

Module Interface Specification for Lighting Models

Sasha Soraine

December 5, 2019

1 Revision History

Date		Version	Notes
November 2019	29	1.0	Submitted Document to GitHub after extension

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at [../Commonality-Analysis/CA.pdf](#)

Contents

1	Revision History	i
2	Symbols, Abbreviations and Acronyms	ii
3	Introduction	1
4	Notation	1
5	Module Decomposition	2
6	MIS of Input Parameters Module	3
6.1	Module	3
6.2	Uses	3
6.3	Syntax	3
6.3.1	Exported Constants	3
6.3.2	Exported Access Programs	3
6.4	Semantics	4
6.4.1	State Variables	4
6.4.2	Environment Variables	4
6.4.3	Assumptions	4
6.4.4	Access Routine Semantics	4
6.4.5	Local Functions	4
7	MIS of Output Parameters Module	5
7.1	Module	5
7.2	Uses	5
7.3	Syntax	5
7.3.1	Exported Types	5
7.3.2	Exported Access Programs	5
7.4	Semantics	5
7.4.1	State Variables	5
7.4.2	Environment Variables	5
7.4.3	Assumptions	5
7.4.4	Access Routine Semantics	5
7.4.5	Local Functions	6
8	MIS of Point3D	7
8.1	Template Module	7
8.2	Uses	7
8.3	Syntax	7
8.3.1	Exported Types	7
8.3.2	Exported Access Programs	7

8.4	Semantics	7
8.4.1	State Variables	7
8.4.2	Environment Variables	7
8.4.3	Assumptions	8
8.4.4	Access Routine Semantics	8
8.4.5	Local Functions	8
9	MIS of Colour	9
9.1	Template Module	9
9.2	Uses	9
9.3	Syntax	9
9.3.1	Exported Types	9
9.3.2	Exported Access Programs	9
9.4	Semantics	9
9.4.1	State Variables	9
9.4.2	Environment Variables	9
9.4.3	Assumptions	10
9.4.4	Access Routine Semantics	10
9.4.5	Local Functions	11
10	MIS of Vector	12
10.1	Template Module	12
10.2	Uses	12
10.3	Syntax	12
10.3.1	Exported Types	12
10.3.2	Exported Access Programs	12
10.4	Semantics	12
10.4.1	State Variables	12
10.4.2	Environment Variables	12
10.4.3	Assumptions	13
10.4.4	Access Routine Semantics	13
10.4.5	Local Functions	13
11	MIS of Light Type	14
11.1	Template Module	14
11.2	Uses	14
11.3	Syntax	14
11.3.1	Exported Types	14
11.3.2	Exported Access Programs	14
11.4	Semantics	14
11.4.1	State Variables	14
11.4.2	Environment Variables	14
11.4.3	Assumptions	15

11.4.4	Access Routine Semantics	15
11.4.5	Local Functions	15
12	MIS of Polygon	16
12.1	Template Module	16
12.2	Uses	16
12.3	Syntax	16
12.3.1	Exported Types	16
12.3.2	Exported Access Programs	16
12.4	Semantics	16
12.4.1	State Variables	16
12.4.2	Environment Variables	17
12.4.3	Assumptions	17
12.4.4	Access Routine Semantics	17
12.4.5	Local Functions	18
13	MIS of Mesh	19
13.1	Template Module	19
13.2	Uses	19
13.3	Syntax	19
13.3.1	Exported Types	19
13.3.2	Exported Access Programs	19
13.4	Semantics	19
13.4.1	State Variables	19
13.4.2	Environment Variables	19
13.4.3	Assumptions	20
13.4.4	Access Routine Semantics	20
13.4.5	Local Functions	21
14	MIS of LightSources	22
14.1	Template Module	22
14.2	Uses	22
14.3	Syntax	22
14.3.1	Exported Types	22
14.3.2	Exported Access Programs	22
14.4	Semantics	22
14.4.1	State Variables	22
14.4.2	Environment Variables	23
14.4.3	Assumptions	23
14.4.4	Access Routine Semantics	23
14.4.5	Local Functions	23

15 MIS of Observer	24
15.1 Template Module	24
15.2 Uses	24
15.3 Syntax	24
15.3.1 Exported Types	24
15.3.2 Exported Access Programs	24
15.4 Semantics	24
15.4.1 State Variables	24
15.4.2 Environment Variables	24
15.4.3 Assumptions	24
15.4.4 Access Routine Semantics	25
15.4.5 Local Functions	25
16 MIS of NormalMap	26
16.1 Template Module	26
16.2 Uses	26
16.3 Syntax	26
16.3.1 Exported Types	26
16.3.2 Exported Access Programs	26
16.4 Semantics	26
16.4.1 State Variables	26
16.4.2 Environment Variables	26
16.4.3 Assumptions	26
16.4.4 Access Routine Semantics	26
16.4.5 Local Functions	27
17 MIS of Object	28
17.1 Template Module	28
17.2 Uses	28
17.3 Syntax	28
17.3.1 Exported Types	28
17.3.2 Exported Access Programs	29
17.4 Semantics	29
17.4.1 State Variables	29
17.4.2 Environment Variables	30
17.4.3 Assumptions	30
17.4.4 Access Routine Semantics	30
17.4.5 Local Functions	33
18 MIS of Scene Module	34
18.1 Module	34
18.2 Uses	34
18.3 Syntax	34

18.3.1	Exported Constants	34
18.3.2	Exported Access Programs	35
18.4	Semantics	35
18.4.1	State Variables	35
18.4.2	Environment Variables	35
18.4.3	Assumptions	35
18.4.4	Access Routine Semantics	36
18.4.5	Local Functions	36
19	MIS of VecMath	37
19.1	Module	37
19.2	Uses	37
19.3	Syntax	37
19.3.1	Exported Constants	37
19.3.2	Exported Access Programs	37
19.4	Semantics	37
19.4.1	State Variables	37
19.4.2	Environment Variables	37
19.4.3	Assumptions	37
19.4.4	Access Routine Semantics	37
19.4.5	Local Functions	39
20	MIS of Shader	40
20.1	Module	40
20.2	Uses	40
20.3	Syntax	40
20.3.1	Exported Constants	40
20.3.2	Exported Access Programs	40
20.4	Semantics	40
20.4.1	State Variables	40
20.4.2	Environment Variables	40
20.4.3	Assumptions	40
20.4.4	Access Routine Semantics	40
21	MIS of LightingModel	42
21.1	Module	42
21.2	Uses	42
21.3	Syntax	42
21.3.1	Exported Constants	42
21.3.2	Exported Access Programs	42
21.4	Semantics	42
21.4.1	State Variables	42
21.4.2	Environment Variables	42

21.4.3 Assumptions	43
21.4.4 Access Routine Semantics	43
21.4.5 Local Functions	43
22 Appendix	45

3 Introduction

The following document details the Module Interface Specifications for Lighting Models.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at <https://github.com/sorainism/library-of-lighting-models>.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol $:=$ is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | \dots | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Lighting Models.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	\mathbb{Z}	a number without a fractional component in $(-\infty, \infty)$
natural number	\mathbb{N}	a number without a fractional component in $[1, \infty)$
real	\mathbb{R}	any number in $(-\infty, \infty)$
3D Cartesian Coordinate	Point3D	A 3-dimensional cartesian coordinate, represented as an (x,y,z)-tuple where all three are \mathbb{R} values
RGB Colour	Colour	A 3-tuple represented as (r,g,b)- where all three are \mathbb{R} values
Shape of Object	Shape	The abstract shape that an object mesh is classified as. It can be one of the following : sphere, cube, torus, teapot.
Polygon Mesh	Mesh	Mesh constructed of vertices, edges, and triangle surfaces to create one of the allowed shapes.
Normal Map of Object	nMap	A structure maintaining a list of the normal vectors for the measured points on the mesh.

[There is already an issue about this, but I'm mention again that types like RGB Colour cannot really be primitive data types. They'll need to be defined in the MIS. —SS]

The specification of Lighting Models uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Lighting Models uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	Level 2
Hardware-Hiding Module	
Behaviour-Hiding Module	Input Parameters Module Output Format Module Polygon Module Colour Module 3D Cartesian Coordinate (Point3D) Module Polygon Mesh Module Normal Maps Module Scene Module Object Module Light Source Module Observer Module Vector Math Module Shader Module Lighting Model Module
Software Decision Module	JSON Module Rendering Module

Table 1: Module Hierarchy

The following sections of this document will outline the module interface specifications for the modules listed in the module hierarchy. Three modules are omitted from this discussion: the hardware-hiding module, the JSON module, and the rendering module. The hardware hiding module is provided via the syntax of the programming language to interface with the computer hardware, as such I will not be documenting it here. The JSON and rendering module will be handled natively by the Unity environment, the process of documenting how it would work wouldn't be appropriate for the scope of this work. Suffice it to say that the documentation for the rendering module can be found in the documentation for the Unity Engine.

6 MIS of Input Parameters Module

2

[Please add a makefile for building the documents so that the cross-references between documents can be kept up to date. The module reference number without an introduction is confusing. Even putting an M in front of it would help identify it as a module. —SS] The Input Parameters Module converts the JSON data from the input file into the objects usable by the system. During this process, the input is read as a string into the system from the file and passed through the JSON module to parse them.

6.1 Module

Input Parameters

6.2 Uses

6.3 Syntax

6.3.1 Exported Constants

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
loadFile	String	-	INVALID_FILE_NAME, IN- VALID_FILE_TYPE, FILE_EMPTY
convertJSONtoScene	JSON File	s: Scene o : Object l : Light- Source v : Ob- server	

[Where are the types `Object`, `LightSource` etc. defined? This is what the `Uses` clause is for. You can tell the reader where they can find the details they need to understand the specification of this module. —SS]

6.4 Semantics

6.4.1 State Variables

`loaded` : `String`

6.4.2 Environment Variables

`input`: `File`

6.4.3 Assumptions

N/A

6.4.4 Access Routine Semantics

`loadFile(f:String)`:

- transition: `loaded := ReadFile(f)` [You should make the connection that the filename `f` refers to the environment variable `input`. —SS]
- exception: `exc := {f does not exist \implies INVALID_FILE_NAME
| f \rightarrow input \wedge input is not a JSON file \implies INVALID_FILE_TYPE
| ReadFile(f) outputs an empty string \implies FILE_EMPTY }`

`convertJSONtoScene()`: [Without knowing the types, I cannot understand what is happening here. I can see why you would not want to get into the details of the JSON format, but you should point to a reference (probably in Unity) that tells the readers the connection between the file and the outputs. —SS]

- `output := s : initScene, o : Object, l : LightSource , v: Observer`
- exception: N/A

6.4.5 Local Functions

N/A

7 MIS of Output Parameters Module

3

The Output Parameters Module converts the data from the scene into JSON formatted data.

7.1 Module

Output Parameters

7.2 Uses

7.3 Syntax

7.3.1 Exported Types

OutputObject = ? [You use this notation for ADTs, but I do not get the impression that this is what you mean here. For an ADT the word Template should be in the heading. Also, ADTs have constructors. You only have one access program and it a constructor. —SS]

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
convert	Scene, String	-	INVALID_FILE_NAME, NO_DATA_TO_WRITE

7.4 Semantics

7.4.1 State Variables

writing : (Scene, Object, LightSource, Observer)[The notation used here is not the Hoffman and Strooper notation. Is this a tuple? What are the field variables and what are the types? If these are types, where are they defined? —SS]

7.4.2 Environment Variables

output: File

7.4.3 Assumptions

N/A

7.4.4 Access Routine Semantics

convert(s: Scene, o: String):

- transition: writing := (s, s.os, s.ls, s.obs)

- output: $\text{output} := \text{OpenFile}(o) \text{ and } \text{ToJSON}(\text{writing})$
- exception: $\text{exc} := \{o \text{ already exists} \implies \text{INVALID_FILE_NAME}$
 $| \text{ writing is empty} \implies \text{NO_DATA_TO_WRITE}$
 $\}$

7.4.5 Local Functions

N/A

8 MIS of Point3D

6

The Point3D module captures the structure of a 3D Caretsian Coordinate and functions that are useful for this structure.

8.1 Template Module

Point3D

8.2 Uses

-

8.3 Syntax

8.3.1 Exported Types

Point3D = ?

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
Point	$\mathbb{R}, \mathbb{R}, \mathbb{R}$	Point3D	—
.x	—	\mathbb{R}	—
.y	—	\mathbb{R}	—
.z	—	\mathbb{R}	—
distance_abs	Point3D	\mathbb{R}	—

8.4 Semantics

8.4.1 State Variables

$x : \mathbb{R}$

$y : \mathbb{R}$

$z : \mathbb{R}$

8.4.2 Environment Variables

N/A

8.4.3 Assumptions

Point3D positions (x,y,z) are only set once (at initialization). This means there will be no individual setter methods.

We assume that all the routines can only be called after Point() has been called once. This means there needs to be at least one Point3D before you can call other routines.

8.4.4 Access Routine Semantics

Point($Ix : \mathbb{R}, Iy : \mathbb{R}, Iz : \mathbb{R}$):

- transition: $x, y, z := Ix, Iy, Iz$
- output:= self
- exception: N/A

.x():

- output:= self.x
- exception: N/A

.y():

- output:= self.y
- exception: N/A

.z():

- output:= self.z
- exception: N/A

distance_abs(p:Point3D):

- output:= $\sqrt{(p.x - self.x)^2 + (p.y - self.y)^2 + (p.z - self.z)^2}$
- exception: N/A

8.4.5 Local Functions

N/A

[\[The Point3D ADT looks good. —SS\]](#)

9 MIS of Colour

5

The Colour module captures the structure of colours used in this program. [\[This module looks fine. —SS\]](#)

9.1 Template Module

Colour

9.2 Uses

-

9.3 Syntax

9.3.1 Exported Types

Colour = ?

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
Colour	$\mathbb{Z}^+, \mathbb{Z}^+, \mathbb{Z}^+$		INVALID_R, IN- VALID_G, IN- VALID_B
.r	—	\mathbb{Z}^+	—
.g	—	\mathbb{Z}^+	—
.b	—	\mathbb{Z}^+	—
.set_r	\mathbb{Z}^+		—
.set_g	\mathbb{Z}^+		—
.set_b	\mathbb{Z}^+		—

9.4 Semantics

9.4.1 State Variables

r : \mathbb{Z}^+
g : \mathbb{Z}^+
b : \mathbb{Z}^+

9.4.2 Environment Variables

N/A

9.4.3 Assumptions

- Colours can be changed at any point in time - therefore setters will be needed.
- Colours are represented by RGB values that (individually) range from 0 to 255.

9.4.4 Access Routine Semantics

Colour($Ir : \mathbb{Z}^+, Ig : \mathbb{Z}^+, Ib : \mathbb{Z}^+$):

- transition: $r, g, b := Ir, Ig, Ib$
- exception: $\text{exc} := (r < 0 \parallel r > 255) \implies \text{INVALID_R}$
 $\quad \quad \quad (g < 0 \parallel g > 255) \implies \text{INVALID_G}$
 $\quad \quad \quad (b < 0 \parallel b > 255) \implies \text{INVALID_B}$

.r():

- output: $\text{self}.r$
- exception: N/A

.g():

- output: $\text{self}.g$
- exception: N/A

.b():

- output: $\text{self}.b$
- exception: N/A

.set_r($Ir : \mathbb{Z}^+$):

- transition: $r := Ir$
- exception: $\text{exc} := (r < 0 \parallel r > 255) \implies \text{INVALID_R}$

.set_g($Ig : \mathbb{Z}^+$):

- transition: $g := Ig$
- exception: $\text{exc} := (g < 0 \parallel g > 255) \implies \text{INVALID_G}$

.set_b($Ib : \mathbb{Z}^+$):

- transition: $b := Ib$
- exception: $\text{exc} := (b < 0 \parallel b > 255) \implies \text{INVALID_B}$

9.4.5 Local Functions

N/A

10 MIS of Vector

14

The Vector module captures the structure of Vector objects.

10.1 Template Module

Vector

10.2 Uses

10.3 Syntax

10.3.1 Exported Types

Vector = ?

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
Vector_P	Point3D, Point3D	–	SAME_POINTS
Vector	$\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}$	–	INVALID_UX, IN- VALID_UY, IN- VALID_UZ, IN- VALID_M
.m		\mathbb{R}	–
direction		$\mathbb{R}, \mathbb{R}, \mathbb{R}$	–

[You might think about direction outputting a vector. —SS]

10.4 Semantics

10.4.1 State Variables

start := Point3D
ux := \mathbb{R}
uy := \mathbb{R}
uz := \mathbb{R}
m := \mathbb{R}

10.4.2 Environment Variables

N/A

10.4.3 Assumptions

- Vectors can be created infinitely; we will only set them once during initialization.

10.4.4 Access Routine Semantics

Vector(p:Point3D, q:Point3D):

- transition: start:= p
ux:= (q.x - p.x)/m
uy:= (q.y - p.y)/m
uz:= (q.z - p.z)/m
m := start.distance_abs(q)

- exception: exc:= { p == q \implies SAME_POINTS }

[You haven't actually defined what it means for two vectors to be equal. You might want to add an access program for this. —SS]

Vector(Ix : \mathbb{Z} , Iy : \mathbb{Z} , Iz : \mathbb{Z} , Im : \mathbb{R}):

- transition: ux, uy, uz, m := Ix, Iy, Iz, Im
- exception: exc := (ux < -1 || ux > 1) \implies INVALID_UX
|(ux < -1 || ux > 1) \implies INVALID_UY
|(ux < -1 || ux > 1) \implies INVALID_UZ
|(m < 0) \implies INVALID_M [The vector 0.9, 0.9, 0.9 satisfies your rules, but the magnitude is greater than 1. You could just look at the distance from the origin to the end of the proposed unit vector. If it is close enough to 1 you could say that the unit vector is valid. —SS]

.m():

- output: *self.m*
- exception: N/A

direction():

- output: *self.ux, self.uy, self.uz*
- exception: N/A

.start():

- output: *self.start*
- exception: N/A

10.4.5 Local Functions

N/A

11 MIS of Light Type

7

The Light Type module is an abstract data type which captures information related to the different types of light sources.

11.1 Template Module

LightType

11.2 Uses

N/A

11.3 Syntax

11.3.1 Exported Types

LightType = ?

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
LightType	{ambient,point,spotlight,directional}	LightType	–
.name		LightType	–
.i	LightType	$\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$	–

[I suggest that somewhere you export the type {ambient,point,spotlight,directional} and give it a convenient name. —SS]

11.4 Semantics

11.4.1 State Variables

name := { ambient, point, spotlight, directional }

i := Function that describes how the light intensity changes as a function of distance. Every type of light has an associated function - so this should really be a set of functions.

11.4.2 Environment Variables

N/A

11.4.3 Assumptions

11.4.4 Access Routine Semantics

LightType(inName):

- transition: $\text{self.name} := \text{inName}$
 $\text{self.i} := (\text{name} == \text{ambient} \implies \lambda d, i_0 \rightarrow i_0$
| $\text{name} == \text{directional} \implies \lambda d, i_0 \rightarrow \frac{1}{d^2} i_0$
| $\text{name} == \text{point} \implies \lambda d, i_0 \rightarrow i_0$
| $\text{name} == \text{spotlight} \implies)$ [H&S do not use == for equality; they use = —SS]
- output: self
- exception: $\text{exc} := \{\text{inName} \notin \text{ambient, spotlight, point, directional} \implies \text{INVALID_LIGHT_TYPE}\}$

.name():

- output: *self.name*
- exception: N/A

.i():

- output: *self.i*
- exception: N/A

11.4.5 Local Functions

N/A

12 MIS of Polygon

4

The Polygon module is an abstract data type captures the structure of polygons used in polygon meshes.

12.1 Template Module

Polygon

12.2 Uses

12.3 Syntax

12.3.1 Exported Types

Polygon = ?

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
Polygon	{triangle, quad}, (Point3D, Vector) ⁿ [What does <i>n</i> mean here? —SS]	—	—
.shape	—	{triangle, quad}	—
.bounds	—	Set of (Point3D, Vector)	—
.s_norm	—	Vector	—
getEdges	Point3D	Set of Vectors	—
getPoints		Set of Point3D	—

12.4 Semantics

12.4.1 State Variables

shape := {triangle, quad} [As mentioned elsewhere, giving sets like this a name will make your document easier to read and write. —SS]

bounds := Set of (Point3D, Vector) tuples [Please use the H&S notation. —SS]

s_norm := Vector [You could calculate this as needed, rather than have it as a state variable. —SS]

[A polygon can be represented by a sequence of points. I'm not sure why you need vectors in the state variables. You could calculate a vector between two points if you need it, but it seems redundant to keep state information on both. It is an extra headache to make sure that your points and vectors are consistent. Alternatively, you could store a set of vectors.

The way you have defined vectors to contain a starting point, the vectors alone contain all of the information you need. —SS]

12.4.2 Environment Variables

N/A

12.4.3 Assumptions

12.4.4 Access Routine Semantics

Polygon($t : \{triangle, quad\}, (p : Point3D, v : Vector)^n$):

- transition:= $shape := t$;
 $bounds := \cup(p, v)$ [what sets is this a union of? You cannot have a union of sets of different types. I think you might need a different notation for what you are trying to do? —SS]
 $s_norm :=$ Calculate norm as cross-product of two vectors from 1 vertex. [Your vector ADT does not have cross product —SS]
- exception: $exc := \{(t \notin \{triangle, quad\} \implies INVALID_SHAPE) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == triangle, \text{sizeOfBounds} < 6 \implies TOO_FEW_POINTS) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == triangle, \text{sizeOfBounds} > 6 \implies TOO_MANY_POINTS) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == quad, \text{sizeOfBounds} > 8 \implies TOO_MANY_POINTS) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == quad, \text{sizeOfBounds} < 8 \implies TOO_FEW_POINTS) \}$

.shape():

- output:= self.shape
- exception: N/A

.bounds():

- output:= self.bounds
- exception: N/A

.s_norm():

- output:= self.s_norm
- exception: N/A

getEdges(p:Point3D):

This method retrieves all the edges that are connected to the vertex represented by Point3D p. Individual polygons should have a maximum of two edges per vertex based on the polygon assumptions.

- output:= Set of Vectors := $\forall b : (Point3D, Vector) | (b \in self.bounds \wedge b[0] == p) \implies \cup b[1]$
- exception: N/A

getPoints():

This method retrieves the set of points in the polygon.

- output: Set of Point3D := $b : (Point3D, Vector) | \forall b \in self.bounds \cup b.[0]$
- exception: N/A

12.4.5 Local Functions

sizeOfBounds \equiv Number of elements in the set of (Point3D, Vector) tuples.

13 MIS of Mesh

8

The Mesh module is an abstract data type that captures the structure of polygon meshes as used by this program. It also provides methods to find out basic data about the polygon mesh.

13.1 Template Module

Mesh

13.2 Uses

13.3 Syntax

13.3.1 Exported Types

Mesh = ?

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
Mesh	Set of Polygons	—	—
.Surfaces	-	Set of Polygons	—
.Edges	-	Set of Vectors	—
.Vertices	-	Set of Point3D	—
isInMesh	Polygon	\mathbb{B}	—
numPoly	Point3D	\mathbb{Z}^+	—
intersects	Vector	Polygon	—
pointsOnMesh	Point3D	\mathbb{B}	—

13.4 Semantics

13.4.1 State Variables

Vertices : Set of Point3D

Edges : Set of Vectors

Surfaces : Set of Polygons [\[Why can't your mesh just be a set of polygons? The polygons already contain points and vectors. —SS\]](#)

13.4.2 Environment Variables

N/A

13.4.3 Assumptions

13.4.4 Access Routine Semantics

Mesh($P : Set of Polygons$):

- transition: Surfaces := P Vertices := ($p : Polygon | \forall p \in P \rightarrow \cup p.getPoints$)
(Vertices pulls its values from the bounds of the polygons in P)
Edges := ($p : Polygon, v : Point3D | \forall p \in P \forall v \in p.getPoints \cup (p.getEdges(v))$)
(Edges pulls its values from the bounds of the polygons in P)
- exception: exc := { $P == \emptyset \implies INVALID_MESH$
 $|(p, q : Polygon | \forall p, q \in P, p \neq q \wedge p.shape \neq q.shape \implies POLYGON_SHAPES_MISMATCH)$
 $|(p, q : Polygon, p_1, q_1 : Point3D | \forall p \in P, \exists q \in P \text{ such that } \exists p_1 \in p.getPoints() \wedge \exists q_1 \in q.getPoints() \text{ such that } p_1 \neq q_1 \implies INVALID_POLYS)$ }

.Surfaces():

- output := self.Surfaces
- exception: N/A

.Vertices():

- output := self.Vertices
- exception: N/A

.Edges():

- output := self.Edges
- exception: N/A

isInMesh($p : Polygon$):

- output := ($q : Polygon | \exists q \in self.Surfaces \text{ where } q == p$)
- exception: N/A

numPoly($p : Point3D$):

- output:= counter := $p \in self.Vertices \implies (s : Polygon | \forall s \in self.Surfaces) \text{ if } p \in s.bounds \text{ then } counter++$
- exception: exc := { $p \notin self.Vertices \implies ERR_POINT_NOT_IN_MESH$ }

intersects($r : Vector$):

- output := calculate whether the given vector intersects with any polygon on the mesh, and return the first polygon it intersects with.
- exception: exc :=

pointsOnMesh(p: Point3D):

- output := return true if p is a point on a polygon in the mesh.
- exception: exc :=

13.4.5 Local Functions

N/A

14 MIS of LightSources

12

The Light Source module is an Abstract Data Type that defines the structure and behaviours of light sources in the scene.

14.1 Template Module

LightSource

14.2 Uses

[You want to put the modules here that define the types you will be using, like Point3D, Colour, etc. —SS]

14.3 Syntax

14.3.1 Exported Types

LightSource = ?

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
LightSource	Point3D, Colour, LightType, \mathbb{R} , Set of Vectors	LightSource	
.origin		Point3D	
.colour		Colour	
.type		LightType	
.intensity		\mathbb{R}	

[The Set of Vectors type in the interface may give you headaches. Internally using sets is fine, but when it is exposed at the interface, you will have to still find a set type for the implementation. A sequence of vectors might be easier. —SS]

14.4 Semantics

14.4.1 State Variables

o: Point3D
c: Colour
t: lightType
 i_0 : \mathbb{R}
ds: Set of Vector

14.4.2 Environment Variables

N/A

14.4.3 Assumptions

14.4.4 Access Routine Semantics

LightSource(inP: Point3d, inC: Colour, lt: LightType, ins: \mathbb{R} inDs: Set of Vectors):

- transition: o, c, t, i, ds := inP, inC, lt, ins, inDs
- exception: N/A

.origin():

- output:= self.o
- exception: N/A

.colour():

- output:= self.c
- exception: N/A

.type():

- output:= self.t
- exception: N/A

.intensity():

- output: self.i
- exception: N/A

14.4.5 Local Functions

N/A

15 MIS of Observer

13

The Observer Module is an Abstract Data Type which captures information related to the camera in a scene. While there's no behaviour and this type of information could be represented as an Abstract Object since there's only one at any time in the scene, I'm attempting to future proof the design by keeping it an Abstract Data Type.

15.1 Template Module

Observer

15.2 Uses

15.3 Syntax

15.3.1 Exported Types

Observer = ?

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
Observer	Point3D, Vector	Observer	—

[If you only have a constructor, how do any other modules use the state information? If this module really is just data, you might save yourself some work by defining it as an exported type. —SS]

15.4 Semantics

15.4.1 State Variables

p : Point3D

d : Vector

15.4.2 Environment Variables

N/A

15.4.3 Assumptions

There is only one observer in the scene at any time. This might change in future versions of this software, but as it exists we're only looking at the objects from one view.

15.4.4 Access Routine Semantics

Observer(inP: Point3D, inD : Vector):

- transition: $p := \text{inP}$
 $d := \text{inD}$
- output := self
- exception: $\text{exc} :=$ [If there are no exceptions, you should say “none” explicitly. —SS]

15.4.5 Local Functions

N/A

16 MIS of NormalMap

9

The NormalMap module is an Abstract Data Type which captures information about the normal maps of an object mesh. This information is necessary for calculating reflections and is easier to calculate once and store instead of calculating on the fly.

16.1 Template Module

NormalMap

16.2 Uses

16.3 Syntax

16.3.1 Exported Types

NormalMap = ?

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
NormalMap	Set of (Point3D, Vector)	NormalMap	–

[Again, you only have a constructor in the interface. You likely want some getters at least. Having Set as the type in your interface may cause you some headaches. If you really want Set, you should probably define an Generic ADT for Set. —SS]

16.4 Semantics

16.4.1 State Variables

NormalMap = Set of (Point3D, Vector)

16.4.2 Environment Variables

N/A

16.4.3 Assumptions

16.4.4 Access Routine Semantics

NormalMap(ns : (Point3D,Vector)):

- transition: $p := \text{inP}$
 $d := \text{inD}$
- output := self
- exception: exc :=

getNormal(p:Point3D):

- output :=
- exception: exc :=

16.4.5 Local Functions

N/A

17 MIS of Object

11

The Object module is an abstract data type that captures the structure of objects in the scenes defined by this program.

17.1 Template Module

Object

17.2 Uses

17.3 Syntax

17.3.1 Exported Types

Object = ?

17.3.2 Exported Access Programs

Name	In	Out	Exceptions
Object	Mesh, Point3D, \mathbb{R} , Colour, Colour, \mathbb{Z} , \mathbb{Z} , \mathbb{Z} , \mathbb{N} , {FLAT, GOURAUD, PHONG}	Object	
.Mesh	-	Mesh	-
.Position	-	Point3D	-
.Size	-	\mathbb{Z}	
.BaseColour	-	Colour	
.SpecColour	-	Colour	
.kd	-	\mathbb{R}	
.ka	-	\mathbb{R}	
.ks	-	\mathbb{R}	
.alpha	-	\mathbb{N}	
.nmap	-	NormalMap	
SetObj_Position	Point3D	-	
SetObj_Size	\mathbb{R}	-	
SetObj_BaseColour	Colour	-	
SetObj_SpecColour	Colour	-	
SetObj_kd	\mathbb{R}	-	IV_OUT_OF_BOUNDS
SetObj_ka	\mathbb{R}	-	IV_OUT_OF_BOUNDS
SetObj_ks	\mathbb{R}	-	IV_OUT_OF_BOUNDS
SetObj_alpha	\mathbb{Z}^+	-	IV_OUT_OF_BOUNDS
SetObj_NormalMap	<i>nMap</i>	-	-

17.4 Semantics

17.4.1 State Variables

baseColour : Colour
 specColour : Colour
 centrePoint : Point3D
 mesh : Mesh
 ka : \mathbb{R}

$ks : \mathbb{R}$
 $kd : \mathbb{R}$
 $alpha : \mathbb{Z}^+$
 $nMap : \text{NormalMap}$
 $size : \mathbb{R}$
 $shade : \{\text{FLAT}, \text{GOURAUD}, \text{PHONG}\}$

17.4.2 Environment Variables

N/A

17.4.3 Assumptions

17.4.4 Access Routine Semantics

Object(inM: Mesh, inP : Point3D, inSize : \mathbb{R} , inBase : Colour, inSpec : Colour, inD : \mathbb{Z} , inA : \mathbb{R} , inS : \mathbb{R} , inAlpha : \mathbb{N} , inShade : {FLAT, GOURAUD, PHONG}):

- transition: mesh, baseColour, specColour, centrePoint, ka, kd, ks, alpha, size := inM, inBase, inSpec, inP, inA, inD, inS, inAlpha, inSize
nMap := Shader.findNormals(shade, self)
- exception: N/A

.Mesh():

- output:= self.m
- exception: N/A

.Position():

- output:= self.centrePoint
- exception: N/A

.Size():

- output:= self.size
- exception: N/A

.BaseColour():

- output:= self.baseColour
- exception: N/A

.SpecColour():

- output:= self.specColour
- exception: N/A

.kd():

- output:= self.kd
- exception: N/A

.ka():

- output:= self.ka
- exception: N/A

.ks():

- output:= self.ks
- exception: N/A

.alpha():

- output: self.alpha
- exception: N/A

.NormalMap():

- output:= self.nMap
- exception: N/A

SetObj_Position(p: Point3D):

- transition: centrePoint := p
- exception: N/A

SetObj_Size(s : \mathbb{R}):

- transition: size := s
- exception: N/A

SetObj_BaseColour(c : Colour):

- transition: baseColour := c

- exception: $\text{exc} :=$

$$\begin{array}{l}
c.r > 255 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.g \wr 255 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.b \wr 255 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.r \wr 1 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.g \wr 1 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.b \wr 1 \implies \text{IV_OUT_OF_BOUNDS}
\end{array}$$

[I fixed one of your \wr to be $>$, you will have to fix the other ones too. —SS]

$\text{SetObj_SpecColour}(c : \text{Colour})$:

- transition: $\text{specColour} := c$
- exception: $\text{exc} :=$

$$\begin{array}{l}
c.r \wr 255 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.g \wr 255 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.b \wr 255 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.r \wr 1 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.g \wr 1 \implies \text{IV_OUT_OF_BOUNDS} \\
| \\
c.b \wr 1 \implies \text{IV_OUT_OF_BOUNDS}
\end{array}$$

$\text{SetObj_kd}(d : \mathbb{R})$:

- transition: $\text{kd} := d$
- exception: $\text{exc} :=$

$$\begin{array}{l}
d \wr 1 \implies \text{COEFFICIENT_TOO_HIGH} \\
d \wr 0.5 \implies \text{COEFFICIENT_TOO_LOW}
\end{array}$$

$\text{SetObj_ka}(a : \mathbb{R})$:

- transition: $\text{ka} := a$

- exception: $\text{exc} :=$
 $a \wr 1 \implies \text{COEFFICIENT_TOO_HIGH}$
 $a \wr 0 \implies \text{COEFFICIENT_TOO_LOW}$

$\text{SetObj_ks}(s: \mathbb{R})$:

- transition: $\text{ks} := s$
- exception: $\text{exc} :=$
 $s \wr 1 \implies \text{COEFFICIENT_TOO_HIGH}$
 $s \wr 0 \implies \text{COEFFICIENT_TOO_LOW}$

$\text{SetObj_alpha}(al: \mathbb{N})$:

- transition: $\text{alpha} := al$
- exception: $\text{exc} :=$
 $a \wr 0 \implies \text{COEFFICIENT_TOO_LOW}$

$\text{SetObj_NormalMap}()$:

- output: A normal map of the object. This is a list of normals based on shader calculations, and a string literal that describes the type of normals (vertex, surface, pixel).
- exception: N/A

17.4.5 Local Functions

N/A

18 MIS of Scene Module

10

The Scene Module is an abstract object module that contains the structure for the overall scene. It maintains information about the entities in the scene (object, light source, observer) regarding their distances between each other. It constrains the positions, sizes, and directions of entities based on the specified size of the scene.

18.1 Module

Scene

18.2 Uses

Input, Output

18.3 Syntax

18.3.1 Exported Constants

SCENE_MAX_H : \mathbb{R}^+
SCENE_MIN_H : \mathbb{R}^+
SCENE_MAX_W : \mathbb{R}^+
SCENE_MIN_W : \mathbb{R}^+
SCENE_MAX_D : \mathbb{R}^+
SCENE_MIN_D : \mathbb{R}^+

18.3.2 Exported Access Programs

Name	In	Out	Exceptions
initScene	\mathbb{R}^+	Scene	HEIGHT_TOO_SMALL,
			HEIGHT_TOO_LARGE
	\mathbb{R}^+		WIDTH_TOO_SMALL,
			WIDTH_TOO_LARGE
	\mathbb{R}^+		DEPTH_TOO_SMALL,
			DEPTH_TOO_LARGE
	Object		INVALID_OBJECT_POSITION
	LightSource		INVALID_LIGHT_POSITION
	Observer		INVALID_OBSV_POSITION
	{DIFFUSE, HALF-LAMBERT, PHONG, BLINN-PHONG}		
compile	–	OutputObject	
changeObj	Object	-	
changeLightSource	LightSource	-	
changeLightModel	LightingModel	-	
changeShader	ShadingModel	-	
changeRoomSize	Size, \mathbb{R}	-	

18.4 Semantics

18.4.1 State Variables

height : \mathbb{R}

width : \mathbb{R}

depth : \mathbb{R}

obs : Observer

ls : LightSource

os : Object [Your scenes have just one object? That is fine, but I want to make sure that is your intention. —SS]

lightModel : {DIFFUSE, HALF-LAMBERT, PHONG, BLINN-PHONG}

18.4.2 Environment Variables

N/A

18.4.3 Assumptions

N/A

18.4.4 Access Routine Semantics

$\text{initScene}(h : \mathbb{R}, w : \mathbb{R}, d : \mathbb{R}, o : \text{Object}, l : \text{LightSource}, ob : \text{Observer}, lm : \{\text{DIFFUSE}, \text{HALF-LAMBERT}, \text{PHONG}, \text{BLINN-PHONG}\})$:

- transition: height, width, depth, obs, ls, os, lightModel := h, w, d, ob, l, o, lm
- output := self
- exception: $\text{exc} := \{(h \leq \text{SCENE_MIN_H} \implies \text{HEIGHT_TOO_SMALL})$
 $\mid (h \geq \text{SCENE_MAX_H} \implies \text{HEIGHT_TOO_LARGE})$
 $\mid (w \leq \text{SCENE_MIN_W} \implies \text{WIDTH_TOO_SMALL})$
 $\mid (w \geq \text{SCENE_MAX_W} \implies \text{WIDTH_TOO_LARGE})$
 $\mid (d \leq \text{SCENE_MIN_D} \implies \text{DEPTH_TOO_SMALL})$
 $\mid (d \geq \text{SCENE_MAX_D} \implies \text{DEPTH_TOO_LARGE})$
 $\mid (\neg \text{objectInScene}(o)) \implies \text{INVALID_OBJECT_POSITION}$
 $\mid (\neg \text{lightInScene}(l)) \implies \text{INVALID_LIGHT_POSITION}$
 $\mid (\neg \text{obsvInScene}(obs)) \implies \text{INVALID_OBSV_POSITION}$
 $\}$

18.4.5 Local Functions

$\text{objectInScene}(o : \text{Object}) \equiv (\text{SCENE_MIN_H} \leq o.\text{position}.y \leq \text{SCENE_MAX_H}) \wedge (\text{SCENE_MIN_W} \leq o.\text{position}.x \leq \text{SCENE_MAX_W}) \wedge (\text{SCENE_MIN_D} \leq o.\text{position}.z \leq \text{SCENE_MAX_D})$

$\text{lightInScene}(l : \text{LightSource}) \equiv (\text{SCENE_MIN_H} \leq l.\text{position}.y \leq \text{SCENE_MAX_H}) \wedge (\text{SCENE_MIN_W} \leq l.\text{position}.x \leq \text{SCENE_MAX_W}) \wedge (\text{SCENE_MIN_D} \leq l.\text{position}.z \leq \text{SCENE_MAX_D})$

$\text{obsvInScene}(o : \text{Observer}) \equiv (\text{SCENE_MIN_H} \leq o.\text{position}.y \leq \text{SCENE_MAX_H}) \wedge (\text{SCENE_MIN_W} \leq o.\text{position}.x \leq \text{SCENE_MAX_W}) \wedge (\text{SCENE_MIN_D} \leq o.\text{position}.z \leq \text{SCENE_MAX_D})$

19 MIS of VecMath

15

The Vector Math module is a library of services that can be used with Vectors. All functions here take in 2 Vectors and output either a Vector or a scalar value. [This is fine. Earlier I pointed out that your Vector ADT did not have a cross product. I see now that you have a library for these calculations. You should use the Uses clause as it was intended so that the reader can find the required information for one module by reading that module's spec, and the spec for the Uses modules. —SS]

19.1 Module

VecMath

19.2 Uses

19.3 Syntax

19.3.1 Exported Constants

N/A

19.3.2 Exported Access Programs

Name	In	Out	Exceptions
add	Vector, Vector	Vector	—
sclMult	Vector, \mathbb{R}	Vector	—
dot	Vector, Vector	\mathbb{R}	—
cross	Vector, Vector	Vector	—
angleBetween	Vector, Vector	rad	—

19.4 Semantics

19.4.1 State Variables

19.4.2 Environment Variables

N/A

19.4.3 Assumptions

19.4.4 Access Routine Semantics

$\text{add}(v1 : \text{Vector}, v2 : \text{Vector})$:

- output: $\text{Vector}((v1.x+v2.x), (v1.y+v2.y), (v1.z, v2.z), \sqrt{(v1.x + v2.x)^2 + (v1.y + v2.y)^2 + (v1.z, v2.z)^2})$
- exception: `exc :=`

[Your add will have a problem because the three components are unlikely to form a unit vector. —SS]

`sclMult(v1 : Vector, r : ℝ):`

- output: $ux := r \times v1.x$
 $uy := r \times v1.y$
 $uz := r \times v1.z$
- exception:

[You do not have access to the state variables here. Also, these will not be unit vectors, except in very rare cases. Your output should call the Vector constructor with the right parameters. You are also missing the start point, which is something you made part of the Vector type. —SS]

`dot(v1 : Vector, v2 : Vector):`

- output: $ux := v1.x \times v2.x$
 $uy := v1.y \times v2.y$
 $uz := v1.z \times v2.z$
- exception:

[This isn't the definition of dot product. Dot product is a scalar (real) value. —SS]

`cross(v1 : Vector, v2 : Vector):`

- output: $ux := (v1.y \times v2.z) - (v1.z \times v2.y)$
 $uy := (v1.z \times v2.x) - (v1.x \times v2.z)$
 $uz := (v1.x \times v2.y) - (v1.y \times v2.x)$
- exception:

[You need to call your Vector constructor with the appropriate arguments. —SS]

`angleBetween(v1 : Vector, v2 : Vector):`

- output: $\cos^{-1}(\frac{\text{dot}(v1, v2)}{v1.m \times v2.m})$
- exception:

19.4.5 Local Functions

N/A

20 MIS of Shader

16

The Shader module is a library that calculates the normal map of an object given a shading model and said object. It handles the different types of shadings that are possible and handles the interpolation of normals between points.

20.1 Module

Shader

20.2 Uses

20.3 Syntax

20.3.1 Exported Constants

N/A

20.3.2 Exported Access Programs

Name	In	Out	Exceptions
interpolate	(Point3D, Vector), (Point3D, Vector), Point3D	Vector	–
findNormals	ShadingModel, Ob- ject	NormalMap	–

20.4 Semantics

20.4.1 State Variables

N/A

20.4.2 Environment Variables

N/A

20.4.3 Assumptions

20.4.4 Access Routine Semantics

interpolate(s: (Point3D, Vector), e: (Point3D, Vector), p: Point3D):

- output:= Linear interpolation of normal values between starting vertex (s[0]) and ending vertex (s[1]).
- exception:

[It would be nice to see the formula for the interpolation. It is a simple formula, but I have no idea how it relates to the arguments that are provided. More information is needed to make it clear. What is being interpolated? Is it the vector at the two points? Is the interpolation component wise? —SS]

findNormals(s:ShadingModel, o:Object):

- output: ns : NormalMap := (s == FLAT \implies all points on the mesh have a normal equal to their polygon's surface normal.

$\forall q:Point3D, \exists p:Polygon \mid q \in p.getPoints() \wedge p \in o.Mesh.Surfaces() \rightarrow (q, p.s_norm)$

| s == GOURAUD \implies all vertices on the mesh have a normal equal to the average of the surface normals of the polygons they are a part of. The normals of the points in between the vertices are not calculated.

$\forall v : Point3D \mid v \in o.Mesh.Vertices() \rightarrow \forall p : Polygon \mid v \in p.getPoints()$

begin:

sum := +(p.s_norm) — Add the surface norms together.

counter++ — Count how many polygons are a part of this.

end to (v, sum/counter)

[Since you are not using the notation we used in class, I do not know how to read this. Does the arrow mean implication? Is this a boolean expression? It is great that you are working on the math, but this might be a case where a pseudo code algorithm is more practical. —SS]

| s == PHONG \implies all vertices on the mesh have a normal equal to the average of the surface normals of the polygons they are a part of. The normals of the points in between the vertices of a polygon are calculated by interpolating their values between the vertices.

begin:

1. $ns := ns \cup (\forall v : Point3D \mid v \in o.Mesh.Vertices() \rightarrow \forall p : Polygon \mid v \in p.getPoints())$

begin:

sum := +(p.s_norm) — Add the surface norms together.

counter++ — Count how many polygons are a part of this.

end $\rightarrow (v, sum/counter)$

2. $ns := ns \cup (\forall \text{ start, end, } p : \text{Point3D} \mid \text{start, end, } p \in o.\text{Mesh.pointsOnMesh}() \wedge \text{start, end} \in o.\text{Mesh.Vertices}() \wedge p \notin o.\text{Mesh.Vertices}() \rightarrow (p, \text{interpolate}((\text{start,}, (\text{end, }, p))))$

end
)

- exception: –

21 MIS of LightingModel

17

The LightingModel module is a library that provides the intensity functions for final scene colouring calculations.

21.1 Module

LightModel

21.2 Uses

21.3 Syntax

21.3.1 Exported Constants

models := {DIFFUSE, HALF-LAMBERT, PHONG, BLINN-PHONG}

21.3.2 Exported Access Programs

Name	In	Out	Exceptions
intensity	models	LightSource, Object $\rightarrow \lambda I$	

21.4 Semantics

21.4.1 State Variables

N/A

21.4.2 Environment Variables

N/A

21.4.3 Assumptions

21.4.4 Access Routine Semantics

intensity(l: LightSource, o: Object):

- output:= { model == DIFFUSE $\implies \lambda l, o \rightarrow$ a function where $((l.direction() \bullet o.intersects(l.direction()))(I_{L_i p}) \cdot (l.colour()))$
| model == HALF-LAMBERT $\implies \lambda l, o \rightarrow$ a function where $[(obj.intersects(l.direction()) \bullet l.direction()) \cdot obj.k_d + (1 - obj.k_d)]^2$
| model == PHONG $\implies \lambda l, o \rightarrow i(p, p_0) \cdot k_a + k_d \cdot i(p, p_0) \cdot \max(0, (L_i \bullet N)) + k_s \cdot i(p, p_0) \cdot \max(0, (L_r \bullet V))^\alpha$
| model == BLINN-PHONG $\implies \lambda l, o \rightarrow$ a function where $i(p, p_0) \cdot o.ka + o.kd \cdot \max(0, (N \bullet l.direction())) \cdot i(p, p_0) + o.ks \cdot \max(0, (N \bullet H))^{o.alpha} \cdot i(p, p_0)$ }

[I think I can see what you are getting at, but the math notation needs some work. In particular, I don't know what the big dots mean. Is this multiplication? Is it a dot product? Do the types all make sense for whatever operation the big dot implies? —SS]

21.4.5 Local Functions

N/A

References

- Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.
- Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL <http://citeseer.ist.psu.edu/428727.html>.

22 Appendix