posInf  posInf posInf),(Point.
                   mkPoint negInf negInf negInf))
                [tMinHighFront; tMaxHighFront; tMinLowFront;
                   tMaxLowFront; tMinHighBack; tMaxHighBack;
                   tMinLowBack; tMaxLowBack]
        SceneAssets.mkBbox min max

    let transform (sh : Shape) (tr : Transformation) : Shape =
        {
            new Shape with
                member this.HitFunc (ray : Ray) =
                        let p = getOriginPoint ray
                        let v = getDirection ray
                        let (m,inv) = tr
                        transformHit (sh.HitFunc (mkRay (
                           transformPoint p inv) (transformVector
                           v inv))) (m,inv)

                member this.Bbox = transformBbox sh.Bbox tr
                member this.HitBbox ray = BoundingBox.
                   checkBboxIntersection this.Bbox ray
                member this.IsInside p =
                    let (m,inv) = tr
                    sh.IsInside (transformPoint p inv)
        }

    // Merge two transformations into one transformation
    let mergeMatrices (m1:matrix) (m2:matrix) : matrix =
            let merge (row:int) (column:int):float =
                m1.[0,row]*m2.[column,0]+m1.[1,row]*m2.[column,1]
                + m1.[2,row]*m2.[column,2]+ m1.[3,row]*m2.[column
                   ,3]
            let arrayOfArrays =
                [|[| merge 0 0; merge 1 0; merge 2 0; merge 3 0|];
                [|merge 0 1; merge 1 1; merge 2 1; merge 3 1|];
                [|merge 0 2; merge 1 2; merge 2 2; merge 3 2|];
                [|merge 0 3; merge 1 3; merge 2 3; merge 3 3|]|]
```

```
                Array2D.init 4 4 (fun i j -> arrayOfArrays.[i].[j
                    ])

    // Merge multiple transformations into one transformation
    let mergeTransformations (transformations: Transformation list
        ) =
        List.fold (fun ((m1:matrix),(i1:matrix)) ((m2:matrix),(i2:
            matrix)) ->
            (mergeMatrices m2 m1, mergeMatrices i1 i2)) (init(),
                init()) transformations
```

8.6.16 PolynomialFormulas.fsi

```
module PolynomialFormulas

type Poly = ExprToPoly.poly
type Expr = ExprParse.expr

val solveSecondDegree: float -> float -> float -> float option
val solveSecondDegreeP: Poly -> float option
val solveFirstDgr: Expr -> float option
//val solveCubic: Poly -> string -> float option
//val solveQuartic: Poly -> string -> float option
val polyLongDiv: Poly -> Poly -> string -> Expr*Expr
val deriveExpr: Expr -> Expr
val simplifyToExpr: Expr -> Expr
val newtonsMethod: Expr -> Expr -> float -> int -> float option
val sturmSeq: Expr -> (float*float) option
val sturmPolynomials: Expr -> int -> Expr list
//val newtonConverge: float -> float -> Poly -> float option
```

8.6.17 PolynomialFormulas.fs

```
(*
This module concerns itself with solving polynomials of different
    degrees.

Helpful links:

The cubic formula (3rd)   -> https://www.youtube.com/watch?v=
    wRkXP3eRiy8
The quartic formula (4th) -> https://www.youtube.com/watch?v=3
    lYTBIEgyaM
*)
module PolynomialFormulas

    open ExprToPoly
    open ExprParse
    open RayTracer.FloatHelper
    type Poly = ExprToPoly.poly
```

```
type Expr = ExprParse.expr
type Atom = ExprToPoly.atom
type AtomGroup = ExprToPoly.atomGroup
type SimpleExpr = ExprToPoly.simpleExpr

//converts atom list list to an atom group list
let rec atomListListToAtomGrp (list : atom list list) =
    match list with
    |x::xs -> (x:AtomGroup) :: atomListListToAtomGrp xs
    |[] -> []

//Converts atom group list to type poly
let atomGrpListToPoly (atomGrp:AtomGroup list) (s:string) =
   simpleExprToPoly (SE atomGrp) s

//Converts an atom list list to type poly
let atLLToPoly (list : atom list list) (s:string) = (
   atomListListToAtomGrp >> atomGrpListToPoly) list s

//Converts type poly to expression in given string
let polyToExpr (polynomial:Poly) (s:string) = (ppPoly s
   polynomial) |> scan |> insertMult |> parse

let getPolyMap (p:poly) =
    match p with
    P map -> map

//Converts poly to a map<int, expr> with variables in it.
let polyToExprMap (p:poly) (var:string) =
    let map = getPolyMap p
    Map.fold (fun state key value -> Map.add key (FMult(
       simpleExprToExpr value, FExponent(FVar var, key)))
       state) Map.empty map

let ExprMapToExpr (map:Map<int, Expr>) =
    Map.fold (fun state key value -> FAdd(state, value)) (FInt
        0) map

let simplifyToExpr (e:Expr) = (exprToSimpleExpr >>
   simplifySimpleExpr >> simpleExprToExpr) e


let polyLongDiv (p1:Poly) (p2:Poly) (variable:string) =
    let expr2 = polyToExpr p2 variable
    let expr2Dgr = (getExprDegree (simplifyToExpr expr2) 0)
    let sndLeadMap = polyToExprMap p2 variable

    let rec divide q expr1 =
        let expr1Dgr = (getExprDegree expr1) 0
```

```
        match expr1Dgr >= expr2Dgr with
        | true -> let leadMap = polyToExprMap (exprToPoly
           expr1 variable) variable //Creates leadmap (r)
                  //t = lead(r)/lead(d)
                  let leadR = (Map.find expr1Dgr leadMap)
                  let leadD = (Map.find expr2Dgr sndLeadMap)
                  let t = FDiv(leadR, leadD)

                  //r = r - (t*d)
                  let newExpr1 = simplifyToExpr (FMin(expr1,
                     FMult(t, expr2)))

                  //q = q+t
                  let newq = simplifyToExpr (FAdd(q, t))
                  divide newq newExpr1
        | false -> (q, expr1)
    divide (FNum 0.0) ((polyToExprMap p1 variable) |>
       ExprMapToExpr)

let rec containsVar = function
    | FAdd(e1, e2) -> containsVar e1 || containsVar e2
    | FMin(e1, e2) -> containsVar e1 || containsVar e2
    | FMult(e1, e2) -> containsVar e1 || containsVar e2
    | FDiv(e1, e2) -> containsVar e1 || containsVar e2
    | FRoot(e1, _) -> containsVar e1
    | FExponent(e1, _) -> containsVar e1
    | FVar s -> true
    | _ -> false

let rec deriveExpr (e:expr) =
    match e with
    |FExponent(e1, 0) -> FInt 1
    |FExponent(e1, 1) -> e1
    |FExponent(FVar s, i) -> FMult(FInt i, FExponent(FVar s, i
       -1))
    |FAdd(e1, e2) ->
        match (containsVar e1, containsVar e2) with
            | (true, true) -> FAdd(deriveExpr e1, deriveExpr
               e2)
            | (true, false) -> FAdd(deriveExpr e1, FInt 0)
            | (false, true) -> FAdd(FInt 0, deriveExpr e2)
            | (false, false) -> FInt 0
    |FMin(e1, e2) ->
        match (containsVar e1, containsVar e2) with
        | (true, true) -> FMin(deriveExpr e1, deriveExpr e2)
        | (true, false) -> FMin(deriveExpr e1, FInt 0)
        | (false, true) -> FMin(FInt 0, deriveExpr e2)
        | (false, false) -> FInt 0
```

```
    |FMult(e1, e2) ->
        match (containsVar e1, containsVar e2) with
            | (true, true) -> FMult(deriveExpr e1, deriveExpr
               e2)
            | (true, false) -> FMult(deriveExpr e1, e2)
            | (false, true) -> FMult(e1, deriveExpr e2)
            | (false, false) -> FMult(e1, e2)
    |FDiv(e1, e2) -> FDiv(FMin(FMult(e2, deriveExpr e1), FMult
       (e1, deriveExpr e2)), FExponent(e2, 2)) //http://
       archives.math.utk.edu/visual.calculus/2/quotient_rule
       .4/
    |FVar s -> FInt 1
    |FNum f -> FNum f
    |FInt i -> FInt i

//Solves a second degree polynomial given a, b and c.
//Only returns smallest root
// Todo replace with separate fs/fsi for solving second degree
    polynomials
let solveSecondDegree a b c =
    let disc = b*b - (4.0*a*c)
    if disc < 0.0
    then None
    else if disc = 0.0
    then Some ((-1.0*b) / 2.0*a)
    else
        let dist1 = (-1.0*b + System.Math.Sqrt(disc))/(2.0*a)
        let dist2 = (-1.0*b - System.Math.Sqrt(disc))/(2.0*a)
        if(dist1>ZeroThreshold && dist2>ZeroThreshold) then
            if dist1 < dist2 then Some dist1
            else
                Some dist2
        else if(dist1>ZeroThreshold) then
            Some dist1
        else if (dist2>ZeroThreshold) then
            Some dist2
        else
            None


let solveSecondDegreeP (p:Poly) =
    let map = getPolyMap p
    match map.Count - 1 with
    | 2 -> let exValues = Map.foldBack (fun key value state ->
        Array.append [|(value |> simpleExprToExpr |>
       getExprValue)|] state) map [| |]
           solveSecondDegree exValues.[2] exValues.[1]
              exValues.[0]
    | _ -> None
```

```
let solveFirstDgr (ex: expr) =
     let rec isolateVar (left : expr) (right:expr) =
         match left with
         | FAdd(e1, e2) ->
            match (containsVar e1, containsVar e2) with
                | (true, false) -> isolateVar e1 (FAdd(right,
                   (FMult(FInt -1, e2))))
                | (false, true) -> isolateVar e2 (FAdd(right,
                   (FMult(FInt -1, e1))))
                | (false, false) -> failwith "No var found."
         | FMin(e1, e2) ->
            match (containsVar e1, containsVar e2) with
                | (true, false) -> isolateVar e1 (FAdd(right,
                   e2))
                | (false, true) -> isolateVar e2 (FMin (right,
                    e1))
                | (false, false) -> failwith "No var found."
         | FMult(e1, e2) ->
            match (containsVar e1, containsVar e2) with
                | (true, false) -> isolateVar e1 (FDiv(right,
                   e2))
                | (false, true) -> isolateVar e2 (FDiv(right,
                   e1))
                | (false, false) -> failwith "No var found."
         | FDiv(e1, e2) ->
            match (containsVar e1, containsVar e2) with
                | (true, false) -> isolateVar e1 (FMult(right,
                    e2))
                | (false, true) -> isolateVar e2 (FMult(right,
                    e1))
                | (false, false) -> failwith "No var found."
         | FExponent(e1, 0) -> failwith "Var to the pwr of 0."
         | FExponent(e1, 1) -> isolateVar e1 right
         | FVar s -> right
         | _ -> failwith "no var found."
     let dist = getExprValue (isolateVar ex (FInt 0))
     if dist >= 0.0 then Some dist
     else None

//given a poly and its degree
let rec sturmPolynomials (p0:expr) (dgr:int) =

    let p1 = deriveExpr p0
    let rec findSturmSeq (px0:expr) (px1:expr) (dgr1:int) (acc
       :expr list) =
       match dgr >= 0 with
       | true -> let px = FMult(FInt -1, (snd (polyLongDiv (
          exprToPoly px0 "t") (exprToPoly px1 "t") "t")))
```

```
                findSturmSeq px1 px (dgr1-1) (px::acc)
        | false -> acc
    findSturmSeq p0 p1 (dgr-2) [p1;p0]

let signSwitch (v1:expr) (v2:expr) =
    match (getExprValue v1, getExprValue v2) with
    | (x, y) when x > 0.0 && y < 0.0 -> true
    | (x, y) when y > 0.0 && x < 0.0 -> true
    | (0.0, y) when y < 0.0 -> true
    | (x, 0.0) when x < 0.0 -> true
    | _ -> false

let rec signSwitchSum (el : expr list) (acc:int) =
    match el with
    |x::x1::xs -> if signSwitch x x1
                  then signSwitchSum (x1::xs) (acc+1)
                  else signSwitchSum (x1::xs) acc
    |x::xs -> if signSwitch x (el |> Seq.skip (el.Length - 1)
        |> Seq.head)
              then acc + 1
              else acc
    |[] -> acc

let rec sturmSeq (p:expr) =
    let sturm = sturmPolynomials p (getExprDegree p 0)

    let rec partition (l:expr list) (floor:float) (range:float
        ) (i:int) =
        if i >= 7 then Some (floor, range)
        else
            let a = List.fold (fun s x -> ((subst x ("t", FNum
                floor))) :: s) [] l
            let b = List.fold (fun s x -> ((subst x ("t", FNum
                range))) :: s) [] l
            let roots = (signSwitchSum a 0) - (signSwitchSum b
                0)
            if roots <= 0 then None
            else
            match partition l floor (range - ((range-floor)
               /2.0)) (i+1) with //Look in left interval
            | Some l1 -> Some l1//Look onwards in left
            | None -> partition l (floor + (range - floor)
               /2.0) range (i+1)//Look in right interval
    partition sturm 0.0 100.0 0

let rec newtonsMethod (p:Expr) (pd:Expr) (x0:float) (i:int) =
    //Max 20 recursive calls.
    if i >= 5 then Some x0
    else
```

```
            //Insert x0 into y and y'
            let y = subst p ("t", FNum x0) |> simplifyToExpr |>
               getExprValue
            let y' = subst pd ("t", FNum x0) |> simplifyToExpr |>
               getExprValue
            newtonsMethod p  pd (x0 - (y/y')) (i+1)
```

8.6.18 KDTree.fsi

```
namespace RayTracer

module KDTree =
    type Shape = SceneAssets.IShape
    type Scene = SceneAssets
    type BoundingBox = RayTracer.BoundingBox.BoundingBox
    type Point = Point.Point
    type Vector = Vector.Vector
    type Ray = SceneAssets.Ray
    type Material = SceneAssets.Material

    type Tree =
        | Leaf of Shape list
        | Node of int * float * Tree * Tree

    val traverse:  Tree -> BoundingBox -> Ray ->  (Point*Vector*
       Material) option
    val buildKdTree: Shape list -> Tree * BoundingBox.BoundingBox
```

8.6.19 KDTree.fs

```
namespace RayTracer

open System
open Point
open BoundingBox
open SceneAssets

//The module represents the KD-tree and contains its functions
module KDTree =
    type Shape = SceneAssets.IShape
    type Scene = SceneAssets
    type BoundingBox = RayTracer.BoundingBox.BoundingBox
    type Point = Point.Point
    type Vector = Vector.Vector
    type Ray = SceneAssets.Ray
    type Material = SceneAssets.Material

    type Tree =
        | Leaf of Shape list
        | Node of int * float * Tree * Tree
```

```
//---- Help Functions ----

//Orders tuple based on d
let order (d, left, right) =
    if d > 0.0
    then (left,right)
    else (right,left)

//Retrieves a tuple stored in a node
let getNodeTuple node =
    match node with
    | Node(dim,value,left,right) -> (dim, value, left, right)
    | Leaf l -> failwith "Can't return tuple from Leaf"

//Compute bounding box of whole tree
let computeTreeBbox (shapes : Shape list) : BoundingBox =
    let startPoint = (Point.mkPoint 0.0 0.0 0.0)
    let inf = (float) Single.PositiveInfinity
    let negInf = (float) Single.NegativeInfinity
    let lowerPoint = List.fold (fun point (shape : Shape)->
       findLowestPoint point shape.Bbox.lower) (Point.mkPoint
       inf inf inf) shapes
    let higherPoint = List.fold (fun point (shape : Shape) ->
       findHighestPoint point shape.Bbox.higher) (Point.
       mkPoint negInf negInf negInf) shapes
    mkBbox lowerPoint higherPoint


//Gets coordinate based on what dimension to retrieve
let getCoordByDimension (dimension : int) =
    match dimension with
    | 1 -> Point.getX
    | 2 -> Point.getY
    | 3 -> Point.getZ
    | _ -> failwith "Invalid dimension"

//Get coordinate from vector by a given dimension
let getVectorCoord (dimension : int) =
    match dimension with
    | 1 -> Vector.getX
    | 2 -> Vector.getY
    | 3 -> Vector.getZ
    | _ -> failwith "Invalid dimension"

//Calculates the split coordinate by calculating a mean based
   on a given dimension and a list of bounding boxes (Used to
   find the split point)
let findMean (dimension : int) (shapes : Shape list) =
```

```
    let sumOfCoordinates = (List.fold (fun sum (shape : Shape)
        -> sum + (getCoordByDimension dimension (shape.Bbox.
       lower) + getCoordByDimension dimension (shape.Bbox.
       higher))/2.0) 0.0 shapes)
    let mean = sumOfCoordinates/ float (shapes.Length)
    mean

//Calculate longest dimension of a given bounding box
let findLargestDimension (bbox : BoundingBox.BoundingBox) =
 //Calculate dimension size
    let xDim = Point.getX bbox.higher - Point.getX bbox.lower
    let yDim = Point.getY bbox.higher - Point.getY bbox.lower
    let zDim = Point.getZ bbox.higher - Point.getZ bbox.lower
    let x = 1
    let y = 2
    let z = 3

    //Compares dimensions
    if(xDim >= yDim && xDim >= zDim)        //return x
        then x
    else if (yDim >= xDim && yDim >= zDim)  //return y
        then y
    else                                    //return z
             z

//Distribute shapes based on their bounding box and mean
let rec distributeBboxes (dimension : int) (mean : float) (
   shapes : Shape list) =
    List.fold (fun (lower, both, upper) (shape : Shape)
        ->
           let higherBboxPoint = getCoordByDimension dimension
               shape.Bbox.higher
           let lowerBboxPoint = getCoordByDimension dimension
              shape.Bbox.lower
           //Shape is below mean
           if higherBboxPoint < mean
                then   (shape :: lower, both, upper)
           //Shape is above mean
           else if lowerBboxPoint > mean
                then   (lower, both, shape :: upper)
           //Shape is inbetween
           else
               (shape :: lower, both + 1, shape :: upper)
       ) ([], 0, []) shapes

//Finds the closest coordinate above and below the mean - Used
    to calculate innerEmptySpace
let rec findInnerEmptySpaceCoords (dimension : int) (mean :
   float) (shapes : Shape list) =
```

```
        List.fold (fun (lowerMax, upperMin) (shape : Shape)
          ->
            //Find the two points that are closest to the mean
                above an below
            let upperBboxPoint = getCoordByDimension dimension
                shape.Bbox.higher
            let lowerBboxPoint = getCoordByDimension dimension
                shape.Bbox.lower

            //Shape is below the mean
            if (upperBboxPoint < mean) then
                let newLowerMax =
                    if upperBboxPoint > lowerMax
                    then upperBboxPoint
                    else lowerMax
                (newLowerMax, upperMin)

            //Shape is above the mean
            else if  (lowerBboxPoint > mean) then
                let newUpperMin =
                    if lowerBboxPoint < upperMin
                    then lowerBboxPoint
                    else upperMin
                (lowerMax, newUpperMin)

           //Shape is in between the mean
           else
                let newLowerMax =
                    if lowerMax < upperBboxPoint //upper point
                        becomes lower max
                    then upperBboxPoint
                    else lowerMax
                let newUpperMin =
                    if upperMin > lowerBboxPoint //lower point
                        becomes upper min
                    then lowerBboxPoint
                    else upperMin
                (newLowerMax, upperMin))
          ((float) Single.NegativeInfinity, (float) Single.
             PositiveInfinity) shapes

//Creates upper and lower child bounding boxes based on
   dimensions
let findChildBboxes dimension mean (outerBbox : BoundingBox.
   BoundingBox) =
    match dimension with
    //X axis
    | 1 ->
        //Lower Bbox
```

```
    ((mkBbox
        //Lower Point
            (outerBbox.lower)
        //Higher Point
            (Point.mkPoint
            mean
            (Point.getY outerBbox.higher)
            (Point.getZ outerBbox.higher))),
    //Upper Bbox
      (mkBbox
        //Lower Point
            (Point.mkPoint
            mean
            (Point.getY outerBbox.lower)
            (Point.getZ outerBbox.lower))
        //Higher Point
            (outerBbox.higher)))
//Y axis
| 2 ->
    //Lower Bbox
    ((mkBbox
        //Lower Point
            (outerBbox.lower)
       //Higher Point)
            (Point.mkPoint
            (Point.getX outerBbox.higher)
            mean
            (Point.getZ outerBbox.higher))),
    //Upper Bbox
      (mkBbox
        //Lower Point
            (Point.mkPoint
            (Point.getX outerBbox.lower)
            mean
            (Point.getZ outerBbox.lower))
        //Higher Point
            (outerBbox.higher)))
//Z axis
| 3 ->
    //Lower Bbox
      ((mkBbox
         //Lower Point
            (outerBbox.lower)
         //Higher Point
            (Point.mkPoint
            (Point.getX outerBbox.higher)
            (Point.getY outerBbox.higher)
            mean)),
    //upper Bbox
```

```
            (mkBbox
             //Lower Point
                (Point.mkPoint
                (Point.getX outerBbox.lower)
                (Point.getY outerBbox.lower)
                (mean))
             //Higher Point
                (outerBbox.higher)))
    | _ -> failwith "Invalid dimension"

//Finds the outer empty space
let getOuterEmptySpace (outerBbox : BoundingBox.BoundingBox) (
   shapes : Shape list) dimension =
    let (l, h) = List.fold (fun (lowerOuter, higherOuter) (
       shape : Shape)
        ->
            let bboxLower = getCoordByDimension dimension
               shape.Bbox.lower
            let bboxOuter = getCoordByDimension dimension
               shape.Bbox.higher
            let lower =
                if bboxLower < lowerOuter then bboxLower
                else lowerOuter
            let outer =
                if bboxOuter > higherOuterthen bboxOuter
                else higherOuter
            (lower, outer)
          ) ((float)Single.PositiveInfinity, (float) Single.
             NegativeInfinity) shapes
    let lowerDist = l - getCoordByDimension dimension
       outerBbox.lower
    let higherDist = getCoordByDimension dimension outerBbox.
       higher - h
    (lowerDist, l, higherDist, h)


//---- HEURISTICS ----
let splitHeuristic = 0.60 //Heuristic percentage used to check
    splitting.
let emptySpaceHeuristic = 0.10//Heuristic percentage used to
   cut off empty space.
// -------------------

//Move the current mean so that it takes inner empty space
   into account and return the new position
let computeNearestMean lowerMax upperMin mean bboxLength =
    let lowerDistRatio = (mean-lowerMax)/bboxLength
    let upperDistRatio = (upperMin - mean)/bboxLength
    match (lowerDistRatio, upperDistRatio) with
```

```
        | (lowerDist, upperDist)
            when lowerDistRatio > emptySpaceHeuristic &&
               lowerMax < mean && lowerDistRatio >=
               upperDistRatio
                 -> lowerMax
        | (lowerDist, upperDist)
            when upperDistRatio > emptySpaceHeuristic &&
               upperMin > mean
                 -> upperMin
        | _ -> mean

//Calculate the ratio for how many shapes that overlaps the
   mean
let calculatePlacementRatios lowerCount upperCount bothCount =
    let bl = (float bothCount/float lowerCount)
    let br = (float bothCount/float upperCount)
    (bl, br)

//Main split function
let rec splitOnDimension (outerBbox : BoundingBox) (shapes :
   Shape list) c =

    //Finds the dimension, which is largest (this is the one
       we split on)
    let largestDimension = findLargestDimension outerBbox
    let dimLength = (getCoordByDimension largestDimension
       outerBbox.higher - getCoordByDimension largestDimension
        outerBbox.lower)

    //Computes the empty space that there is from the edges of
        the outer bbox to the closest shapes.
    let (lowerDist, lower, higherDist, higher) =
       getOuterEmptySpace outerBbox shapes largestDimension
    //If the distance from the lower end of the bbox is larger
        than the distance from the higher end
    //and it satisfies the heuristics, then split on this,
       leaving empty space on the right.
    if (lowerDist/dimLength) >= (higherDist/dimLength) &&
       lowerDist >= emptySpaceHeuristic then
        let splitValue = lower - FloatHelper.ZeroThreshold
        let (bboxLower, bboxHigher) = findChildBboxes
           largestDimension splitValue outerBbox
        splitOnDimension bboxHigher shapes (fun leftTree -> c
           (Node(largestDimension, splitValue, Leaf (List.
           Empty), leftTree)))

    //Else if the distance from the higher end is largest and
       satisfies heuristics, split on this value
    //leaving empty space to the left.
```

```
        else if ((higherDist/dimLength) >= emptySpaceHeuristic)
           then
            let splitValue = higher + FloatHelper.ZeroThreshold
            let (bboxLower, bboxHigher) = findChildBboxes
               largestDimension splitValue outerBbox
            splitOnDimension bboxLower shapes (fun rightTree -> c
               (Node(largestDimension, splitValue, rightTree, Leaf
                (List.Empty))))
        else
        //If the empty space is not large enough to satisfy
           heuristics, split normally
            splitOnMean largestDimension shapes outerBbox
               dimLength c

    and splitOnMean largestDimension shapes outerBbox dimLength c=
        //Returns the split value based on where the bboxes are
           situated (the mean)
        let mean = findMean largestDimension shapes

        //Calculates which shapes lie within the lower half, upper
            half and a count of how many lie within both.
        //In addition, returns the point of the shapes that lie
           closest to the mean on either side of the mean
        let (lowerShapes : Shape list, bothCount, upperShapes :
           Shape list) = distributeBboxes largestDimension mean
           shapes
        let (lowerMax, upperMin) = findInnerEmptySpaceCoords
           largestDimension mean shapes

        //Computes the ratio of how many shapes lie in the lower
           and upper half compared to how many lie in both
        let (blRatio, brRatio) = calculatePlacementRatios
           lowerShapes.Length upperShapes.Length bothCount

        //Avoid dividing along a plane which causes many overlaps
           (i.e. with many shapes on both sides)
        if (blRatio > splitHeuristic || brRatio > splitHeuristic)
            then c (Leaf(shapes))
        else
        //Calculates the best mean, taking empty space ito account
        let closestMean = computeNearestMean lowerMax upperMin
           mean dimLength

        //Calculates bboxes of the areas split by the mean
        let (lowerBbox, upperBbox) = findChildBboxes
           largestDimension closestMean outerBbox

        //Calculates KDTree based on above calculations
        splitOnDimension lowerBbox lowerShapes
```

```
                (fun rightTree -> splitOnDimension upperBbox
                   upperShapes (fun leftTree -> c (Node(
                   largestDimension, closestMean, rightTree,
                   leftTree))))


//----- tree traversal functions ------//

    let closestHit (leaf : Tree) (ray:Ray) =
        match leaf with
        | Leaf shapes ->
            let possibleHits = List.map(fun (s : Shape)-> s.
               HitFunc ray) shapes //make the hitFunc for all
               shapes
            let filteredHits = List.filter(fun ((d: float option),
               _,_,_) -> d.IsSome ) possibleHits
            let sortedHits = List.sortBy(fun ((d: float option),_,
               _,_) -> d.Value ) filteredHits // sort in ascending
                order

            match sortedHits with
                |(Some dist,Some hit,Some normal,mat)::_ when dist
                   >FloatHelper.ZeroThreshold ->
                    Some(dist, hit,normal,mat()) //return first
                       hit if any
                |_ ->  None
        | _ -> failwith "Expects a leaf but the given param is not
            a leaf"

    let isLeaf (node : Tree) =
        match node with
        | Leaf l -> true
        | _ -> false

    let searchLeaf leaf (ray:Ray) (t':float) =
        let hit = closestHit leaf ray
        match hit with
        | Some (dist, hit, normal, mat) -> Some (hit, normal, mat)
        | _ -> None

    let rec traverse (tree:Tree) (bbox:BoundingBox.BoundingBox)
       ray =
            let (intersects,t,t') = BoundingBox.
               getBboxIntersection bbox ray //Returns a tuple of (
               bool,t,t') //check bounding box of tree t and 't is
                the distance to enter and exit
            if intersects then search tree ray t t'
            else None
    and search node (ray:Ray) (t:float) (t':float) =
```

```
        if isLeaf node
        then
            searchLeaf  node ray t'
        else
            searchNode node ray t t'

    and searchNode node (ray:Ray) t t' =
        let (axis,value,left,right) = getNodeTuple node
        let vectorCoordDim = getVectorCoord axis (ray.direction)
        let originCoordDim = getCoordByDimension axis ray.origin
        if vectorCoordDim = 0.0 then
            if originCoordDim <= value then
                search left ray t (t')
            else
                search right ray t t'
        else
            let tHit = (value - originCoordDim) / vectorCoordDim
            let (fst, snd) = order(vectorCoordDim, left, right)
            if tHit <=  t || tHit < 0.0 then
                search snd ray t t'
            else if tHit >= t' then
                search fst ray t t'
            else
                match (search fst ray t tHit) with
                | Some hit -> Some hit
                | None -> search snd ray tHit t'

    //This function builds a KD-tree based on a list of shapes
    //Mainly used for Triangle meshes but is also used for the
       scene
    let buildKdTree (shapes : Shape list) =
        let kdBbox = computeTreeBbox shapes
        (splitOnDimension kdBbox shapes id, kdBbox)//Cut off empty
            space, check if it is even necessary to split
```

8.6.20 TriangleMesh.fsi

```
namespace RayTracer

module TriangleMesh =
    type Tree = KDTree.Tree
    type BoundingBox = BoundingBox.BoundingBox
    type Shape = SceneAssets.IShape
    type Ray = SceneAssets.Ray
    type Point = Point.Point
    type Vector = Vector.Vector
    type Material = SceneAssets.Material

    val mkTriangleMesh : Shape list -> Shape
    val mkKDtree : Shape list -> Shape
```

8.6.21 TriangleMesh.fs

```
namespace RayTracer

open KDTree
module TriangleMesh =

    type Tree = KDTree.Tree
    type BoundingBox = BoundingBox.BoundingBox
    type Shape = SceneAssets.IShape
    type Ray = SceneAssets.Ray
    type Point = Point.Point
    type Vector = Vector.Vector
    type Material = SceneAssets.Material
    type Vertex = SceneAssets.Vertex

    let TriangleMesh(tree : Tree, (hitBbox : Ray -> bool), bbox :
       BoundingBox) =
        {
            new Shape with
                member this.Bbox = bbox
                member this.HitBbox (ray : Ray) = hitBbox ray
                member this.HitFunc (ray : Ray) =
                    let res = traverse tree bbox ray
                    match res with
                    | Some(hit,normal,mat) -> (Some (Vector.
                       magnitude (Point.distance ray.origin hit)),
                       Some(hit),Some(normal),(fun () -> mat))
                    | None -> (None,None,None,SceneAssets.black)
                member this.IsInside p = failwith "Shape is not
                   solid!"
        }

    let mkKDtree (shapes : Shape list) =
        let (tree, bbox) = buildKdTree shapes
        let hitBb = BoundingBox.checkBboxIntersection bbox
        TriangleMesh(tree, hitBb, bbox)

    let mkTriangleMesh (shapes : Shape list)  = mkKDtree shapes //
       For the sake of abstraction
```

8.6.22 MeshBuilder.fsi

```
namespace RayTracer
(*
    This module is responsible for creating triangle based on the
    information given by an existing PLY file.
*)
module MeshBuilder =
```

```
    type Shape = Shape.IShape
    type TriangleMesh = TriangleMesh
    type Vector = Vector.Vector
    type Vertex = SceneAssets.Vertex
    type Face = SceneAssets.Face
    type Texture= SceneAssets.Texture

    val buildTriangleMesh:  Texture -> bool -> string  -> Shape
```

8.6.23 MeshBuilder.fs

```
namespace RayTracer

(*
    This module is responsible for creating triangle based on the
    information given by an existing PLY file.
*)
open RayTracer.SceneAssets
open RayTracer.TriangleMesh
open RayTracer.Shape
module MeshBuilder =
    type TriangleMesh = TriangleMesh
    type Shape = Shape.IShape
    type Vector = Vector.Vector
    type Vertex = SceneAssets.Vertex
    type Face = SceneAssets.Face
    type Point = SceneAssets.Point
    type Texture = SceneAssets.Texture

    //This function calculates the normals for each vertices
       within a triangle
    let calcVertexNormalsForSmoothShading faces (vertices : Vertex
       []) =
        let normals : Vector[] = Array.init vertices.Length (fun i
            -> Vector.mkVector 0.0 0.0 0.0)
        Array.iter (fun face ->
            match face with
                | Face.F(A,B,C) -> //Calculating normal
                    let pointA = (Array.get vertices A).point
                    let pointB = (Array.get vertices B).point
                    let pointC = (Array.get vertices C).point

                    let u = Point.distance pointB pointA //The
                       side of the triangle u
                    let v = Point.distance pointC pointA //The
                       side of the triangle v
                    let normal = Vector.normalise(Vector.
                       crossProduct u v)
                    //Adds normal to the existing sum of normals
                       in the normals array
```

```
                Array.set normals A ((Array.get normals A) +
                    normal)
                Array.set normals B ((Array.get normals B) +
                    normal)
                Array.set normals C ((Array.get normals C) +
                    normal)) faces

    //Normalize normal vectors
    Array.iteri
    (
        fun i normal ->
            let sumNormal = Array.get normals i
            Array.set normals i (Vector.normalise(sumNormal))
    ) normals

    //Setting new normals to vertices
    Array.iteri2
    (
        fun i newNormal (oldVertex:Vertex) ->
            let newVertex = mkVertex oldVertex.point newNormal
                oldVertex.u oldVertex.v
            Array.set vertices i newVertex
    ) normals vertices

//------- BUILD FUNCTION -------//

//This function takes an existing PLY file, parse it to the
   PLYparser and return a list of triangle where each of their
// respective vertex is normalized
let buildTriangleMesh (texture:Texture) (isSmoothShade: bool)
   filename  =
    let (vertices, faces) = PlyParser.parsePLY filename

    //Check if returned vertices contains normals. If not
       calculate them else use exisiting
    let checkNormal = Vector.getCoord ((Array.get vertices 0).
       normal)
    match checkNormal with
    | (0.0,0.0,0.0) ->
        if isSmoothShade then
           calcVertexNormalsForSmoothShading faces vertices
        else ()
    | (_,_,_) -> ()

    //Create triangle abstraction based on existing faces and
       return them in an array
    let triangles = List.init faces.Length (fun i ->  let face
        = Array.get faces i
        match face with
```

```
                | Face.F(A,B,C) ->
                    let vertexA = Array.get vertices A
                    let vertexB = Array.get vertices B
                    let vertexC = Array.get vertices C
                    (mkMeshTriangle vertexA vertexB vertexC
                       texture isSmoothShade))

        //Optimize structure by putting it into a KDtree and
           returns and ISHAPE
        TriangleMesh.mkTriangleMesh triangles
```

8.6.24 Shape.fsi

```
namespace RayTracer
//This module is responsible for performing functions related to
   shapes
module Shape =
    type Point = Point.Point
    type Texture = SceneAssets.Texture
    type Vector = Vector.Vector
    type BoundingBox = BoundingBox.BoundingBox
    type Ray = SceneAssets.Ray
    type Material = RayTracer.SceneAssets.Material
    type hitBboxFunc = Ray -> bool
    type Vertex = SceneAssets.Vertex
    type IShape = SceneAssets.IShape


    // Shape creation functions
    val mkSphere: Point -> float -> Texture -> IShape
    val mkTriangle : Point -> Point -> Point -> Texture -> IShape
    val mkMeshTriangle : Vertex -> Vertex -> Vertex-> Texture ->
       bool -> IShape
    val mkPlane : Texture -> IShape
    val mkBox : low : Point -> high : Point -> front : Texture ->
       back : Texture ->
              top : Texture -> bottom : Texture -> left : Texture
                 -> right : Texture  -> IShape
    val mkDisc: center : Point -> radius : float -> Texture ->
       IShape
    val mkCylinder: radius : float -> height:  float -> Texture ->
        IShape
    val mkSolidCylinder : r : float -> h : float -> t : Texture ->
        top : Texture -> bottom : Texture -> IShape
    val mkRectangle: bottomLeft:  Point -> width : float -> height
        : float -> Texture -> IShape
    val group : IShape -> IShape  -> IShape

    //val mkImplicit: string -> IShape
```

8.6.25 Shape.fs

```
namespace RayTracer

// Implementation file of shape signature module

open BoundingBox // Used to compute bounding box in shapes
open SceneAssets
open System
open PolynomialFormulas
open Transformation
module Shape =
    type IShape = SceneAssets.IShape
    type Point = Point.Point
    type Expr = ExprParse.expr
    type Vector = Vector.Vector
    type BoundingBox = BoundingBox.BoundingBox
    type Texture = SceneAssets.Texture
    type Ray = SceneAssets.Ray
    type Material = SceneAssets.Material
    type hitBboxFunc = (Ray -> bool)
    type Vertex = SceneAssets.Vertex


    let getNormal (n:Vector) (r:Ray) =
        let v = Vector.normalise r.direction
        let angle = System.Math.Acos ((Vector.dotProduct n v)/(
           Vector.magnitude n * Vector.magnitude v))
        if(angle>Math.PI/2.0)
        then  n
        else -n

    let getNearestHit (shapes: IShape list) ray =
        let possibleHits = List.map(fun (s:IShape) ->  s.HitFunc
           ray) shapes //make the hitFunc for all shapes
        let filteredHits = List.filter(fun ((d: float option),_,_,
           _) -> d.IsSome ) possibleHits
        let sortedHits = List.sortBy(fun ((d: float option),_,_,_)
            -> d.Value ) filteredHits // sort in ascending order
        match sortedHits with
            |(Some dist,Some hit,Some normal,mat)::_ when dist>
               FloatHelper.ZeroThreshold
                      //return first hit if any
                     -> (Some dist,Some hit,Some normal,mat)
            |_ ->  (None, None, None,SceneAssets.black)


    // Shapes and functions used to create them
```

```
let Sphere(center : Point, radius : float, texture : Texture,
    (hitBbox : Ray -> bool), bbox : BoundingBox) =
     {
         new IShape with
             member this.Bbox = bbox
             member this.HitBbox (ray : Ray) = hitBbox ray
             member this.IsInside p = (Vector.magnitude (Point.
                distance p center))<=radius
             member this.HitFunc (ray : Ray) =
                 let p = center
                 let r = radius
                 //help variables for simplicity
                 let px = Point.getX(p) - Point.getX(ray.origin
                    )
                 let py = Point.getY(p) - Point.getY(ray.origin
                    )
                 let pz = Point.getZ(p) - Point.getZ(ray.origin
                    )
                 let dir = Vector.normalise ray.direction

                 let b = -2.0*(px*Vector.getX(dir) +
                                py*Vector.getY(dir) +
                                pz*Vector.getZ(dir))
                 let c = px*px + py*py + pz*pz - r*r // x^2+y
                    ^2+z^2=r^2 equation for a sphere

                 let dist = solveSecondDegree 1.0 b c // a is
                    always 1

                 //Check whether the distance is a valid value
                 if(dist.IsSome) then
                     let hitPoint = Point.move ray.origin (dist
                        .Value*dir)
                     let normal =  Vector.mkVector (2.0 * (
                        Point.getX hitPoint)) (2.0 * (Point.
                        getY hitPoint)) (2.0 * (Point.getZ
                        hitPoint))
                     let matFun =
                         (fun () ->
                           let (hitX,hitY,hitZ) = Point.
                              getCoord hitPoint
                           let pi = System.Math.PI
                           let nx = hitX /r
                           let ny = hitY /r
                           let nz = hitZ /r
                           let theta = acos ny
                           let phi' = atan2 nx nz
                           let phi =
                               if phi' < 0.0
```

```
                                then
                                    phi'+2.0*pi
                                else
                                    phi'
                                getTextureColour texture (phi
                                   /(2.0*pi),1.0-theta/pi))
                            (dist,Some(hitPoint), Some (getNormal
                               normal ray),matFun)

                    else
                      (None, None, None,(fun () ->
                            getTextureColour texture (0.0,0.0)))
            }

let mkSphere (center : Point) (radius : float) (tex : Texture)
    =
    let bbox = BoundingBox.computeSphereBbox radius
    let hitBb = BoundingBox.checkBboxIntersection bbox
    let (x,y,z) = Point.getCoord center
    transform (Sphere(Point.mkPoint 0.0 0.0 0.0,radius,tex,
       hitBb, bbox)) (translate x y z)


let Triangle(A : Point,B : Point,C : Point,texture : Texture,
   (hitBbox : Ray -> bool), bbox : BoundingBox) =
    {
        new IShape with
            member this.Bbox = bbox
            member this.HitBbox (ray : Ray) = hitBbox ray
            member this.HitFunc (ray : Ray) =

                // Triangle sides (u and v) and normal vector
                let us = Point.distance B A
                let vs = Point.distance C A
                let normal = Vector.normalise(Vector.
                   crossProduct us vs)
                let dir = Vector.normalise ray.direction
                //Solving equation with 3 unknowns based on
                   cramer's rule

                //First equation with unknown, ax + by + cz =
                   d
                let a = (Point.getX A) - (Point.getX B)
                let b = (Point.getX A) - (Point.getX C)
                let c = (Vector.getX dir)
                let d = (Point.getX A) - (Point.getX ray.
                   origin)

                //Second equation with unknown, ez + fy + gz =
```

```
    h
let e = (Point.getY A) - (Point.getY B)
let f = (Point.getY A) - (Point.getY C)
let g = (Vector.getY dir)
let h = (Point.getY A) - (Point.getY ray.
   origin)

//Third equation with unknown, ix + jy + kz =
   l
let i = (Point.getZ A) - (Point.getZ B)
let j = (Point.getZ A) - (Point.getZ C)
let k = (Vector.getZ dir)
let l = (Point.getZ A) - (Point.getZ ray.
   origin)

//Finding the unknowns and calculates if there
    is any hit based previous calculations
let D = a*(f*k - g*j) + b*(g*i-e*k) + c*(e*j -
    f*i)
let texture = getTextureColour texture
   (0.0,0.0) // Calculate material

if(D <> 0.0) then
    let beta = (d*(f*k - g*j) + b*(g*l - h*k)
       + c*(h*j - f*l))/D //Corresponds to x
    let gamma = (a*(h*k-g*l) + d*(g*i-e*k) + c
       *(e*l-h*i))/D //Corresponds to y
    let t = (a*(f*l - h*j) + b*(h*i - e*l) + d
       *(e*j-f*i))/D //Corresponds to z
    let alpha = 1.0 - beta - gamma

    if beta >= 0.0 && gamma >= 0.0 && gamma +
       beta <= 1.0 // Need sum to be 1 and in
       between 0 and 1
     then
       let distance = Some t
       let hitPoint = Some(Point.move ray.
          origin (t*dir) )
       let normal = Some (normal)
       // Returns distance to hit point,
          normal of hit point and material of
          hit point
       (distance, hitPoint, normal, fun () ->
          texture)
     else (None, None, None, fun() -> texture)

else (None, None, None,(fun () -> texture)) //
    No division by zero, thus no hit
```

```
            member this.IsInside p = failwith "Shape is not
               solid!"
    }


let mkTriangle (a : Point) (b : Point) (c : Point) (tex :
   Texture) =
    let bbox = BoundingBox.computeTriangleBbox a b c
    let hitBb = BoundingBox.checkBboxIntersection (bbox)
    Triangle(a,b,c, tex, hitBb, bbox)

let meshTriangle (cornerA, cornerB, cornerC, (tex : Texture),
   (hitBbox : Ray -> bool), (bbox : BoundingBox), (
   isSmoothShade : bool)) =
    {
        new IShape with
            member this.Bbox = bbox
            member this.HitBbox (ray : Ray) = hitBbox ray
            member this.HitFunc (ray : Ray) =

              Triangle sides (u and v) and normal vector
              let us = Point.distance cornerB.point cornerA.
                 point
              let vs = Point.distance cornerC.point cornerA.
                 point
              let aNorm = cornerA.normal
              let bNorm = cornerB.normal
              let cNorm = cornerC.normal
              let dir = Vector.normalise ray.direction
              //Solving equation with 3 unknowns based on
                 cramer's rule

              //First equation with unknown, ax + by + cz = d
              let a = (Point.getX cornerA.point) - (Point.getX
                  cornerB.point)
              let b = (Point.getX cornerA.point) - (Point.getX
                  cornerC.point)
              let c = (Vector.getX dir)
              let d = (Point.getX cornerA.point) - (Point.getX
                  ray.origin)

              //Second equation with unknown, ez + fy + gz = h
              let e = (Point.getY cornerA.point) - (Point.getY
                  cornerB.point)
              let f = (Point.getY cornerA.point) - (Point.getY
                  cornerC.point)
              let g = (Vector.getY dir)
              let h = (Point.getY cornerA.point) - (Point.getY
```

```
    ray.origin)

//Third equation with unknown, ix + jy + kz = l
let i = (Point.getZ cornerA.point) - (Point.getZ
    cornerB.point)
let j = (Point.getZ cornerA.point) - (Point.getZ
    cornerC.point)
let k = (Vector.getZ dir)
let l = (Point.getZ cornerA.point) - (Point.getZ
    ray.origin)

//Finding the unknowns and calculates if there
   is any hit based previous calculations
let D = a*(f*k - g*j) + b*(g*i-e*k) + c*(e*j - f
   *i)

if(D <> 0.0) then
  let beta = (d*(f*k - g*j) + b*(g*l - h*k) + c
     *(h*j - f*l))/D //Corresponds to x
  let gamma = (a*(h*k-g*l) + d*(g*i-e*k) + c*(e*
     l-h*i))/D //Corresponds to y
  let t = (a*(f*l - h*j) + b*(h*i - e*l) + d*(e*
     j-f*i))/D //Corresponds to z
  let alpha = 1.0 - beta - gamma

  if (beta >= 0.0) && (gamma >= 0.0) && ((gamma
     + beta) <= 1.0) // Need sum to be 1 and in
     between 0 and 1
      then
       let distance = Some t
       let hitPoint = Some(Point.move ray.origin
           (t*dir))
       let normal =
         if isSmoothShade then
           let norm = alpha*aNorm + beta*bNorm +
               gamma*cNorm
           Some (getNormal norm ray)
         else
            Some(getNormal (Vector.normalise(
               Vector.crossProduct us vs)) ray)
         // Returns distance to hit point,
            normal of hit point and material of
            hit point
       (distance, hitPoint, normal,
          fun () ->
              let v = alpha * cornerA.v + beta *
                  cornerB.v + gamma * cornerC.v
              let u = alpha * cornerA.u + beta *
                  cornerB.u + gamma * cornerC.u
```

```
                                getTextureColour tex (u,v))
                        else (None, None, None, SceneAssets.black)
                    // No division by zero, thus no hit
                    else (None, None, None, SceneAssets.black)

                member this.IsInside p = failwith "Shape is not
                    solid!"
        }

let mkMeshTriangle (cornerA:Vertex) (cornerB:Vertex) (cornerC:
    Vertex) (tex:Texture) (isSmoothShade : bool) =
        let bbox = BoundingBox.computeTriangleBbox cornerA.point
            cornerB.point cornerC.point
        let hitBb = BoundingBox.checkBboxIntersection (bbox)
        meshTriangle(cornerA,cornerB,cornerC, tex, hitBb, bbox,
            isSmoothShade)


let Plane(point : Point, normalVector : Vector, texture :
    Texture, (hitBbox : Ray -> bool*float*float), bbox :
    BoundingBox) =
        {
            new IShape with
                member this.Bbox = bbox
                member this.HitBbox (ray : Ray) = match hitBbox
                    ray with (isHit,_,_) -> isHit
                member this.HitFunc (ray:Ray) =
                    // point is the direction of axis z we move in
                        and u = Px and v = Py (u,v) coordinates
                    let dir = ray.direction
                    let pz = Point.getZ ray.origin
                    let vz = Vector.getZ dir
                    let dist = -pz/vz;
                    if(dist>=0.0)
                    then
                        let hitPoint = Point.move ray.origin (dist
                            * dir) // Scalar used to get hit point
                            with direction vector from origin to
                           plane
                        (Some dist, Some hitPoint, Some (getNormal
                            normalVector ray),
                            (fun () ->
                                let (hitX,hitY,hitZ) = Point.
                                    getCoord hitPoint
                                let pi = System.Math.PI
                                let u =
                                let u = (hitX - Point.getX point)
                                    % 1.0
                                    if(u<0.0) then u+1.0
```

```
                                    else u
                                let v =
                                    let v = (hitY - Point.getY
                                       point) % 1.0
                                        if(v<0.0) then v+1.0
                                        else v
                                getTextureColour texture (u,v)))
                    else
                         (None, None, None, SceneAssets.black)

                member this.IsInside p = failwith "Shape is not
                   solid!"
}

let mkPlane (t : Texture) =
    let bbox = SceneAssets.mkBbox (Point.mkPoint 0.0 0.0 0.0)
       (Point.mkPoint 0.0 0.0 0.0)
    let hitBb = fun (r : Ray) -> (true,0.0,0.0)
    Plane((Point.mkPoint 0.0 0.0 0.0), (Vector.mkVector 0.0
       0.0 1.0), t, hitBb, bbox)

let Disc(radius : float, texture : Texture, (hitBbox : Ray ->
   bool), bbox) =

    let plane = mkPlane texture
    let center = Point.mkPoint 0.0 0.0 0.0
    {
       new IShape with
           member this.Bbox = bbox
           member this.HitBbox (ray : Ray) = hitBbox ray
           member this.HitFunc (ray:Ray) =
                // Check for intersection between ray and
                   plane
                let hitPoint = plane.HitFunc ray
                let dist, hit, normal,_ = hitPoint
                if  hit.IsSome then
                    let distance = Vector.magnitude (Point.
                       distance center hit.Value) // Distance
                       between center of disc and hit point
                    // Check if hit disc within plane
                    if distance <= radius then
                        (dist, hit, normal,
                            (fun () ->
                                let (hitX,hitY,hitZ) = Point.
                                   getCoord hit.Value
                                let pi = System.Math.PI
                                let u = (hitX+radius)/(2.0*
                                   radius)
                                let v = (hitY+radius)/(2.0*
```

```
                                    radius)
                                getTextureColour texture (u,v)
                                    ))
                        else
                            (None, None, None,SceneAssets.black)
                    else
                        (None, None, None,SceneAssets.black)

            member this.IsInside p = failwith "Shape is not
                solid!"
    }

let mkDisc (center : Point) (radius : float) (tex : Texture) =
    let bbox = BoundingBox.computeDiscBbox radius
    let hitBb = BoundingBox.checkBboxIntersection bbox
    let x,y,z = Point.getCoord center
    transform (Disc(radius, tex, hitBb, bbox)) (translate x y
        z)

let Cylinder(radius : float, height : float, texture : Texture
    , (hitBbox : Ray -> bool), bbox : BoundingBox) =
    {
        new IShape with
            member this.Bbox = bbox
            member this.HitBbox (ray : Ray) = hitBbox ray
            member this.HitFunc (ray : Ray) =
                // Infinite cylinder formula: px^2+pz^2 - r^2
                    = 0
                // Ray hit point formula: p + td with hit
                    point p, direction vector d and distance t
                let dir = Vector.normalise ray.direction
                let checkHitPoint hitPoint =
                let hitPy = Point.getY hitPoint
                //Check for hit within the height of cylinder

                if hitPy <= height/2.0 && hitPy >= -height/2.0
                    then
                    let normalX = (Point.getX hitPoint) /
                        radius
                    let normalY = 0.0
                    let normalZ = (Point.getZ hitPoint) /
                        radius
                    (Some(Vector.magnitude(Point.distance(ray.
                        origin) (hitPoint))), Some hitPoint,
                        Some (getNormal (Vector.mkVector
                            normalX normalY normalZ) ray), //p/
                        r -> normal TODO:// zerothreshhold on
                        normal vector
                                (fun () ->
```

```
                    let (hitX,hitY,hitZ) =
                       Point.getCoord hitPoint
                    let pi = System.Math.PI
                    let nx = hitX /radius
                    let ny = hitY /radius
                    let nz = hitZ /radius
                    let phi' = atan2 nx nz
                    let phi = if phi' < 0.0
                                then phi'+2.0*
                                   pi
                                else phi'
                    let u = phi/(2.0*pi)
                    let v = (hitY/height)+0.5
                    getTextureColour texture (
                       u,v)))
        else (None, None, None,SceneAssets.
           black)
//Formula variables
let px = Point.getX ray.origin
let pz = Point.getZ ray.origin
let (dx,_,dz) = Vector.getCoord dir

//Discriminant variables
let a = ((dx*dx) + (dz*dz))
let b = 2.0*px*dx + 2.0*pz*dz
let c = ((px*px) + (pz*pz) - radius*radius
   )

//Distance
let t = solveSecondDegree a b c
if t.IsSome then
    let hitPoint = Point.move ray.origin (
       t.Value * dir)
    let hit1 = checkHitPoint(hitPoint)
    match hit1 with
    |(None,_,_,_) ->
        let point = Point.move hitPoint  (
           FloatHelper.ZeroThreshold * dir
           )
        let px = Point.getX point
        let pz = Point.getZ point
        let a = ((dx*dx) + (dz*dz))
        let b = 2.0*px*dx + 2.0*pz*dz
        let c = ((px*px) + (pz*pz) -
           radius*radius)
        let t2 = solveSecondDegree a b c
        if t2.IsSome then
            let hitPoint = Point.move
               point (t2.Value * dir)
```

```
                                    let (dist,hitPoint,normal,mat)
                                        = checkHitPoint(hitPoint)
                                    if(normal.IsSome) then
                                        (dist,hitPoint,Some(normal
                                           .Value),mat)
                                    else
                                        (None, None, None,
                                           SceneAssets.black)
                                else
                                    (None, None, None,SceneAssets.
                                       black)
                            |_ -> hit1

                        else (None, None, None,SceneAssets.black)

            member this.IsInside p = failwith "Shape is not
               solid!"
    }

let mkCylinder (radius : float) (height : float) (tex :
   Texture) =
    let bbox = BoundingBox.computeCylinderBbox radius height
    let hitBb = BoundingBox.checkBboxIntersection bbox
    Cylinder(radius, height, tex, hitBb, bbox)

let SolidCylinder(top : IShape, bottom : IShape, openCylinder
   : IShape, height: float, radius: float) =
    let parts = TriangleMesh.mkKDtree [bottom;top;openCylinder
       ]
    {
        new IShape with
            member this.Bbox = openCylinder.Bbox
            member this.HitBbox (ray : Ray) = openCylinder.
               HitBbox ray
            member this.HitFunc (ray : Ray) =
                parts.HitFunc ray


            member this.IsInside p =
                let py = Point.getY p
                if (py<=(height/2.0)) && (py>=(-height/2.0))
                   then
                    (Vector.magnitude (Point.distance p (Point
                       .mkPoint 0.0 py 0.0)))<=radius
                else
                    false
    }
```

```
let mkSolidCylinder radius height tex toptex bottex =

    let cyl = mkCylinder radius height tex

    let transTop = mergeTransformations [(rotateX (Math.PI
       /2.0));(translate  0.0 (height/2.0) 0.0)]
    let top = transform (mkDisc (Point.mkPoint 0.0 0.0 0.0)
       radius toptex) transTop

    let transBot = mergeTransformations [(rotateX (Math.PI
       /2.0));(translate  0.0 (-height/2.0) 0.0)]
    let bottom = transform (mkDisc (Point.mkPoint 0.0 0.0 0.0)
        radius bottex) transBot
    SolidCylinder (top,bottom,cyl,height,radius)

let Rectangle(width : float, height : float, texture : Texture
   , (hitBbox : Ray -> bool), bbox : BoundingBox) =
    let bottomLeft = Point.mkPoint 0.0 0.0 0.0
    let plane = mkPlane texture
    {
        new IShape with
            member this.HitFunc (ray : Ray) =
                let norm = Vector.mkVector 0.0 0.0 0.0

                let topRight = Point.move bottomLeft (Vector.
                   mkVector width height 0.0)
                let (distance,hitPoint,normal,_) = plane.
                   HitFunc ray // Check fi hit hit rectangle
                   within infinite plane
                match hitPoint with
                |Some point ->
                                let x,y,z = Point.getCoord
                                   point

                                // Range variables to check if
                                   hit is within rectangle
                                let xInRange = ((Point.getX
                                   bottomLeft)<= x && x <= (
                                   Point.getX topRight))
                                let yInRange = ((Point.getY
                                   bottomLeft)<= y && y <= (
                                   Point.getY topRight))
                                // If his is within rectangle
                                   return hit
                                if xInRange  && yInRange then (
                                   distance, hitPoint, normal,
                                   fun () ->
                                     let u = x/width
                                     let v = y/height
```

```
                                        getTextureColour texture (
                                            u,v))
                                    else (None, None, None,
                                        SceneAssets.black)
                        |None -> (None, None, None, SceneAssets.black)

                member this.Bbox = bbox
                member this.HitBbox (ray : Ray) = hitBbox ray
                member this.IsInside p = failwith "Shape is not
                    solid!"
        }

let mkRectangle (bottomLeft : Point) (width : float) (height :
    float) (tex : Texture) =
    let bbox = BoundingBox.computeRectangleBbox width height
    let hitBb = BoundingBox.checkBboxIntersection bbox
    let x,y,z = Point.getCoord bottomLeft
    transform (Rectangle(width, height, tex, hitBb, bbox) ) (
        translate x y z)

let Box(lowerCorner : Point, upperCorner : Point, sides:
    IShape list, (hitBbox : Ray -> bool), bbox : BoundingBox) =

    {
        new IShape with
            member this.HitFunc (ray : Ray) =
                getNearestHit sides ray
            member this.Bbox = bbox

            member this.HitBbox (ray : Ray) = hitBbox ray

            member this.IsInside p =
                let (x2,y2,z2) = Point.getCoord upperCorner
                let (x1,y1,z1) = Point.getCoord lowerCorner
                let (px,py,pz) = Point.getCoord p
                x1<=px && px<=x2 && y1 <= py && py <=y2 && z1
                    <= pz && pz <= z2
    }

let mkBox (low : Point) (high : Point) (front : Texture) (back
    : Texture) (top : Texture) (bottom : Texture) (left :
    Texture)  (right : Texture)   =
    let bbox = BoundingBox.computeBoxBbox low high
    let hitBb = BoundingBox.checkBboxIntersection (bbox)
    let toRad n = n*Math.PI/180.0
    let xRot = rotateX (toRad 90.0)
    let yRot = rotateY (toRad -90.0)
    let lx,ly,lz = Point.getCoord low
    let hx,hy,hz = Point.getCoord high
```

```
        let w,h,d = hx-lx,hy-ly,hz-lz

        let back = mkRectangle (Point.mkPoint 0.0 0.0 0.0) w h
           back
        let back = transform back (mergeTransformations [mirrorZ;
           translate lx ly lz])

        let front = mkRectangle (Point.mkPoint 0.0 0.0 0.0) w h
           front
        let front = transform front (mergeTransformations [
           translate lx ly hz])

        let left = mkRectangle (Point.mkPoint 0.0 0.0 0.0) d h
           left
        let left = transform left (mergeTransformations [yRot;
           translate lx ly lz])

        let right = mkRectangle (Point.mkPoint 0.0 0.0 0.0) d h
           right
        let right = transform right (mergeTransformations [mirrorZ
           ; yRot; translate hx ly lz])

        let bottom = mkRectangle (Point.mkPoint 0.0 0.0 0.0) w d
           bottom
        let bottom = transform bottom (mergeTransformations [
           mirrorZ; xRot;translate lx ly lz])

        let top = mkRectangle (Point.mkPoint 0.0 0.0 0.0) w d top
        let top = transform top (mergeTransformations [mirrorZ;
           xRot; translate lx hy lz])

        Box(low, high, [bottom;
                        top;
                        left;
                        back;
                        front;
                        right
                        ], hitBb, bbox)

let group (s1:Shape) (s2:Shape) =
    {
        //Make a new shape by combining two and drawing the
           first one you meet only
        new Shape with
            member this.HitFunc (ray : Ray) =
                    let hit1 = s1.HitFunc ray
                    let hit2 = s2.HitFunc ray
                    let (d1,p1,_,_) = hit1
                    let (d2,p2,_,_) = hit2
```

```
                        match (d1,d2) with
                        |(Some dist1,Some dist2) -> // both hits
                           and are not inside of eachother
                                if(dist1>dist2) then hit2
                                else hit1
                        |(Some dist,_) -> hit1 // first hit hits
                           and are not inside of eachother
                        |(_,Some dist) -> hit2 // second hit hits
                           and are not inside of eachother
                        |(_,_) -> (None,None,None,SceneAssets.
                           black) // there were no hits

                //Make new bounding box for the new shape - take
                   lowest point and highest point of the union
                member this.HitBbox (ray : Ray) = s1.HitBbox ray
                   || s2.HitBbox ray
                member this.Bbox = let s1Lower = s1.Bbox.lower
                                   let s1Higher = s1.Bbox.higher
                                   let s2Lower = s2.Bbox.lower
                                   let s2Higher = s2.Bbox.higher
                                   let newLower = BoundingBox.
                                      findLowestPoint s1Lower
                                      s2Lower
                                   let newHigher = BoundingBox.
                                      findHighestPoint s1Higher
                                      s2Higher
                                   SceneAssets.mkBbox newLower
                                      newHigher
                member this.IsInside p = s1.IsInside p || s2.
                   IsInside p
            }
```

8.6.26 ImplicitSurface.fsi

```
module ImplicitSurface

    type Point = Point.Point
    type Vector = Vector.Vector
    type Ray = RayTracer.Shape.Ray
    type Material = RayTracer.SceneAssets.Material
    type BoundingBox = BoundingBox//.BoundingBox
    type Texture = RayTracer.SceneAssets.Texture
    // Interface used by all shapes with common functions
    type IShape = RayTracer.Shape.IShape

    val mkImplicit: string -> Texture -> IShape
```

8.6.27 ImplicitSurface.fs

```
module ImplicitSurface
```

```
open RayTracer.SceneAssets
open RayTracer.Shape
open ExprParse
open ExprToPoly
open PolynomialFormulas

type Point = Point.Point
type Vector = Vector.Vector
type Ray = RayTracer.Shape.Ray
type Material = RayTracer.SceneAssets.Material
type IShape = RayTracer.Shape.IShape
type BoundingBox = BoundingBox
type Texture = RayTracer.SceneAssets.Texture

//Expression tree from substituting x, y & z with ray
   direction and origin point values.
let implicitSubstExpr (expression:expr) =
    let ex = FAdd(FVar "px", FMult(FVar "t", FVar "dx"))
    let ey = FAdd(FVar "py", FMult(FVar "t", FVar "dy"))
    let ez = FAdd(FVar "pz", FMult(FVar "t", FVar "dz"))

    let subX = subst expression ("x", ex)
    let subY = subst subX ("y", ey)
    subst subY ("z", ez)

//The substituting of direction- & originpoint-variables with
   values from param ray values, in the param expr.
let insertDirOriValues (ray : Ray) (ex:expr) =
    let dex = subst ex ("dx", FNum (Vector.getX ray.direction)
       )
    let dey = subst dex ("dy", FNum (Vector.getY ray.direction
       ))
    let dez = subst dey ("dz", FNum (Vector.getZ ray.direction
       ))
    let pex = subst dez ("px", FNum (Point.getX ray.origin))
    let pey = subst pex ("py", FNum (Point.getY ray.origin))
    subst pey ("pz", FNum (Point.getZ ray.origin))

let findNormal (e:expr) s v =
    let ex = match s with
             | "x" -> subst (deriveExpr (subst (subst e ("y",
                FInt 0)) ("z", FInt 0))) ("x", FNum v)
             | "y" -> subst (deriveExpr (subst (subst e ("x",
                FInt 0)) ("z", FInt 0))) ("y", FNum v)
             | "z" -> subst (deriveExpr (subst (subst e ("y",
                FInt 0)) ("x", FInt 0))) ("z", FNum v)
    getExprValue (simplifyToExpr ex)

//Creates IShape given param s (polynomial). Uses mat yellow
```

```
        texture.
    let mkImplicit (s:string) (texture:Texture) =
        let expression = s |> scan |> insertMult |> parse
        let subZ = implicitSubstExpr expression |> simplifyToExpr
        let dgr = getExprDegree subZ 0

        let implicit =
            {
                new IShape with
                    member this.HitFunc (ray : Ray) =
                        let finalExpr = insertDirOriValues ray
                           subZ
                        let dist = match dgr with
                            | 0 -> None //Not a shape.
                            | 1 -> solveFirstDgr finalExpr
                            | 2 -> let poly = exprToPoly finalExpr
                                   "t"
                                   solveSecondDegreeP poly
                            | _ -> None //not working
//                                       match sturmSeq finalExpr with
//                                          | Some (s, e) ->
   newtonsMethod finalExpr (deriveExpr finalExpr) (s + ((e - s)
   /2.0)) 0 //not implemented fully.
//                                              | None -> None

                        match dist with
                            | Some f when f >= 0.0 ->
                                let hitPoint = Point.move ray.
                                   origin (Vector.multScalar ray.
                                   direction dist.Value)
                                let normal = Vector.mkVector (
                                   findNormal finalExpr "x" (Point
                                   .getX hitPoint)) (findNormal
                                   finalExpr "y" (Point.getY
                                   hitPoint)) (findNormal
                                   finalExpr "z" (Point.getZ
                                   hitPoint))
                                (dist,Some(hitPoint), Some(normal)
                                   ,(fun () -> getTextureColour
                                   texture (0.0,0.0)))
                            | _ -> (None, None, None,(fun () ->
                                getTextureColour texture (0.0,0.0))
                                )

                        member this.Bbox = mkBbox (Point.mkPoint 0.0
                           0.0 0.0) (Point.mkPoint 0.0 0.0 0.0)
                        member this.HitBbox (ray : Ray) = failwith "
                           Not imeplemented."
                        member this.IsInside p = failwith "Shape is
```

```
                        not solid!"
            }
        implicit
```

8.6.28 ConstructiveGeometry.fsi

```
namespace RayTracer

module ConstructiveGeometry =
    type Point = Point.Point
    type Vector = Vector.Vector
    type Ray = Shape.Ray
    type Shape = Shape.IShape
    type Texture = RayTracer.SceneAssets.Texture

    val union : s1:Shape -> s2:Shape -> Shape
    val intersection : s1:Shape -> s2:Shape -> Shape
    val subtraction : s1:Shape -> s2:Shape -> Shape
```

8.6.29 ConstructiveGeomtry.fs

```
namespace RayTracer
open RayTracer.Shape
open RayTracer.SceneAssets

module ConstructiveGeometry =

    type Point = Point.Point
    type Vector = Vector.Vector
    type Ray = Shape.Ray
    type Shape = Shape.IShape
    type Texture = RayTracer.SceneAssets.Texture

    let union (s1:Shape) (s2:Shape) =
            {
            //Make a new shape by combining two and drawing the
               first one you meet only
            new Shape with
                member this.HitFunc (ray : Ray) =
                        let hit1 = s1.HitFunc ray
                        let hit2 = s2.HitFunc ray
                        let (d1,p1,_,_) = hit1
                        let (d2,p2,_,_) = hit2
                        match (d1,d2) with
                        |(Some dist1,Some dist2) when not (s1.
                           IsInside p2.Value) && not (s2.IsInside
                           p1.Value) -> // both hits and are not
                           inside of eachother
                            if(dist1>dist2) then hit2
                            else hit1
```

```
                        |(Some dist,_) when not (s2.IsInside p1.
                           Value)  -> hit1 // first hit hits and
                           are not inside of eachother
                        |(_,Some dist) when not (s1.IsInside p2.
                           Value) -> hit2 // second hit hits and
                           are not inside of eachother
                        |(None,None) -> (None,None,None,
                           SceneAssets.black) // there were no
                           hits
                        |(_,_) -> // there were only hits inside
                           the other shape
                            let d = min d1 d2
                            if(d.IsSome) then
                                let p = Point.move ray.origin ((d.
                                   Value+FloatHelper.ZeroThreshold
                                   )*(Vector.normalise ray.
                                   direction))
                                this.HitFunc (mkRay p ray.
                                   direction)
                            else
                                (None,None,None,SceneAssets.black)

                //Make new bounding box for the new shape - take
                   lowest point and highest point of the union
                member this.HitBbox (ray : Ray) = s1.HitBbox ray
                   || s2.HitBbox ray
                member this.Bbox =
                    let s1Lower = s1.Bbox.lower
                    let s1Higher = s1.Bbox.higher
                    let s2Lower = s2.Bbox.lower
                    let s2Higher = s2.Bbox.higher
                    let newLower = BoundingBox.findLowestPoint
                       s1Lower s2Lower
                    let newHigher = BoundingBox.findHighestPoint
                       s1Higher s2Higher
                    SceneAssets.mkBbox newLower newHigher
                member this.IsInside p = s1.IsInside p || s2.
                   IsInside p
            }

    let intersection (s1:Shape) (s2:Shape) =
            {
            //Make a new shape by taking only the intersection of
               two shapes
            new Shape with
                member this.HitFunc (ray : Ray) =
                    let hit1 = s1.HitFunc ray
                    let hit2 = s2.HitFunc ray
                    let (_,p1,_,_) = hit1
```

```
                let (_,p2,_,_) = hit2
                match (p1,p2) with
                    |(Some p1,Some p2) ->
                        if(s1.IsInside p2) then hit2
                        else if (s2.IsInside p1) then hit1
                        else (None,None,None,SceneAssets.black
                            )
                    |(_,_) -> (None,None,None,SceneAssets.
                       black)

            member this.HitBbox (ray : Ray) =
                s1.HitBbox ray && s2.HitBbox ray
            member this.Bbox =
                let s1Lower = s1.Bbox.lower
                let s1Higher = s1.Bbox.higher
                let s2Lower = s2.Bbox.lower
                let s2Higher = s2.Bbox.higher
                let xPointList = Array.sort [|Point.getX
                   s1Lower;Point.getX s1Higher;Point.getX
                   s2Lower;Point.getX s2Higher|]
                let yPointList = Array.sort [|Point.getY
                   s1Lower;Point.getY s1Higher;Point.getY
                   s2Lower;Point.getY s2Higher|]
                let zPointList = Array.sort [|Point.getZ
                   s1Lower;Point.getZ s1Higher;Point.getZ
                   s2Lower;Point.getZ s2Higher|]
                let middleX = Array.get xPointList 1
                let middleX2 = Array.get xPointList 2
                let middleY = Array.get yPointList 1
                let middleY2 = Array.get yPointList 2
                let middleZ = Array.get zPointList 1
                let middleZ2 = Array.get zPointList 2
                let newPoint = Point.mkPoint middleX middleY
                   middleZ
                let newPoint2 = Point.mkPoint middleX2
                   middleY2 middleZ2
                SceneAssets.mkBbox newPoint newPoint2
            member this.IsInside p = s1.IsInside p && s2.
               IsInside p
        }

let subtraction (s1:Shape) (s2:Shape) : Shape =
    {
        //Make a new shape by subtracting two shapes from
           eachother
        new Shape with
            member this.HitFunc (ray : Ray) =
                let hit1 = s1.HitFunc ray
                let (_,p1,_,_) = hit1
```

```
                    if (p1.IsSome && s2.IsInside(p1.Value)) then
                        let hit2 = s2.HitFunc (mkRay p1.Value (
                           getDirection ray))
                        let (d,p2,n,m) = hit2
                            if(p2.IsSome && s1.IsInside(p2.Value))
                                then
                                (d,p2,Some(n.Value),m)
                            else
                                (None,None,None,SceneAssets.black)
                    else hit1

                member this.HitBbox (ray : Ray) = s1.HitBbox ray
                member this.Bbox = s1.Bbox
                member this.IsInside p = s1.IsInside p && not (s2.
                   IsInside p)
            }
```

8.6.30 Camera.fsi

```
namespace RayTracer
[<Sealed>]

module Camera =
    type Point = Point.Point
    type Vector = Vector.Vector
    type Camera
    type Ray = RayTracer.SceneAssets.Ray
    type LightType = RayTracer.SceneAssets.LightType
    type Shape = Shape.IShape
    type Texture = RayTracer.SceneAssets.Texture

    val mkCamera : pos: Point -> look : Point -> up : Vector ->
       zoom : float ->
        width : float -> height : float -> pwidth : int -> pheight
            : int -> Camera
    val createBitmap : shapes:Shape list -> lights:LightType list
       -> AmbientLight:LightType -> Camera -> max_reflections:int
       -> System.Drawing.Bitmap
```

8.6.31 Camera.fs

```
namespace RayTracer
open System.Drawing
open System.IO
open System
open System.Windows
open System.Windows.Forms
open System.Windows.Media.Imaging
open RayTracer.SceneAssets
open System.Threading
```

```
open System.Threading.Tasks
open RayTracer.Shape

module Camera =
    type Point = Point.Point
    type Vector = Vector.Vector
    type Colour = RayTracer.SceneAssets.Colour
    type Camera = {position : Point; lookat : Point; up : Vector;
       zoom : float; unitHeight: float;
        unitWidth: float; pixelWidth : int; pixelHeight : int;
           image : Bitmap}
    type LightType = RayTracer.SceneAssets.LightType
    type Ray = Shape.Ray
    type Shape = Shape.IShape
    type Texture = RayTracer.SceneAssets.Texture

    let saturate value = min 1.0 (max 0.0 value)




    let mkCamera
        (pos : Point) (look : Point) (up : Vector) (zoom : float)
        (width : float)(height : float)
        (pwidth : int) (pheight : int) : Camera =
            {
                position = pos;
                lookat = look;
                up = up ;
                zoom = zoom ;
                unitWidth = width ;
                unitHeight = height ;
                pixelWidth = pwidth ;
                pixelHeight = pheight;
                image = new Bitmap (pwidth,pheight)
           }


    let findPixelSize (c : Camera) =
        (c.unitWidth/(float) c.pixelWidth, c.unitHeight/(float) c.
           pixelHeight)


    let rec applyLights (lights:LightType list) (s:Shape list) (
       hitPoint:Point) (normal:Vector) (rayVector:Vector) ambient=
        let hitPoint = Point.move hitPoint (FloatHelper.
           ZeroThreshold*normal)
```

```
    List.fold( fun acc light ->
        let lightPos = getLightPosition light; //position of
           light
        let vectorToLight =  (Point.direction hitPoint
           lightPos) //vector from hitpoint to light.
        let shadowRay = mkRay hitPoint vectorToLight //the ray
            from hitpoint to light
        if List.exists(fun (shape:IShape) ->
                            let (d,_,_,_) = shape.HitFunc
                               shadowRay
                            d.IsSome && d.Value>FloatHelper.
                               ZeroThreshold && (Vector.
                               magnitude (Point.distance
                               hitPoint lightPos))>d.Value) s
                               //this is true if it hits shape
                                on the way
        then
            (acc+mkColour 0.0 0.0 0.0)
        else
            let L = Vector.normalise vectorToLight
            let N = Vector.normalise normal
            (acc+(saturate (Vector.dotProduct N L) *
               getLightIntensity light * getLightColour light)
               )
    ) ambient lights

let fireRay ray (shapes:Shape list) =
    let possibleHits = List.map(fun (s:IShape) ->  (s.HitFunc
       ray)) shapes //make the hitFunc for all shapes
    let filteredHits = List.filter(fun ((d: float option),_,_,
       _) -> d.IsSome && d.Value>FloatHelper.ZeroThreshold )
       possibleHits
    let sortedHits = List.sortBy(fun ((d: float option),_,_,_)
        -> d.Value ) filteredHits // sort in ascending order
    match sortedHits with
        |(Some dist,Some hit,Some normal,mat)::_->
                        Some(hit,normal,mat()) //return first
                           hit if any
        |_ ->  None


let rec applyReflections reflections normal (v:Vector) shapes
   startPoint mat (lights:LightType list) ambient: Colour =
    let normal = Vector.normalise normal
    let v = Vector.normalise v
    //The vector going in the opposite direction from the
       vector that hits the
    //shape, in the opposite direction
    let reflectionVector = v - (2.0*(Vector.dotProduct v
```

```
            normal)*normal)
        //Normalized reflectionvector
        let dir = Vector.normalise(reflectionVector)
        let ref = getMaterialReflexivity mat
        let color = getMaterialColour mat
       //Make ray in the direction of reflection vector in order
          to see if there is anything out there
       //that needs to be reflected
        let ray = mkRay (Point.move startPoint (FloatHelper.
            ZeroThreshold*normal)) dir
        match reflections with
            | 0 ->
                (getMaterialColour mat) * (applyLights lights
                   shapes startPoint normal v ambient)
            | _ ->
                let hit = fireRay ray shapes
                //Check if the ray hits anything in the space that
                    needs to be reflected onto this shape
                let light = (applyLights lights shapes startPoint
                   normal v ambient)
                if(hit.IsSome) then
                    let (hitPoint,norm,matr) = hit.Value
                    //take the material of the current shape,
                       blend in with lights, and blend
                    //with the color of the shape that has been
                       hit + ambient light
                    (1.0-ref) * color * light + ref*(
                       applyReflections (reflections-1) norm dir
                       shapes hitPoint matr lights ambient)
                //otherwise, just blend lights with the material
                   and ambient light
                else (1.0-ref) * color*light


let generateRays (shapes:Shape list) (lights:LightType list) (
   ambientLight:LightType) (camera : Camera) (reflections:int)
    : Colour[] =
    let p = camera.position
    let q = camera.lookat
    let u = camera.up
    let z = camera.zoom
    let l = Vector.normalise (Point.direction p q) //
       Normalized direction vector
    let r = Vector.normalise (Vector.crossProduct l u) //
       Normalized right direction vector of the image plane
    let d = Vector.normalise (Vector.crossProduct l r) //
       Normalized down direction vector of the image plane
    let center = Point.move p (z * l) //Center of image plane
    let top = Point.move center ((camera.unitHeight/2.0) * u)
```

```
        //Top center of image plane
    let left = Point.move top ((-camera.unitWidth/2.0) * r) //
        Top left of image plane from the scene's point of view
    let pixelSize = findPixelSize camera
    let W = fst pixelSize
    let H = snd pixelSize
    // Create points in middle of each pixel used for vectors
    let tasks =
    [
     for i in 0..(camera.pixelWidth*camera.pixelHeight-1)
      do yield async
      {
        let a = (float) (i%camera.pixelWidth)
        let b = (float) (i/camera.pixelWidth)
        let point = Point.move left (((a + 0.5)*W) * r)
        let point = Point.move point (((b + 0.5)*H) * d)
        let v = Vector.normalise (Point.direction camera.
            position point)
        let ray = mkRay camera.position v
        let hit = fireRay ray shapes

        if(hit.IsSome) then
            let (hitPoint,normal,mat) = hit.Value
            let ambient = (getLightIntensity ambientLight *
                getLightColour ambientLight)
            let color = applyReflections reflections (Vector.
                normalise normal) v shapes hitPoint mat lights
                ambient
            return color
        else
            return mkColour 0.0 0.0 0.0 //return black if no
                hits
      }
    ]

    Async.RunSynchronously (Async.Parallel tasks)

let generateRaysSync (shapes:Shape list) (lights:LightType
    list) (ambientLight:LightType) (camera : Camera) (
    reflections:int) : Colour[] =
    let p = camera.position
    let q = camera.lookat
    let u = camera.up
    let z = camera.zoom
    let l = Vector.normalise (Point.direction p q) //
        Normalized direction vector
    let r = Vector.normalise (Vector.crossProduct l u) //
        Normalized right direction vector of the image plane
    let d = Vector.normalise (Vector.crossProduct l r) //
```

```
        Normalized down direction vector of the image plane
    let center = Point.move p (z * l) //Center of image plane
    let top = Point.move center ((camera.unitHeight/2.0) * u)
        //Top center of image plane
    let left = Point.move top ((-camera.unitWidth/2.0) * r) //
        Top left of image plane from the scene's point of view
    let pixelSize = findPixelSize camera
    let W = fst pixelSize
    let H = snd pixelSize
    // Create points in middle of each pixel used for vectors
    let tasks = [|0..(camera.pixelWidth*camera.pixelHeight-1)
        |];
    Array.map(fun (i:int) ->
                let a = (float) (i%camera.pixelWidth)
                let b = (float) (i/camera.pixelWidth)
                let point = Point.move left (((a + 0.5)*W) * r
                    )
                let point = Point.move point (((b + 0.5)*H) *
                    d)
                let v = Vector.normalise (Point.direction
                    camera.position point)
                let ray = mkRay camera.position v
                let hit = fireRay ray shapes

                if(hit.IsSome) then
                    let (hitPoint,normal,mat) = hit.Value
                    let ambient = (getLightIntensity
                        ambientLight * getLightColour
                        ambientLight)
                    let color = applyReflections reflections (
                        Vector.normalise normal) v shapes
                        hitPoint mat lights ambient

                    color
                else
                    (mkColour 0.0 0.0 0.0) //return black if
                        no hits
                ) tasks


//creates an image of a bitmap
let createBitmap (s:Shape list) (lights:LightType list) (
    ambientLight:LightType) (cam:Camera) (reflections:int) :
    Bitmap =
    let w = cam.image.Width
    let h = cam.image.Height
    let colors = generateRays s lights ambientLight cam
        reflections;
    for i in 0..(w*h-1) do cam.image.SetPixel((i%w),(i/w),
```

```
            getColor(Array.get colors i))
        cam.image
```

8.6.32 Scene.fsi

```
namespace RayTracer

module Scene =
    type Point = Point.Point
    type Vector = Vector.Vector
    type Camera = Camera.Camera
    type Shape = Shape.IShape
    type LightType = RayTracer.SceneAssets.LightType
    type Scene = {shapes : Shape list; lights : LightType list;
       ambientLight : LightType; camera : Camera; max_reflect :
       int}
    type Ray = SceneAssets.Ray
    type IShape = RayTracer.Shape.IShape


    type Colour = RayTracer.SceneAssets.Colour


    val mkScene : shapes : IShape list -> lights : LightType list
       -> ambientLight:LightType -> Camera -> max_reflect : int ->
        Scene

    val renderToScreen : Scene -> unit
    val renderToFile : Scene -> filename:string -> unit
```

8.6.33 Scene.fs

```
namespace RayTracer

open System.Drawing
open System.IO
open System
open System.Windows
open System.Windows.Forms
open System.Windows.Media.Imaging
open Point
open TriangleMesh
module Scene =
    type IShape = Shape.IShape
    type Camera = RayTracer.Camera.Camera
    type Point = Point.Point
    type Vector = Vector.Vector
    type Ray = SceneAssets.Ray
    type LightType = RayTracer.SceneAssets.LightType
    type Shape = Shape.IShape
```

```
type Colour = RayTracer.SceneAssets.Colour
type Scene = {shapes : Shape list; lights : LightType list;
   ambientLight:LightType; camera : Camera; max_reflect : int}



let mkScene (sha : Shape list) lig amb cam max =
    let (nonKD,shapes) =
        List.fold(fun ((nons: Shape list),(shapes: Shape list)
           ) (shape: Shape) ->
            let isNonKD = Point.distance shape.Bbox.higher
               shape.Bbox.lower
                if (Vector.magnitude isNonKD)<FloatHelper.
                   ZeroThreshold
                then (shape::nons,shapes)
                else (nons,shape::shapes)) ([],[]) sha
    let sceneTree =
        if shapes.Length > 0
            then [TriangleMesh.mkKDtree shapes]
        else []
    let sceneShapes = sceneTree@nonKD
    {shapes = sceneShapes; lights = lig;ambientLight = amb;
       camera = cam; max_reflect = max}

//creates a new form containing the image
let renderToScreen (s:Scene) =
    //create the image with the camera
    let bitmap : Bitmap = RayTracer.Camera.createBitmap s.
       shapes s.lights s.ambientLight s.camera s.max_reflect
    //creating a windows form in the
    let form = new Form(Width = bitmap.Width, Height = bitmap.
       Height, MaximizeBox = false , MinimizeBox = false,
                FormBorderStyle = FormBorderStyle.FixedSingle,
                    Text = "Ray Tracer")
    //getting graphics from form
    let graph : Graphics = form.CreateGraphics()

    //on paint, paint the image received
    let paint e =
        graph.DrawImage(bitmap, 0, 0)
    //add eventhandler
    form.Paint.Add paint
    //display form
    do Application.Run form
    ()

//saves the image as a file
let renderToFile (s:Scene) filename =
    let bitmap : Bitmap = RayTracer.Camera.createBitmap s.
```

```
   shapes s.lights s.ambientLight s.camera s.max_reflect
bitmap.Save(filename)
()
```

8.6.34 PlyParser.fsi

```
namespace RayTracer

(*
    Signature file used to give a short summary of the PLY Parser
       and its public interface
    The module is used to parse PLY files (polygon file format)
       storing graphical objects described as a collection of
       polygons.
    Specifically, we use it to store vertices that are referenced
       by multiple triangle meshes.
    The vertices and faces are used in the MeshBuilder module to
       construct triangle meshes.
    Note that the Ply Parser does not take calculate normals for
       the vertices without, which is instead done in the
       MeshBuilder.
    Also note that the length of the properties are currently hard
        coded to conform with the format of e.g. the "Stanford
       Bunny".
*)

module PlyParser =

    type Point = Point.Point
    type Vector = Vector.Vector
    type Vertex = SceneAssets.Vertex // Vertex composed of a Point
       , normal vector and (u,v) coordinates
    type Face = SceneAssets.Face

    val readLines : fileName:string -> seq<string>
    val (|ParseRegex|_|) : regex:string -> str:string -> string
       list option
    val parseHeader : header:seq<string> -> unit
    val stringToVertex : s:string -> Vertex
    val parseVertices : vertices:seq<string> -> Vertex[]
    val stringToFace : s:string -> Face
    val parseFaces : vertices:seq<string> -> Face[]
    val parsePLY : fileName:string -> Vertex[]*Face[]
```

8.6.35 PlyParser.fs

```
namespace RayTracer

(*
This module is used to parse PLY files (polygon file format)
   storing graphical objects described as a collection of polygons
   .
```

```
Specifically, we use it to store vertices that are referenced by
   multiple triangle meshes.
The vertices and faces are used in the MeshBuilder module to
   construct triangle meshes.
Note that the Ply Parser does not take calculate normals for the
   vertices without, which is instead done in the MeshBuilder.
Also note that the length of the properties are currently hard
   coded to conform with the format of e.g. the "Stanford Bunny".

Typical structure of PLY file:
- Header: describes remainder of file including element type, name
   , amount and a list of properties associated with each element.
- Vertex List: list of triples (x,y,z) for vertices used to build
   different elements in Face list
- Face List: list of other elements composed of vertices
   referenced in the Vertex List

A typical PLY object definition is simply a list of (x,y,z)
   triples for vertices
and a list of faces that are described by indices into the list of
    vertices.
See link: http://paulbourke.net/dataformats/ply/
*)

module PlyParser =

    // Libraries used for parsing including regular expression
       support
    open System
    open System.IO
    open System.Collections.Generic
    open System.Text.RegularExpressions
    open RayTracer.SceneAssets

    // Types used to represent vertices consisting of 3 points and
        face polygons consisting of 3 sides for triangle meshes
    type Vector = Vector.Vector
    type Point = Point.Point
    type Vertex = SceneAssets.Vertex
    type Face = SceneAssets.Face

    // Exception used if malformed lines occur in the PLY file
    exception ParseError of string

    /// Read lines from a PLY file into a sequence of trimmed
       strings.
    /// Takes the filename of the file to be parsed as argument.
    /// Note that the path should be located within the 'Resource'
        folder located in the project files.
```

```
/// Returns a sequence of trimmed lines from the file.
let readLines (fileName : string) =
   seq { use streamReader = new StreamReader (fileName)
         while not streamReader.EndOfStream do
         yield streamReader.ReadLine().Trim() } // Yield each
            trimmed line

// Mutable values used during parsing
// TODO: replace vertexCount and faceCount with a tuple
let mutable vertexCount = 0
let mutable faceCount = 0
let mutable (vertices: seq<Vertex>) = Seq.empty
let mutable (faces: seq<Face>) = Seq.empty
let mutable vertexPropertyMap = Dictionary<string,int>() //
   Property map (name, position) for fast lookups


/// Reinitializes all the mutable values to allow parsing a
   sequence of PLY files
let initialize () =
    vertexCount <- 0
    faceCount <- 0
    vertices <- Seq.empty
    faces <- Seq.empty
    vertexPropertyMap <- Dictionary<string,int>()

// Retrieves the position of a given property in a map
   composed of property names and positions.
// Returns -1 if the property does not exist in the map and
   thus does not have a position.
let getPropertyMapPosition (propertyName : string) (properties
    : Dictionary<string,int>) =
    if (properties.ContainsKey(propertyName))
    then
        let propertyPosition = properties.Item propertyName
        propertyPosition
    else
        -1 // No position

// Retrieves a property from an array of properties extracted
   from property lines in the parseVertices and parseFaces
   methods.
// Returns the property on a given position, else returns 0.0
   which is evaluated in the MeshBuilder for e.g. computing
   normals.
let getProperty (position : int) (properties : string[]) =
    if position = -1
    then 0.0
    else
```

```
        if (position < properties.Length) // Check that
           property position is within amount of properties
        then (float) (Array.get properties position)
        else 0.0

/// ParseRegex parses a regular expression
/// and returns a list of the strings that match each group in
    the regular expression.
/// List.tail is called to eliminate the first element in the
   list,
/// which is the full matched expression, since only the
   matches for each group are wanted.
/// Active patterns always takes one argument but may take
   additional arguments to specialize pattern (e.g. match
   string).
/// This active pattern uses regular expressions to parse
   strings including extra parameter.
/// The method is used to match on strings that use regular
   expressions for various PLY formats
/// in the parseHeader method that customize the general
   ParseRegex active pattern.
/// See link (Active Patterns: https://msdn.microsoft.com/en-
   us/library/dd233248.aspx#Anchor_3)
let (|ParseRegex|_|) regex input =
    let m = Regex(regex).Match(input)
    if m.Success
    then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None



/// Parses a sequences of lines from the header of the PLY
   file omitting the "end_header".
/// Specifically, the amount of vertices and faces are used to
    determine the mutable counter values.
/// Lines that are not Exception 'ParseError' is raised if
   input is not recognized.
/// \d stands for a single character from the set [0..9]
/// [A-Za-z0-9\-]+ is one or more alphanumeric characters (e.g
   . float)
let parseHeader (header: seq<string>) =
    initialize()

    let counter = ref 0
    for line in header do
        match line with
        | ParseRegex "element vertex (\d+)" [amount] -> //
           Amount of vertices
            vertexCount <- int (amount)
```

```
        | ParseRegex "element face (\d+)" [amount] -> //
           Amount of face elements (polygons)
            faceCount <- int (amount)
        | ParseRegex "property ([A-Za-z0-9]+) ([A-Za-z0-9]+)"
           [datatype; name] -> // Vertex properties
              if not (vertexPropertyMap.ContainsKey(name))
                 then do
                vertexPropertyMap.Add(name, counter.Value)
                counter := counter.Value + 1
        | _ -> ()

/// Converts a string representation of a vertex to a vertex
   as represented in the program.
/// Exception if thrown when length of input is less than
   three (no vertex).
/// Note that the property values are determined by the actual
    appearance of them in the given PLY file.
/// If the property is not found a default value of 0 is set
   for further validation in the MeshBuilder.
let stringToVertex (s: string) : Vertex =
   let properties = s.Split[|' '|]
   let x =  getProperty (getPropertyMapPosition "x"
      vertexPropertyMap) properties
   let y =  getProperty (getPropertyMapPosition "y"
      vertexPropertyMap) properties
   let z =  getProperty (getPropertyMapPosition "z"
      vertexPropertyMap) properties
   let nx = getProperty (getPropertyMapPosition "nx"
      vertexPropertyMap) properties
   let ny = getProperty (getPropertyMapPosition "ny"
      vertexPropertyMap) properties
   let nz = getProperty (getPropertyMapPosition "nz"
      vertexPropertyMap) properties
   let u =  getProperty (getPropertyMapPosition "u"
      vertexPropertyMap) properties
   let v =  getProperty (getPropertyMapPosition "v"
      vertexPropertyMap) properties
   let normal = Vector.mkVector nx ny nz
   mkVertex (Point.mkPoint x y z) normal v u

/// Converts a a sequences of strings to an array of vertices
   as represented in the program.
/// Note that the counter is a ref type since you cannot use
   mutable values inside a closure.
let parseVertices (vertices: seq<string>) =
   let counter = ref 0
   let vertexarr =
    Seq.fold (fun array a -> // Accumulator function
        (Array.set array counter.Value (stringToVertex a)) //
```

```
            Set array index value
          counter := counter.Value + 1 array)
          (Array.zeroCreate vertexCount) // Initial array with
            size of vertex amount
          vertices // Sequence to convert
    vertexarr

/// Convert a string PLY representation of a face to a face as
     represented in the program.
/// Throws a 'ParseError' exception when length of inpus is
    less than 3 (polygon should atleast have 3 sides).
/// Note that the property length is hard coded to be three as
     used in the triangle meshes.
/// The property length should not be hard coded but be
    determined by the actual amount of properties in any given
    file.
let stringToFace (s: string) : Face =
    let faceProperties = s.Split[|' '|]
    if (int) (Array.get faceProperties 0) <> 3
    then failwith "Face polygon is not a triangle"
    else
    match faceProperties with
    | face when face.Length < 3 ->
        System.Console.WriteLine(s)
        raise (ParseError("Malformed polygon"))
    | _->
     // Face with triple (x,y,z) reference
     // Position is incremented by 1 to skip amount of face
        sides
         let x =  getProperty ((getPropertyMapPosition "x"
            vertexPropertyMap) + 1) faceProperties
         let y =  getProperty ((getPropertyMapPosition "y"
            vertexPropertyMap) + 1) faceProperties
         let z =  getProperty ((getPropertyMapPosition "z"
            vertexPropertyMap) + 1) faceProperties
         F(int x, int y, int z)

/// Converts a sequence of string containing PLY face
    information to an array of faces as represented in the
    program.
/// Note that the counter is a ref type since you cannot use
    mutable values inside a closure.
let parseFaces (faces: seq<string>) =
      let counter = ref 0
      let facearr =
        Seq.fold (fun array a -> // Accumulator function
        (Array.set array counter.Value (stringToFace a)) //
            Set array index value
        counter := counter.Value + 1 array)
```

```
            (Array.zeroCreate faceCount) // Initial array with
               size of vertex amount
            faces // Sequence to convert
        facearr

    /// Main function of the PLY parser used to parse the
       specified PLY file.
    /// The parsing is done using the three helper functions;
       parseHeader, parseVertices and parseFaces.
    /// Returns an array of vertices and faces used to construct
       triangle meshes in the Mesh Builder.
    let parsePLY fileName =
       let lines = readLines fileName

       // Find header and parse until end ('end_header')
       let bodyPosition = lines |> Seq.findIndex(fun f -> f = "
          end_header")
       let header = lines |> Seq.take bodyPosition
       parseHeader header

       // Find and parse vertices and faces based on appearance in
           line numbers using vertexCount and faceCount
       let vertexPart = lines |> Seq.skip (bodyPosition + 1) |>
          Seq.take vertexCount
       let facePart = (lines |> Seq.skip (bodyPosition +
          vertexCount + 1) |> Seq.take faceCount)
       let vertices = parseVertices vertexPart
       let faces = parseFaces facePart
       (vertices, faces)
```

8.6.36 Program.fs

```
open RayTracer.SceneAssets
open RayTracer.Camera
open RayTracer.Scene
open RayTracer.Shape
open RayTracer.Transformation
open ExprParse
open PolynomialFormulas
open ExprToPoly
open RayTracer.ConstructiveGeometry
open RayTracer.MeshBuilder
open Point
open RayTracer.BoundingBox
open RayTracer.KDTree
open RayTracer.TriangleMesh
open RayTracer.PlyParser
open ImplicitSurface

[<EntryPoint>]
```

```
let main argv =
    //----------SETTINGS: camera, texture and light
    printfn "Setting Camera...";
    let cam = mkCamera (Point.mkPoint 0.0 0.0 22.0) (Point.mkPoint
        0.0 0.0 0.0) (Vector.mkVector 0.0 1.0 0.0)
                        10.0 10.0 10.0 1000 1000 // pos(0,0,0)
                           look(0,100,0) zoom=100 w=1000px/10
                           height=1000px/10
    let ambient = mkAmbientLight(mkColour 1.0 1.0 1.0) 0.1

    printfn "Loading Textures...";
    let texturePath = "../../../Resources/" //Change path to your
       resource
    let woodTexture = loadTexture (texturePath+"wood_tex.jpg")
    let concreteTexture = loadTexture(texturePath+"concrete_tex.
       jpg")
    let cracksTexture = loadTexture(texturePath+"cracks_tex.jpg")
    let stripesTexture = loadTexture(texturePath+"stripes_tex.jpg"
       )
    let dennisTexture = loadTexture(texturePath+"dennis_tex.jpg")
    let meatTexture = loadTexture(texturePath+"meat_tex.jpg")
    let roadTexture = loadTexture(texturePath+"road.jpg")
    let bunnyTexture = loadTexture(texturePath+"bunny_tex.jpg")
    let red = mkMatTexture (mkMaterial (mkColour 1.0 0.0 0.0) 0.5)
    let white = mkMatTexture (mkMaterial (mkColour 1.0 1.0 1.0)
       0.5)
    let tex = mkMatTexture (mkMaterial (mkColour 0.0 0.0 1.0) 0.6
       )
    let tex2 = mkMatTexture (mkMaterial (mkColour 0.0 1.0 0.4) 0.5
        )

    printfn "Placing lights...";
    let l1 = mkLight (Point.mkPoint 10.0 50.0 60.0) (mkColour 1.0
       1.0 1.0) 0.7
    let l2 = mkLight (Point.mkPoint 20.0 20.0 50.0) (mkColour 1.0
       1.0 1.0) 1.0
    let l3 = mkLight (Point.mkPoint 0.0 00.0 -30.0) (mkColour 1.0
       1.0 1.0) 1.0




    //-----------BASE SHAPES ---------------------
    let basePlane = mkPlane woodTexture
    let s = mkSphere (Point.mkPoint 0.0 0.0 0.0) 3.0 tex
    let s3 = mkSphere (Point.mkPoint 0.0 0.0 0.0) 10.0 woodTexture
    let s2 = transform (mkSphere (Point.mkPoint 0.0 0.0 0.0) 5.0
       tex) (translate 1.0 0.0 0.0)
    let t = mkTriangle (Point.mkPoint 0.0 0.0 0.0) (Point.mkPoint
```

```
   10.0 0.0 0.0) (Point.mkPoint 0.0 10.0 0.0) tex
let c = mkCylinder 5.0 20.0 meatTexture

let c2 = transform (mkCylinder 5.0 10.0 tex2) (translate 14.0
   0.0 0.0)
let sc = mkSolidCylinder 5.0 10.0 tex tex tex  //topDisc
   botDisc baseCylinder tex
let sc2 = transform (mkSolidCylinder 5.0 10.0 tex2 tex2 tex2)
   (translate 8.0 0.0 0.0)  //topDisc botDisc baseCylinder tex
let un = intersection sc sc2


//-----------TRIANGLEMESH---------------------
(*
    Add or find additional PLY files in:
    *\Second_Year_Project_Ray_Tracer\RayTracer\Resources
*)

printfn "TriangleMesh is building... "
let filename = "../../../Resources/posche.ply"
let isSmoothShade = true
let TriangleMesh = buildTriangleMesh red isSmoothShade
   filename

let filename = "../../../Resources/bunny.ply"
let bunny = buildTriangleMesh white isSmoothShade filename
//printfn "TriangleMesh done... "



//-----------TRANSFORMATION-------------------
// Rotate needs shape to be in origo (0,0,0) and rays need to
   be in correct direction (e.g. camera needs to be on z
   direction if testing rotateZ)
// Not working, black screen possibly due to direction of rays
   ???
printfn "transforming shapes...";

//----------- SCENE ---------------------
(*
    Guidelines
    modify mkScene to define your scene
    mkSchene [shapes][lights] ambient cam

    eg
    mkScene [ shape 1 ; shape 2 ; shape N ] [ light 1 ; light
       2 ; light N ] ambient cam 1

*)
```

```
    let transformedMesh = transform TriangleMesh (
       mergeTransformations [rotateY (-System.Math.PI/2.0);
       translate 0.0 -8.0 -12.0])
    let bunny = transform bunny (mergeTransformations [rotateY (
       System.Math.PI/4.0); scale 12.0 12.0 12.0;translate 0.0
       -11.0 -5.0])
    let road =  (mkRectangle (mkPoint -50.0 -33.0 -50.0) 100.0
       80.0 roadTexture)
    printfn "Rendering scene...";
    let shapes = [road;transformedMesh;bunny]  //<-- ADD SHAPES
       HERE
    let lights = [l1]
    let reflections = 0

    let scene = mkScene  shapes lights ambient cam reflections
    renderToScreen scene
    printfn "Image rendered"
    1 // return an integer exit code
```

8.6.37 PointTest.fs

```
module PointTest

open System.Threading
open System.Globalization
Thread.CurrentThread.CurrentCulture <- CultureInfo.Invar
   iantCulture

let chk (name,t,r) =
  printf "%s %s\n" name (if t = r then "OK" else "FAILED[t="+(
     string)t+",r="+(string)r+"]")

let p1 = Point.mkPoint 1.0 1.5 2.5
let p2 = Point.mkPoint -2.5 -3.0 1.5
let p3 = Point.mkPoint -2.355432 -3.564523 443.23245

let v1 = Vector.mkVector 1.0 2.5 -3.5

let tests =
  [("Test01",(string)p1,"(1,1.5,2.5)");
   ("Test02",(string)(Point.getX p1),"1");
   ("Test03",(string)(Point.getY p1),"1.5");
   ("Test04",(string)(Point.getZ p1),"2.5");
   ("Test05",(string)(Point.getCoord p1),"(1, 1.5, 2.5)");
   ("Test06",(string)(Point.move p1 v1),"(2,4,-1)");
   ("Test07",(string)(Point.distance p1 p2),"[-3.5,-4.5,-1]");
   ]

let doTest() =
```

```
  printf "PointTest\n"
  List.iter chk tests
```

8.6.38 VectorTest.fs

```
module VectorTest

open System.Threading
open System.Globalization
Thread.CurrentThread.CurrentCulture <- CultureInfo.Invar
   iantCulture

let chk (name,t,r) =
  printf "%s %s\n" name (if t = r then "OK" else "FAILED[t="+(
     string)t+",r="+(string)r+"]")

let v1 = Vector.mkVector 1.0 1.0 1.0
let v2 = Vector.mkVector -1.5 1.5 2.5

let tests =
  [("Test01",(string)v1,"[1,1,1]");
   ("Test02",(string)(-v1),"[-1,-1,-1]");
   ("Test03",(string)(v1+v2),"[-0.5,2.5,3.5]");
   ("Test04",(string)(v1-v2),"[2.5,-0.5,-1.5]");
   ("Test05",(string)(2.5*v2),"[-3.75,3.75,6.25]");
   ("Test06",(string)(v1*v2),"2.5");
   ("Test07",(string)(Vector.getX v2),"-1.5");
   ("Test08",(string)(Vector.getY v2),"1.5");
   ("Test09",(string)(Vector.getZ v2),"2.5");
   ("Test10",(string)(Vector.getCoord v2),"(-1.5, 1.5, 2.5)");
   ("Test11",(string)(Vector.multScalar v2 2.0),"[-3,3,5]");
   ("Test12",(string)(System.Math.Round(Vector.magnitude v2,3)),"
      3.279");
   ("Test13",(string)(System.Math.Round(Vector.dotProduct v1 v2,3)
      ),"2.5");
   ("Test14",(string)(Vector.crossProduct v1 v2),"[1,-4,3]");
   ]

let doTest() =
  printf "VectorTest\n"
  List.iter chk tests
```

8.6.39 ExprParseTest.fs

```
module ExprParseTest

open ExprParse

let tests =
  [("TestScan01","2*3",[Int 2; Mul; Int 3]);
```

```
    ("TestScan02","-2*3",[Int -2; Mul; Int 3]);
    ("TestScan03","-2*-3",[Int -2; Mul; Int -3]);
    ("TestScan04","2+3*4",[Int 2; Add; Int 3; Mul; Int 4]);
    ("TestScan05","2+-3*-4",[Int 2; Add; Int -3; Mul; Int -4]);
    ("TestScan06","(2.0+3)*4",[Lpar; Float 2.0; Add; Int 3; Rpar;
       Mul; Int 4]);
    ("TestScan07","-1(2+3)",[Int -1; Lpar; Int 2; Add; Int 3; Rpar
       ]);
    ("TestScan08","2^4",[Int 2; Pwr; Int 4]);
    ("TestScan09","x^2",[Var "x"; Pwr; Int 2]);
    ("TestScan10","x2",[Var "x2"]);
    ("TestScan11","2x",[Int 2; Var "x"]);
    ("TestScan12","-1x",[Int -1; Var "x"]);
    ("TestScan13","2 x",[Int 2; Var "x"]);
    ("TestScan14","2 x y",[Int 2; Var "x"; Var "y"]);
    ("TestScan15","2xy",[Int 2; Var "xy"]);
    ("TestScan16","2x(2x)",[Int 2; Var "x"; Lpar; Int 2; Var "x";
       Rpar]);
    ("TestScan17","2 x 2 y(2 x(-2))",[Int 2; Var "x"; Int 2; Var "y
       "; Lpar; Int 2; Var "x"; Lpar; Int -2; Rpar; Rpar]);
    ("TestScan18","2x^2*2y^2",[Int 2; Var "x"; Pwr; Int 2; Mul; Int
        2; Var "y"; Pwr; Int 2]);
    ("TestScan19","2x^2(2y^2)",[Int 2; Var "x"; Pwr; Int 2; Lpar;
       Int 2; Var "y"; Pwr; Int 2; Rpar])]

let testsFail =
  [("TestFail01","-(2+3)");
   ("TestFail02","-x")]

let testsInsertMult =
  [("TestInsertMult01","2*3",[Int 2; Mul; Int 3]);
   ("TestInsertMult02","-2*3",[Int -2; Mul; Int 3]);
   ("TestInsertMult03","-2*-3",[Int -2; Mul; Int -3]);
   ("TestInsertMult04","2+3*4",[Int 2; Add; Int 3; Mul; Int 4]);
   ("TestInsertMult05","2+-3*-4",[Int 2; Add; Int -3; Mul; Int
      -4]);
   ("TestInsertMult06","(2.0+3)*4",[Lpar; Float 2.0; Add; Int 3;
      Rpar; Mul; Int 4]);
   ("TestInsertMult07","-1(2+3)",[Int -1; Mul; Lpar; Int 2; Add;
      Int 3; Rpar]);
   ("TestInsertMult08","2^4",[Int 2; Pwr; Int 4]);
   ("TestInsertMult09","x^2",[Var "x"; Pwr; Int 2]);
   ("TestInsertMult10","x2",[Var "x2"]);
   ("TestInsertMult11","2x",[Int 2; Mul; Var "x"]);
   ("TestInsertMult12","-1x",[Int -1; Mul; Var "x"]);
   ("TestInsertMult13","2 x",[Int 2; Mul; Var "x"]);
   ("TestInsertMult14","2 x y",[Int 2; Mul; Var "x"; Mul; Var "y"
      ]);
   ("TestInsertMult15","2xy",[Int 2; Mul; Var "xy"]);
```

```
    ("TestInsertMult16","2x(2x)",[Int 2; Mul; Var "x"; Mul; Lpar;
       Int 2; Mul; Var "x"; Rpar]);
    ("TestInsertMult17","2 x 2 y(2 x(-2))",[Int 2; Mul; Var "x";
       Mul; Int 2; Mul; Var "y"; Mul; Lpar; Int 2; Mul; Var "x";
       Mul; Lpar; Int -2; Rpar; Rpar]);
    ("TestInsertMult18","2x^2*2y^2",[Int 2; Mul; Var "x"; Pwr; Int
       2; Mul; Int 2; Mul; Var "y"; Pwr; Int 2]);
    ("TestInsertMult19","2x^2(2y^2)",[Int 2; Mul; Var "x"; Pwr; Int
        2; Mul; Lpar; Int 2; Mul; Var "y"; Pwr; Int 2; Rpar])]

let testsParse =
  [("TestParse01","2*3",FMult (FNum 2.0,FNum 3.0));
   ("TestParse02","-2*3",FMult (FNum -2.0,FNum 3.0));
   ("TestParse03","-2*-3",FMult (FNum -2.0,FNum -3.0));
   ("TestParse04","2+3*4",FAdd (FNum 2.0,FMult (FNum 3.0,FNum 4.0)
      ));
   ("TestParse05","2+-3*-4",FAdd (FNum 2.0,FMult (FNum -3.0,FNum
      -4.0)));
   ("TestParse06","(2.0+3)*4",FMult (FAdd (FNum 2.0,FNum 3.0),FNum
       4.0));
   ("TestParse07","-1(2+3)",FMult (FNum -1.0,FAdd (FNum 2.0,FNum
      3.0)));
   ("TestParse08","2^4",FExponent (FNum 2.0,4));
   ("TestParse09","x^2",FExponent (FVar "x",2));
   ("TestParse10","x2",FVar "x2");
   ("TestParse11","2x",FMult (FNum 2.0,FVar "x"));
   ("TestParse12","-1x",FMult (FNum -1.0,FVar "x"));
   ("TestParse13","2 x",FMult (FNum 2.0,FVar "x"));
   ("TestParse14","2 x y",FMult (FMult (FNum 2.0,FVar "x"),FVar "y
      "));
   ("TestParse15","2xy",FMult (FNum 2.0,FVar "xy"));
   ("TestParse16","2x(2x)",FMult (FMult (FNum 2.0,FVar "x"),FMult
      (FNum 2.0,FVar "x")));
   ("TestParse17","2 x 2 y(2 x(-2))",FMult (FMult (FMult (FMult (
      FNum 2.0,FVar "x"),FNum 2.0),FVar "y"), FMult (FMult (FNum
      2.0,FVar "x"),FNum -2.0)));
   ("TestParse18","2x^2*2y^2",FMult (FMult (FMult (FNum 2.0,
      FExponent (FVar "x",2)),FNum 2.0), FExponent (FVar "y",2)));
   ("TestParse19","2x^2(2y^2)",FMult (FMult (FNum 2.0,FExponent (
      FVar "x",2)), FMult (FNum 2.0,FExponent (FVar "y",2))))
   ]

let doTest() =
  let chk (name,t,r) =
    printf "%s %s\n" name (if t = r then "OK" else "FAILED")
  let chkFail (name,fn) =
    try
      let _ = fn()
      printf "%s FAILED\n" name
```

```
    with
      | _ -> printf "%s OK\n" name
  printf "ExprParseTest\n"
  List.iter chk (List.map (fun (n,s,r) -> (n,scan s,r)) tests)
  List.iter chkFail (List.map (fun (n,s) -> (n,fun() -> scan s))
     testsFail)
  List.iter chk (List.map (fun (n,s,r) -> (n,(scan >> insertMult)
     s,r)) testsInsertMult)
  List.iter chk (List.map (fun (n,s,r) -> (n,(scan >> insertMult
     >> parse) s,r)) testsParse)
```

8.6.40 ExprToPolyTest.fs

```
module ExprToPolyTest

open ExprParse
open ExprToPoly

let chk (name,t,r) =
  printf "%s %s\n" name (if t = r then "OK" else "FAILED[t="+(
     string)t+",r="+(string)r+"]")

let chk' (name,t,r) =
  printf "%s %s\n" name (if t = r then "OK" else "FAILED")

let test01() =
  let ex01 =
    FAdd(FAdd(FAdd(FExponent(FVar "x",2),
                   FExponent(FVar "y",2)),
              FExponent(FVar "z",2)),
         FMult(FNum -1.0,FNum 1.0))
  let ex = FAdd(FVar "px", FMult(FVar "t",FVar "dx"))
  let ey = FAdd(FVar "py", FMult(FVar "t",FVar "dy"))
  let ez = FAdd(FVar "pz", FMult(FVar "t",FVar "dz"))
  let ex01Subst = List.fold subst ex01 [("x",ex);("y",ey);("z",ez)
     ]
  let p_d = exprToPoly ex01Subst "t"
  ("TestPoly01",ppPoly "t" p_d,"(pz^2+py^2+px^2+-1)+t(2*dz*pz+2*dy
     *py+2*dx*px)+t^2(dz^2+dy^2+dx^2)")

let test02() =
  let e = FAdd(FAdd(FAdd(FExponent(FVar "x",2),
                         FExponent(FVar "y",2)),
                    FExponent(FVar "z",2)),
               FAdd(FNum -1.0,FNum 1.0))
  let p_x = exprToPoly e "x"
  ("TestPoly02",ppPoly "x" p_x,"(z^2+y^2)+x^2")

let test03() =
```

```
  let plane = FAdd (FMult (FVar "a", FVar "x"), FAdd (FMult (FVar
     "b", FVar "y"), FAdd (FMult (FVar "c", FVar "z"), FVar "d")))
  let planeStr = parseStr "a*x+b*y+c*z+d"
  let ex = FAdd (FVar "ox", FMult (FVar "t", FVar "dx"))
  let ey = FAdd (FVar "oy", FMult (FVar "t", FVar "dy"))
  let ez = FAdd (FVar "oz", FMult (FVar "t", FVar "dz"))
  let planeSubst = List.fold subst plane [("x",ex);("y",ey);("z",
     ez)]
  let planeStrSubst = List.fold subst planeStr [("x",ex);("y",ey)
     ;("z",ez)]
  let plane_d = exprToPoly planeSubst "t"
  let planeStr_d = exprToPoly planeStrSubst "t"

  let circle = FAdd (FExponent (FVar "x", 2) ,
                     FAdd (FExponent (FVar "y", 2),
                           FAdd (FExponent (FVar "z", 2),
                                 FMult (FNum -1.0, FExponent (FVar
                                     "r", 2)))))
  let circleStr = parseStr "x^2+y^2+z^2+-1r^2"
  let circleSubst = List.fold subst circle [("x",ex);("y",ey);("z"
     ,ez)]
  let circleStrSubst = List.fold subst circleStr [("x",ex);("y",ey
     );("z",ez)]
  let circle_d = exprToPoly circleSubst "t"
  let circleStr_d = exprToPoly circleSubst "t"
  [("TestPoly03",ppPoly "t" plane_d,"(d+c*oz+b*oy+a*ox)+t(c*dz+b*
     dy+a*dx)");
   ("TestPoly04",ppPoly "t" planeStr_d,"(d+c*oz+b*oy+a*ox)+t(c*dz+
      b*dy+a*dx)");
   ("TestPoly05",ppPoly "t" circle_d,"(oz^2+oy^2+ox^2+-1*r^2)+t(2*
      dz*oz+2*dy*oy+2*dx*ox)+t^2(dz^2+dy^2+dx^2)");
   ("TestPoly06",ppPoly "t" circleStr_d,"(oz^2+oy^2+ox^2+-1*r^2)+t
      (2*dz*oz+2*dy*oy+2*dx*ox)+t^2(dz^2+dy^2+dx^2)")]


let test04() =
  let es = List.map (fun n -> FExponent (FAdd (FVar "a", FVar "b")
     , n)) [1 .. 5]
  let esStr = List.map (fun n -> parseStr ("(a+b)^"+((string)n)))
     [1 .. 5]
  let es2 = List.map (fun n -> FExponent (FAdd (FVar "a", FAdd(
     FVar "b", FVar "c")),n)) [1 .. 5]
  let es2Str = List.map (fun n -> parseStr ("(a+b+c)^"+((string)n)
     )) [1 .. 5]
  let rs s = [("TestSimplify01"+s,"a+b");
              ("TestSimplify02"+s,"2*a*b+a^2+b^2");
              ("TestSimplify03"+s,"3*a*b^2+3*a^2*b+a^3+b^3");
              ("TestSimplify04"+s,"4*a*b^3+6*a^2*b^2+4*a^3*b+a^4+b
                 ^4");
```

```
                ("TestSimplify05"+s,"5*a*b^4+10*a^2*b^3+10*a^3*b
                   ^2+5*a^4*b+a^5+b^5")]
  let rs2 s = [("TestSimplify06"+s,"a+b+c");
               ("TestSimplify07"+s,"2*a*b+2*a*c+a^2+2*b*c+b^2+c^2"
                  );
               ("TestSimplify08"+s,"6*a*b*c+3*a*b^2+3*a*c^2+3*a^2*
                  b+3*a^2*c+a^3+3*b*c^2+3*b^2*c+b^3+c^3");
               ("TestSimplify09"+s,"12*a*b*c^2+12*a*b^2*c+4*a*b
                  ^3+4*a*c^3+12*a^2*b*c+6*a^2*b^2+6*a^2*c^2+4*a^3*
                  b+4*a^3*c+a^4+4*b*c^3+6*b^2*c^2+4*b^3*c+b^4+c^4"
                  );
               ("TestSimplify10"+s,"20*a*b*c^3+30*a*b^2*c^2+20*a*b
                  ^3*c+5*a*b^4+5*a*c^4+30*a^2*b*c^2+30*a^2*b^2*c
                  +10*a^2*b^3+10*a^2*c^3+20*a^3*b*c+10*a^3*b^2+10*
                  a^3*c^2+5*a^4*b+5*a^4*c+a^5+5*b*c^4+10*b^2*c
                  ^3+10*b^3*c^2+5*b^4*c+b^5+c^5")]
  let allEs = List.zip (rs "") es
  let allEsStr = List.zip (rs "Parse") esStr
  let allEs2 = List.zip (rs2 "") es2
  let allEs2Str = List.zip (rs2 "Parse") es2Str
  let genChk ((n,r),e) = (n,(exprToSimpleExpr >> ppSimpleExpr) e,r
     )
  List.map genChk (allEs @ allEsStr @ allEs2 @ allEs2Str)

let (t1,t2) =
  let sphere = FAdd(FAdd(FAdd(FExponent(FVar "x",2),
                              FExponent(FVar "y",2)),
                         FExponent(FVar "z",2)),
                    FMult(FNum -1.0,FVar "R"))
  let ex = FAdd(FVar "px", FMult(FVar "t",FVar "dx"))
  let ey = FAdd(FVar "py", FMult(FVar "t",FVar "dy"))
  let ez = FAdd(FVar "pz", FMult(FVar "t",FVar "dz"))
  let eR = FNum -1.0
  let sphereSubst = List.fold subst sphere [("x",ex);("y",ey);("z"
     ,ez);("R",eR)]
  let sphereSE = exprToSimpleExpr sphereSubst
  (("TestSphere01",sphereSubst,FAdd (FAdd (FAdd (FExponent (FAdd (
     FVar "px",FMult (FVar "t",FVar "dx")),2),
        FExponent (FAdd (FVar "py",FMult (FVar "t",FVar "dy")),2))
           ,
        FExponent (FAdd (FVar "pz",FMult (FVar "t",FVar "dz")),2))
           ,
        FMult (FNum -1.0,FNum -1.0))),
   ("TestSphere02",sphereSE,SE [[ANum 1.0];
        [ANum 2.0; AExponent ("dx",1); AExponent ("px",1);
           AExponent ("t",1)];
        [AExponent ("dx",2); AExponent ("t",2)];
        [ANum 2.0; AExponent ("dy",1); AExponent ("py",1);
           AExponent ("t",1)];
```

```
            [AExponent ("dy",2); AExponent ("t",2)];
            [ANum 2.0; AExponent ("dz",1); AExponent ("pz",1);
               AExponent ("t",1)];
            [AExponent ("dz",2); AExponent ("t",2)]; [AExponent ("px"
               ,2)];
            [AExponent ("py",2)]; [AExponent ("pz",2)]]))

    let t3 = ("TestSE01",simplifyAtomGroup [AExponent ("px",1);
       AExponent ("px",2); ANum -2.0; ANum -2.0],[ANum 4.0; AExponent
       ("px",3)])
    let t4 = ("TestSE02",
            simplifySimpleExpr (SE [[ANum 3.0];[ANum 4.0];[AExponent("
               x",2);AExponent("y",3)];[AExponent("x",2); AExponent("y
               ",3)]]),
            SE [[ANum 7.0]; [ANum 2.0; AExponent ("x",2); AExponent ("
               y",3)]])
    let t5 = ("TestSimplify11", (parseStr >> exprToSimpleExpr >>
       simplifySimpleExpr >> ppSimpleExpr) "a*b+b*a", "2*a*b")

    let doTest() =
      printf "ExprToPoly Test\n"
      List.iter chk (test01() ::
                     test02() ::
                     test03() @ [t5] @ test04())
      chk' t1
      chk' t2
      chk' t3
      chk' t4
```

8.6.41 HitFunctionTest.fs

```
    module HitFunctionTest

    open System

    module HitFunctionTests =

        open Xunit
        open Swensen.Unquote
        open RayTracer //.HitFunction
        open PolynomialFormulas

        module secondDegreeUnitTest =

            [<Theory>]
            [<InlineData(3.0, 50.0, 10.0, -16.46420728)>]
            let "hit function returns correct result when discriminant
                is positive with two hits" (a: float) (b: float) (c:
               float) (result: float option) =
                let actualSolution = solveSecondDegree a b c
```

```
            let expectedResult = result
            test <@ expectedResult = actualSolution @>
```

8.6.42 Vector.fsi

```
module Vector
[<Sealed>]
type Vector =
  static member ( ~- ) : Vector -> Vector
  static member ( + ) : Vector * Vector -> Vector
  static member ( - ) : Vector * Vector -> Vector
  static member ( - ) : Vector * float -> Vector
  static member ( * ) : float * Vector -> Vector
  static member ( * ) : Vector * Vector -> float

val zero : Vector
val mkVector : x:float -> y:float -> z:float -> Vector
val getX : Vector -> float
val getY : Vector -> float
val getZ : Vector -> float
val getCoord: Vector -> float * float * float
val multScalar : Vector -> s:float -> Vector
val magnitude : Vector -> float
val dotProduct : Vector -> Vector -> float
val crossProduct : Vector -> Vector -> Vector
val normalise : Vector -> Vector
```

8.6.43 Vector.fs

```
module Vector

type Vector =
    V of float * float * float
        override v.ToString() =
            match v with
                V(x,y,z) -> "["+x.ToString()+","+y.ToString()+","+
                   z.ToString()+"]"

let mkVector (x:float) (y:float) (z:float) = V(x,y,z)

let zero = V(0.0,0.0,0.0)

let getX (V(x,_,_)) = x

let getY (V(_,y,_)) = y

let getZ (V(_,_,z)) = z

let getCoord (V(x,y,z)) = (x,y,z)
```

```
let multScalar (V(x,y,z)) s = V(x*s,y*s,z*s)

let magnitude (V(x,y,z)) = System.Math.Sqrt(x*x + y*y + z*z)

let dotProduct (V(x1,y1,z1)) (V(x2,y2,z2)) = (x1*x2)+(y1*y2)+(z1*
   z2)

let crossProduct (V(x1,y1,z1)) (V(x2,y2,z2)) = V(y1*z2-z1*y2,z1*x2
   -x1*z2,x1*y2-y1*x2)

let normalise (V(x,y,z) as v) = let len = (magnitude v)
                                if(len=0.0) then
                                    V(0.0,0.0,0.0)
                                else
                                    V(x/len, y/len, z/len)

type Vector with
  static member ( ~- ) (V(x,y,z)) = V(-x,-y,-z)
  static member ( + ) (V(ux,uy,uz),V(vx,vy,vz)) = V(ux+vx,uy+vy,uz
     +vz)
  static member ( - ) (V(ux,uy,uz),V(vx,vy,vz)) = V(ux-vx,uy-vy,uz
     -vz)
  static member ( - ) (V(x, y, z), s: float) = V(x-s, y-s, z-s)
  static member ( * ) (s:float, v:Vector) = (multScalar v s)
  static member ( * ) (v:Vector,u:Vector) = (dotProduct v u)
```