The *M-DART©* Project

The Development of a Ray Tracing System in the Ada Programming Language

By Marinko Laban

Contents

[I. FOREWORD 4](#_Toc427051750)

[II. PREFACE 5](#_Toc427051751)

[1 ABOUT THE PROJECT 6](#_Toc427051752)

[2 ADA USAGE 7](#_Toc427051753)

[2.1 Standard Types 7](#_Toc427051754)

[3 A SIMPLE GRAPHICS LAYER 8](#_Toc427051755)

[4 MATHEMATICAL FOUNDATIONS AND NOTATIONS 9](#_Toc427051756)

[4.1 Coordinate Systems 9](#_Toc427051757)

[4.2 Basic Mathematics in 3D 9](#_Toc427051758)

[4.2.1 Points in 3D 10](#_Toc427051759)

[4.2.2 Vectors in 3D 11](#_Toc427051760)

[4.2.3 Normals in 3D 11](#_Toc427051761)

[4.2.4 Mathematical Operations on Points, Vectors and Normals 11](#_Toc427051762)

[4.3 Affine Transformations 13](#_Toc427051763)

[4.3.1 Matrices 13](#_Toc427051764)

[4.3.2 Translation 13](#_Toc427051765)

[4.3.3 Scaling 13](#_Toc427051766)

[4.3.4 Rotation 13](#_Toc427051767)

[4.3.5 Skewing 14](#_Toc427051768)

[4.3.6 Combining Transformations 14](#_Toc427051769)

[4.3.7 Transforming Points, Vectors and Normals 14](#_Toc427051770)

[4.4 Other Mathematical Notations 14](#_Toc427051771)

[5 Light and Colors 15](#_Toc427051772)

[6 Ray Tracing Principals 16](#_Toc427051773)

[7 A Simple Sphere Tracer 17](#_Toc427051774)

[7.1 A Simple Reflection Model 17](#_Toc427051775)

[7.2 Rendering the Unit Sphere 17](#_Toc427051776)

[7.3 Transforming the Unit Sphere 17](#_Toc427051777)

[8 Intersecting 3D Objects 18](#_Toc427051778)

[8.1 Generic Approach 18](#_Toc427051779)

[8.2 Sphere 18](#_Toc427051780)

[8.3 Box 19](#_Toc427051781)

[8.4 Cylinder 19](#_Toc427051782)

[8.5 Cone 19](#_Toc427051783)

[8.6 Torus 19](#_Toc427051784)

[8.7 Triangle Mesh 19](#_Toc427051785)

[8.8 CSG Object 19](#_Toc427051786)

[9 Ray Tracing as a Sampling Process 20](#_Toc427051787)

[10 The Rendering Equation 21](#_Toc427051788)

[11 Diffuse Reflection 22](#_Toc427051789)

[12 Specular Reflection 23](#_Toc427051790)

[13 Texture Mapping 24](#_Toc427051791)

[14 Acceleration Methods 25](#_Toc427051792)

[15 Dripping Objects 26](#_Toc427051793)

[16 Radiosity 27](#_Toc427051794)

# FOREWORD

This…

# PREFACE

This..

# ABOUT THE PROJECT

All the code in this book is developed using a simple, but effective project approach. This approach is well-suited for small programming teams, but, as I did the project largely by myself, it proved to be valuable for a "single person project team" as well.

When I started this project, I found myself facing several problems before writing the first line of code:

1. The scope of the project
2. The selection of a project approach
3. The selection of a programming language
4. The selection of a programming environment

Ad 1: The scope of the actual functionality is something that was not very clear at the beginning of the project. However, a few things cropped up immediately. Since I had to do everything in my spare time, I could not cover all aspects of ray tracing and implement all the nice features and functions that have been developed around the world. On the other hand, the scope should not be too small; otherwise the whole project would lose its relevance and would not address the right audience.

Ad 2: In order to stay focused and manage the scope of the project, a project approach would have to be selected that would be simple and effective. My experience with managing larger implementation projects and teams in my professional career helped me selecting a method that is based upon the Agile method. As a single person team, I could use some of its components to keep me focused.

# ADA USAGE

This book is not a text book about programming in Ada. However, to understand the code examples being presented here, some background about Ada is imperative to understand the examples.

The ray tracer presented in this book does not make use of all aspects of Ada, but abstract data types (ADT), object-oriented programming (OOP) and primitive operations are definitely important. On the other hand, concessions are sometimes made to OOP concepts and ADT's to make the code easier to understand and/or perform better.

At the time of writing this book, all coding has been done with GPS & GNAT-GPL 2012, but I expect the code will be portable to an Ada 2005 environment with little problems. I do not expect the code to be portable to Ada 95 or Ada 83.

## Standard Types

The ray tracer is implemented through the use base types, and these are defined in the package ***StdTypes***. This package should be modified if your implementation of Ada 2005 has different sizes or accuracy on the types being used. It should only be needed to change the types and constants in here, as I put a lot of effort in making the rest of the packages independent of the operating system, hardware and compiler.

Check your compiler documentation before modifying the types. Also note that I have implemented type-specific *Put* functions in this package, to allow some basic printing functions.

type Integer\_16 is new Short\_Integer;

type Integer\_32 is new Integer;

type Integer\_64 is new Long\_Long\_Integer;

type Float\_32 is new Float;

type Float\_64 is new Long\_Float;

type Float\_96 is new Long\_Long\_Float;

subtype SMALL\_INTEGER is Integer\_16;

subtype LARGE\_INTEGER is Integer\_32;

subtype HUGE\_INTEGER is Integer\_64;

subtype SMALL\_FLOAT is Float\_32;

subtype LARGE\_FLOAT is Float\_64;

subtype HUGE\_FLOAT is Float\_96;

MyPI: constant LARGE\_FLOAT := Ada.Numerics.Pi;

MyPI2: constant LARGE\_FLOAT := 2.0 \* MyPI;

MyPI\_Inv: constant LARGE\_FLOAT := 1.0 / MyPI;

MyPI2\_Inv: constant LARGE\_FLOAT := 1.0 / MyPI2;

MyPI\_Half: constant LARGE\_FLOAT := MyPI / 2.0;

MyPI\_Third: constant LARGE\_FLOAT := MyPI / 3.0;

MyPI\_Quart: constant LARGE\_FLOAT := MyPI / 4.0;

# A SIMPLE GRAPHICS LAYER

This

# MATHEMATICAL FOUNDATIONS AND NOTATIONS

This book is about ray tracing, a field in the area of Computer Graphics. An intrinsic part of this field is the mathematics around 3-dimensional objects and signal theory, as in electro-magnetic phenomena simulations such as light. It is impossible to address these topics without going into any mathematics and physics. I have tried to discuss the required mathematical and physical theory in each section, but some foundation to do that has to be established first. This foundation is laid out in this chapter, together with some implementation details. Please note that this book is not a text book on mathematics or physics. Some basic (high school?) level of knowledge is assumed here.

## Coordinate Systems

In ray tracing, we are using coordinates to indicate positions, directions and alike. Before we can actually do that, we will have to agree on a Coordinate System. Although this may sound complex, it is actually quite arbitrary what system we agree on. It is a convention to use a *right-handed Cartesian coordinate system*, as this is commonly used in many fields. A Cartesian coordinate system has the following characteristics:

* Each dimension has its own axis
* Each pair of axis is perpendicular
* Distances along any axis are measured in the same unit length

We commonly use x, y and z to indicate the three axis in 3D space (and x and y for 2D space), but this may vary from field to field. We also need to agree to which direction the z-axis is pointing when looking at it from the (x, y) plane. In this case, we want to have the z-axis point towards us, if the x-axis is pointing to the right and the y-axis is pointing up. This is called a right-handed coordinate system and can be depicted as follows:

In case of a left-handed coordinate system, the z-axis would point away from us. See also <http://en.wikipedia.org/wiki/Cartesian_coordinate_system> for more information on coordinate systems.

## Basic Mathematics in 3D

In our ray tracer, we will have to manipulate points, vectors and normal vectors in 3D space. A normal vector is also called simply a normal. In fact, we will also have to manipulate points in 2D space, but more about that later.

There is a distinct theoretical difference between points, vectors and normals, but a lot of ray tracer code actually implements these with a single data type. Quite often this is an array or record, containing 3 floating point variables, representing the X, Y and Z coordinate of the point, vector or normal. However, in a true Object-Oriented Programming approach, you could argue that each would need its own type and operations, as in mathematics they are not the same either. Points, for instance, actually point to a fixed location, while a vector indicates a distance and a direction, but not a fixed point. We will also see that normals and vectors also behave differently under transformations.

So the decision to make points, vectors and normals different types with different operations would prevent you (to some extend) from writing bad code, albeit at the cost of writing more code and some performance penalty. However, I think it is more important to write maintainable code that also helps you to prevent programming mistakes. It is this approach that we will take.

The following sections will discuss the relevant mathematics, types and operations as defined in the source code. The types and operations described below are defined in the package Math3D.

### Points in 3D

Points in 3D are really what the name says: A point in a 3-dimensional space, indicating a unique, fixed location. See the picture below as an example. Point is a position in 3D space, and pointed to by the blue line. The coordinates are presented in brackets behind it, as an example.

Figure ‑: A point in 3D space

Points in either 2D space or 3D space are printed in bold italic letters. The individual coordinates of a point are indicated with a suffix for each axis. So a point in 3D space could be printed like this:

Another notation often used is the transposed notation:

Points (both 2D and 3D) in our ray tracer are implemented in the package ***Math3D***. This package contains all basic mathematic objects, such as points, vectors, matrices and the various operations you can perform on them.

Points in 3D space are implemented as the ***private*** type ***Point3D***. I deliberately chose a private type, so the actual implementation remains hidden from the code using Point3D variables. A programmer may decide to implement a Point3D as a record, an array or whatever he sees fit. Various basic routines have to be implemented to read or set values of a Point3D variable, because other routines will need to do so without knowing the details of the underlying implementation of the data type. This is commonly known as data abstraction by using *Abstract Data Types*. The basic routines implemented for Point3D are a constructor function and routines to get the respective x, y and z coordinates. There is also a function to print a Point3D, but this is only used for debugging purposes, and is not used in the general code. Similar routines are implemented for a Point2D, Vector3D and a Normal3D.

type Point3D is private;

…

type Point3D is record

x, y, z : LARGE\_FLOAT := 0.0;

end record;

…

function ConstructPoint3D (x, y, z : in LARGE\_FLOAT) return Point3D;

function GetX (Pnt : in Point3D) return LARGE\_FLOAT;

function GetY (Pnt : in Point3D) return LARGE\_FLOAT;

function GetZ (Pnt : in Point3D) return LARGE\_FLOAT;

### Vectors in 3D

Vectors in 3D indicate a direction and distance, but not a location. Therefore, they are not the same as points. Vectors indicate a displacement from a starting point, but the starting point can be anywhere. Vectors are indicated by a bold capital letter with a small arrow above it. Its coordinates are represented in the same way as points:

Vectors in 3D space are implemented as the ***private*** type ***Vector3D***, in the package ***Math3D***.

### Normals in 3D

Normals in 3D are vectors, but they represent vectors that are perpendicular to another object, usually a 3D surface. In most formula’s we will need normals with a length of 1, i.e. unit length or *normalized*. Normals are represented as below:

Normals in 3D space are implemented as the ***private*** type, in the package ***Math3D***.

### Mathematical Operations on Points, Vectors and Normals

Besides the functions above, other calculations have to be implemented using Point3D, Vector3D and Normal3D variables as a parameter. Here we will make use of the strong typing features in Ada, as well as overloading of operators like “+”. This will make calculations more readable, as they will be coded much like the mathematic formulas. Points are different than vectors and normals, as is explained in the previous paragraphs. For example, a vector has a length, but a point has not. Therefore, calculating the length is not supported as part of the Point3D type, but it is supported as part of the Vector3D type.

The operations implemented are described in the table below.

|  |  |
| --- | --- |
|  | Displace a point by a vector by adding a vector to a point  function "+" (Left : in Point3D; Right : in Vector3D) return Point3D; |
|  | Displace a point by a vector by subtracting a vector to a point  function "-" (Left : in Point3D; Right : in Vector3D) return Point3D; |
|  | Calculate a vector by subtracting two points. The result is the vector from ***b*** to ***a***  function "-" (Left, Right : in Point3D) return Vector3D; |
|  | Displace a point by scaling it with a value *t*  function "\*" (Left : in Point3D; Right : LARGE\_FLOAT) return Point3D; |
|  | Displace a point by scaling it with a value *t*  function "\*" (Left : in LARGE\_FLOAT; Right : Point3D) return Point3D; |
|  | Calculate the distance between two points  Distance (P1, P2 : in Point3D) return LARGE\_FLOAT; |
|  | Calculate the squared distance between two points  Distance\_Squared (P1, P2 : in Point3D) return LARGE\_FLOAT; |
|  | Calculate the distance between a point and the origin  Distance\_Org (P: in Point3D) return LARGE\_FLOAT; |
|  | Calculate the squared distance between a point and the origin  Distance\_Squared\_Org (P : in Point3D)return LARGE\_FLOAT; |
|  | Add two vectors  function "+" (Left, Right : in Vector3D) return Vector3D; |
|  | Subtract two vectors. The result is the vector from to  function "-" (Left, Right : in Vector3D) return Vector3D; |
|  | Invert a vector.  function "-" (Vec : in Vector3D) return Vector3D; |
|  | Scale a vector by multiplying it with a value *t*  function "\*" (Left : in Vector3D; Right : LARGE\_FLOAT) return Vector3D; |
|  | Scale a vector by multiplying it with a value *t*  function "\*" (Left : in Vector3D; Right : LARGE\_FLOAT) return Vector3D; |
|  | Scale a vector by dividing it with a value *t*  function "/" (Left : in Vector3D; Right : LARGE\_FLOAT) return Vector3D; |
|  | Calculate the dot product between two vectors  function "\*" (Left, Right : in Vector3D) return LARGE\_FLOAT; |
|  | Calculate the cross product between two vectors  function "\*\*" (Left, Right : in Vector3D) return Vector3D; |
|  | Calculate the length of a vector  function Length (Vec : in Vector3D) return LARGE\_FLOAT; |
|  | Calculate the squared length of a vector  Function Length\_Squared (Vec : in Vector3D) return LARGE\_FLOAT; |
|  | Normalize a vector.  function Normalize (Vec : in Vector3D) return Vector3D; |
|  |  |

## Affine Transformations

When modeling a scene, we generally need to position objects at locations we want them. We may also want to rotate or scale objects. In order to do that, you can use *affine transformations*. Affine transformations are linear transformation, meaning that lines will remain lines after the transformation. Affine transformations can be represented by a *matrix*. Applying a transformation to a point, vector or normal simply comes down to multiplying them with the matrix. More about transformations and applying them to 3D objects will be discussed in section 6.3. This chapter will limit to the mathematical theory and implementation.

The transformations that are implemented are described in detail in the sections below, but before going into these, a bit of theory needs to be covered.

### Matrices

A matrix in its generic form is an M x N array of numbers. Each number is called an *element*. Matrices are used throughout this book to represent linear transformation. The numbers M x N represent the *dimensions* of a matrix, where M is the number of rows, and N is the number of columns in the matrix. Hence an M x N matrix can be presented as follows:

A vector can be regarded as an M x 1 matrix (or a 1 x N matrix) in the general theory.

#### Adding & Subtracting

Adding, and also subtracting, matrices is a matter of adding or subtracting the individual elements. It can only be done if the two matrices have the same dimension. The resulting matrix will have the same dimension too. The formula below shows an example of the subtraction of two matrices:

#### Transposing

#### Multiplying

#### Discriminant

### Translation

Translation means that you displace a point to another location. It comes down to adding a vector to a point. Since

### Scaling

This

### Rotation

This

### Skewing

This

### Combining Transformations

This

### Transforming Points, Vectors and Normals

This

## Other Mathematical Notations

This

# Light and Colors

This

# Ray Tracing Principals

Ray tracing can be explained in various ways, but the essence comes down to the description below:

* A process that constructs an image by sampling light in a scene that is represented by virtual objects in the computer’s memory somehow

The sampling process is really what it comes down to.

# A Simple Sphere Tracer

This

## A Simple Reflection Model

Describe simple diffuse shading model with a point light source

## Rendering the Unit Sphere

This

## Transforming the Unit Sphere

This section describes the inverse transformation of a ray into Unit Sphere space (local object coordinates), instead of transforming the object itself.

Describe the effect of the inverse transformation to the intersection point in world coordinates; describe the effect of the inverse transformation to the normal on the intersection point in world coordinates;

# Intersecting 3D Objects

This

## Generic Approach

Intersection of objects is done by intersecting the ray with a unified object, where possible. This greatly simplifies intersection calculations, at the expense of doing an inverse transformation.

The ray in world coordinates is expressed as follows:

, where is the ray origin and is the normalized ray direction in world coordinates

It is quite complex to intersect a ray with a unit object that has gone through various transformations. Instead, we will apply the inverse of the transformations that have been applied to an object to the ray. This will transform the ray into *unit object space*, and will then reduce the intersection to a simple ray-unit object intersection calculation.

, where is the inverse transformation that has been applied to the object

This can be rewritten as

, where is the ray origin and is the (normalized) ray direction in world coordinates

or

, where is the ray origin and is the ray direction in unit object coordinates

Because is a linear transformation, applying the same value for in both world as unit object space will result in two points that correspond with each other by applying the object’s transformation. Note that this is only valid if you do not normalize after the transformation into unit object space [TO BE VERIFIED]

## Sphere

A sphere is defined as an object where all points have a distance equal or less than the *radius* from a fixed point, called the *center point*. Mathematically, a sphere can be expressed as follows:

, where is the center point, and is the radius

However, for a unit sphere, it is convenient to choose the center point as (0, 0, 0) and a radius of 1. This would then result in:

Calculating the intersection of the ray with this unit sphere comes down to inserting as coordinates into the equation above:

The last equation can be resolved by a simple root finder for , as this is a quadratic equation. The generic solutions would be:

Note that the ray only intersects the sphere if the *discriminant* is not negative. This means that one or two intersections will exist if and only if:

In case *D = 0*, a single intersection will exist. In case *D > 0*, then two intersections will exist. The test for *D* < *0* can be used to quickly determine if the ray will hit at all, and an early bail-out of expensive intersection calculations can be done.

## Box

This

## Cylinder

This

## Cone

This

## Torus

This

## Triangle Mesh

This

## CSG Object

This

# Ray Tracing as a Sampling Process

This

An dramatic example to illustrate the effects of sampling a continuous signal at discrete intervals is sampling the following function in different intervals:

(1)

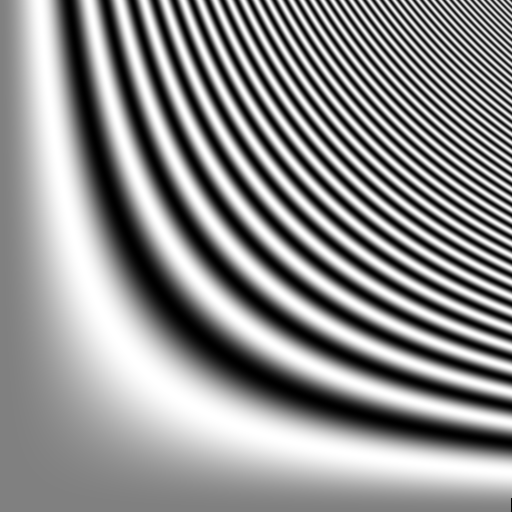
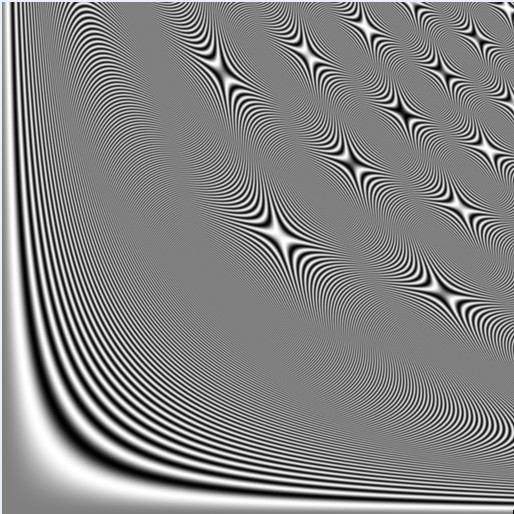
As the return value of this function is always in the range [0, 1], you can easily map this to an integer gray scale value in [0,255].

Figure ‑: Function (1) sampled with x and y ranging from 0 to 4 in the left picture. The right picture was generated by ranging x and y from 0 to 10.

# The Rendering Equation

This

The equation above is the Rendering Equation, where:

= the total amount of light sent out into direction i

# Diffuse Reflection

This

# Specular Reflection

# Texture Mapping

This

# Acceleration Methods

This

# Dripping Objects

This

# Radiosity

This

1. References

[001]

1. REFERENCES

This