

# Module Interface Specification for Lighting Models

Sasha Soraine

December 1, 2019

# 1 Revision History

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

## 2 Symbols, Abbreviations and Acronyms

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

# Contents

<b>1</b>	<b>Revision History</b>	<b>i</b>
<b>2</b>	<b>Symbols, Abbreviations and Acronyms</b>	<b>ii</b>
<b>3</b>	<b>Introduction</b>	<b>1</b>
<b>4</b>	<b>Notation</b>	<b>1</b>
<b>5</b>	<b>Module Decomposition</b>	<b>2</b>
<b>6</b>	<b>MIS of Input Parameters Module</b>	<b>3</b>
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
<b>7</b>	<b>MIS of Output Parameters Module</b>	<b>5</b>
7.1	Module . . . . .	5
7.2	Uses . . . . .	5
7.3	Syntax . . . . .	5
7.3.1	Exported Constants . . . . .	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 . . . . .	5
<b>8</b>	<b>MIS of Point3D</b>	<b>6</b>
8.1	Template Module . . . . .	6
8.2	Uses . . . . .	6
8.3	Syntax . . . . .	6
8.3.1	Exported Types . . . . .	6
8.3.2	Exported Access Programs . . . . .	6

8.4	Semantics . . . . .	6
8.4.1	State Variables . . . . .	6
8.4.2	Environment Variables . . . . .	6
8.4.3	Assumptions . . . . .	7
8.4.4	Access Routine Semantics . . . . .	7
8.4.5	Local Functions . . . . .	7
<b>9</b>	<b>MIS of Colour</b>	<b>8</b>
9.1	Template Module . . . . .	8
9.2	Uses . . . . .	8
9.3	Syntax . . . . .	8
9.3.1	Exported Types . . . . .	8
9.3.2	Exported Access Programs . . . . .	8
9.4	Semantics . . . . .	8
9.4.1	State Variables . . . . .	8
9.4.2	Environment Variables . . . . .	8
9.4.3	Assumptions . . . . .	9
9.4.4	Access Routine Semantics . . . . .	9
9.4.5	Local Functions . . . . .	10
<b>10</b>	<b>MIS of Vector</b>	<b>11</b>
10.1	Template Module . . . . .	11
10.2	Uses . . . . .	11
10.3	Syntax . . . . .	11
10.3.1	Exported Types . . . . .	11
10.3.2	Exported Access Programs . . . . .	11
10.4	Semantics . . . . .	11
10.4.1	State Variables . . . . .	11
10.4.2	Environment Variables . . . . .	11
10.4.3	Assumptions . . . . .	12
10.4.4	Access Routine Semantics . . . . .	12
10.4.5	Local Functions . . . . .	12
<b>11</b>	<b>MIS of Light Type</b>	<b>13</b>
11.1	Template Module . . . . .	13
11.2	Uses . . . . .	13
11.3	Syntax . . . . .	13
11.3.1	Exported Types . . . . .	13
11.3.2	Exported Access Programs . . . . .	13
11.4	Semantics . . . . .	13
11.4.1	State Variables . . . . .	13
11.4.2	Environment Variables . . . . .	13
11.4.3	Assumptions . . . . .	13

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

<b>15 MIS of Observer</b>	<b>23</b>
15.1 Template Module . . . . .	23
15.2 Uses . . . . .	23
15.3 Syntax . . . . .	23
15.3.1 Exported Types . . . . .	23
15.3.2 Exported Access Programs . . . . .	23
15.4 Semantics . . . . .	23
15.4.1 State Variables . . . . .	23
15.4.2 Environment Variables . . . . .	23
15.4.3 Assumptions . . . . .	23
15.4.4 Access Routine Semantics . . . . .	24
15.4.5 Local Functions . . . . .	24
<b>16 MIS of NormalMap</b>	<b>25</b>
16.1 Template Module . . . . .	25
16.2 Uses . . . . .	25
16.3 Syntax . . . . .	25
16.3.1 Exported Types . . . . .	25
16.3.2 Exported Access Programs . . . . .	25
16.4 Semantics . . . . .	25
16.4.1 State Variables . . . . .	25
16.4.2 Environment Variables . . . . .	25
16.4.3 Assumptions . . . . .	25
16.4.4 Access Routine Semantics . . . . .	25
16.4.5 Local Functions . . . . .	26
<b>17 MIS of Object</b>	<b>27</b>
17.1 Template Module . . . . .	27
17.2 Uses . . . . .	27
17.3 Syntax . . . . .	27
17.3.1 Exported Types . . . . .	27
17.3.2 Exported Access Programs . . . . .	28
17.4 Semantics . . . . .	28
17.4.1 State Variables . . . . .	28
17.4.2 Environment Variables . . . . .	29
17.4.3 Assumptions . . . . .	29
17.4.4 Access Routine Semantics . . . . .	29
17.4.5 Local Functions . . . . .	32
<b>18 MIS of Scene Module</b>	<b>33</b>
18.1 Module . . . . .	33
18.2 Uses . . . . .	33
18.3 Syntax . . . . .	33

18.3.1	Exported Constants	33
18.3.2	Exported Access Programs	33
18.4	Semantics	34
18.4.1	State Variables	34
18.4.2	Environment Variables	34
18.4.3	Assumptions	34
18.4.4	Access Routine Semantics	34
18.4.5	Local Functions	34
<b>19</b>	<b>MIS of VecMath</b>	<b>36</b>
19.1	Module	36
19.2	Uses	36
19.3	Syntax	36
19.3.1	Exported Constants	36
19.3.2	Exported Access Programs	36
19.4	Semantics	36
19.4.1	State Variables	36
19.4.2	Environment Variables	36
19.4.3	Assumptions	36
19.4.4	Access Routine Semantics	36
19.4.5	Local Functions	37
<b>20</b>	<b>MIS of Shader</b>	<b>38</b>
20.1	Module	38
20.2	Uses	38
20.3	Syntax	38
20.3.1	Exported Constants	38
20.3.2	Exported Access Programs	38
20.4	Semantics	38
20.4.1	State Variables	38
20.4.2	Environment Variables	38
20.4.3	Assumptions	38
20.4.4	Access Routine Semantics	38
<b>21</b>	<b>MIS of LightingModel</b>	<b>40</b>
21.1	Module	40
21.2	Uses	40
21.3	Syntax	40
21.3.1	Exported Constants	40
21.3.2	Exported Access Programs	40
21.4	Semantics	40
21.4.1	State Variables	40
21.4.2	Environment Variables	40



21.4.3	Assumptions . . . . .	40
21.4.4	Access Routine Semantics . . . . .	40
21.4.5	Local Functions . . . . .	41
<b>22</b>	<b>Appendix</b>	<b>43</b>

### 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 ?, with the addition that template modules have been adapted from ?. The mathematical notation comes from Chapter 3 of ?. 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.

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

??

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	

## 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)
- 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():

- 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

??

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

### 7.1 Module

Input Parameters

### 7.2 Uses

### 7.3 Syntax

#### 7.3.1 Exported Constants

#### 7.3.2 Exported Access Programs

Name	In	Out	Exceptions
convert	String	-	INVALID_FILE_NAME, NO_DATA_TO_WRITE

### 7.4 Semantics

#### 7.4.1 State Variables

writing : (Scene, Object, LightSource, Observer)

#### 7.4.2 Environment Variables

output: File

#### 7.4.3 Assumptions

N/A

#### 7.4.4 Access Routine Semantics

convert(o: String):

- output: output := OpenFile(o) and ToJSON(writing)
- exception: exc := {o already exists  $\implies$  INVALID\_FILE\_NAME  
| writing is empty  $\implies$  NO\_DATA\_TO\_WRITE  
}

#### 7.4.5 Local Functions

N/A

## 8 MIS of Point3D

??

The Point3D module captures the structure of a 3D Cartesian 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:=  $\text{sqrt}(p.x - \text{self}.x)^2 + (p.y - \text{self}.y)^2 + (p.z - \text{self}.z)^2$
- exception: N/A

### 8.4.5 Local Functions

N/A



## 9 MIS of Colour

??

The Colour module captures the structure of colours used in this program.

### 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

??

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, INVALID_UY, INVALID_UZ, IN- VALID_M
.m		$\mathbb{R}$	–
direction		$\mathbb{R}, \mathbb{R}, \mathbb{R}$	–

### 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 }

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

.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

??

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	$\mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$	–

### 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:  $\text{self.name} := \text{inName}$   
 $\text{self.i} := (\text{name} == \text{ambient} \implies \lambda d, i_0 \rightarrow i_0$   
 $\quad | \text{name} == \text{directional} \implies \lambda d, i_0 \rightarrow \frac{1}{d^2} i_0$   
 $\quad | \text{name} == \text{point} \implies \lambda d, i_0 \rightarrow i_0$   
 $\quad | \text{name} == \text{spotlight} \implies )$
- output:  $\text{self}$
- exception:  $\text{exc} := \{\text{inName} \notin \text{ambient, spotlight, point, directional} \implies \text{INVALID\_LIGHT\_TYPE}\}$

$\text{.name}()$ :

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

$\text{.i}()$ :

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

#### 11.4.5 Local Functions

N/A

## 12 MIS of Polygon

??

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) <sup>n</sup>	–	–
.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}

bounds := Set of (Point3D, Vector) tuples

s\_norm := Vector

#### 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:=  $shape := t$ ;  
   $bounds := \cup(p, v)$   
   $s\_norm :=$  Calculate norm as cross-product of two vectors from 1 vertex.
- exception:  $exc := \{(t \notin \{triangle, quad\} \implies INVALID\_SHAPE) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == triangle, sizeOfBounds < 6 \implies TOO\_FEW\_POINTS) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == triangle, sizeOfBounds > 6 \implies TOO\_MANY\_POINTS) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == quad, sizeOfBounds > 8 \implies TOO\_MANY\_POINTS) \mid (t:\{triangle, quad\}, b: \text{Set of } (Point3D, Vector) \mid t == quad, sizeOfBounds < 8 \implies TOO\_FEW\_POINTS) \}$

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

getPoints():

This method retrieves the set of points in the polygon.

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

#### 12.4.5 Local Functions

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

## 13 MIS of Mesh

??

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

#### 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

??

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

### 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		$\mathbb{R}$	

### 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

??

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	–

### 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:  $\text{exc} :=$

#### 15.4.5 Local Functions

N/A

## 16 MIS of NormalMap

??

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	–

### 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 := inP  
d := inD
- output := self
- exception: exc :=

getNormal(p:Point3D):

- output :=
- exception: exc :=

#### **16.4.5 Local Functions**

N/A

## 17 MIS of Object

??

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, GOUBAUD, 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  
 $nMap := \text{Shader.findNormals}(shade, self)$
- 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:  $\text{exc} :=$   
 $\text{c.r} > 255 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.g} > 255 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.b} > 255 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.r} < 1 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.g} < 1 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.b} < 1 \implies \text{IV\_OUT\_OF\_BOUNDS}$

SetObj\_SpecColour(c : Colour):

- transition:  $\text{specColour} := \text{c}$
- exception:  $\text{exc} :=$   
 $\text{c.r} > 255 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.g} > 255 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.b} > 255 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.r} < 1 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.g} < 1 \implies \text{IV\_OUT\_OF\_BOUNDS}$   
 $|$   
 $\text{c.b} < 1 \implies \text{IV\_OUT\_OF\_BOUNDS}$

SetObj\_kd(d:  $\mathbb{R}$ ):

- transition:  $\text{kd} := \text{d}$
- exception:  $\text{exc} :=$   
 $\text{d} > 1 \implies \text{COEFFICIENT\_TOO\_HIGH}$   
 $\text{d} < 0.5 \implies \text{COEFFICIENT\_TOO\_LOW}$

SetObj\_ka(a:  $\mathbb{R}$ ):

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



- exception:  $\text{exc} :=$   
 $a > 1 \implies \text{COEFFICIENT\_TOO\_HIGH}$   
 $a < 0 \implies \text{COEFFICIENT\_TOO\_LOW}$

$\text{SetObj\_ks}(s: \mathbb{R})$ :

- transition:  $\text{ks} := s$
- exception:  $\text{exc} :=$   
 $s > 1 \implies \text{COEFFICIENT\_TOO\_HIGH}$   
 $s < 0 \implies \text{COEFFICIENT\_TOO\_LOW}$

$\text{SetObj\_alpha}(a: \mathbb{N})$ :

- transition:  $\text{alpha} := a$
- exception:  $\text{exc} :=$   
 $a < 0 \implies \text{COEFFICIENT\_TOO\_LOW}$

$\text{SetObj\_NormalMap}()$ :

- 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

??

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}		

## 18.4 Semantics

### 18.4.1 State Variables

height :  $\mathbb{R}$   
width :  $\mathbb{R}$   
depth :  $\mathbb{R}$   
obs : Observer  
ls : LightSource  
os : Object  
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

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

- transition: height, width, depth, obs, ls, os, lightModel := h, w, d, ob, l, o, lm
- output := self
- exception: exc :=  $\{(h \leq \text{SCENE\_MIN\_H} \implies \text{HEIGHT\_TOO\_SMALL})$   
|  $(h \geq \text{SCENE\_MAX\_H} \implies \text{HEIGHT\_TOO\_LARGE})$   
|  $(w \leq \text{SCENE\_MIN\_W} \implies \text{WIDTH\_TOO\_SMALL})$   
|  $(w \geq \text{SCENE\_MAX\_W} \implies \text{WIDTH\_TOO\_LARGE})$   
|  $(d \leq \text{SCENE\_MIN\_D} \implies \text{DEPTH\_TOO\_SMALL})$   
|  $(d \geq \text{SCENE\_MAX\_D} \implies \text{DEPTH\_TOO\_LARGE})$   
|  $(\neg \text{objectInScene}(o)) \implies \text{INVALID\_OBJECT\_POSITION}$   
|  $(\neg \text{lightInScene}(l)) \implies \text{INVALID\_LIGHT\_POSITION}$   
|  $(\neg \text{obsvInScene}(\text{obs})) \implies \text{INVALID\_OBSV\_POSITION}$   
 $\}$

### 18.4.5 Local Functions

objectInScene( $o$  : Object)  $\equiv (\text{SCENE\_MIN\_H} < o.\text{position.y} < \text{SCENE\_MAX\_H}) \wedge$   
 $(\text{SCENE\_MIN\_W} < o.\text{position.x} < \text{SCENE\_MAX\_W}) \wedge (\text{SCENE\_MIN\_D} < o.\text{position.z}$   
 $< \text{SCENE\_MAX\_D})$

$\text{lightInScene}(l : \text{LightSource}) \equiv (\text{SCENE\_MIN\_H} < l.\text{position}.y < \text{SCENE\_MAX\_H}) \wedge$   
 $(\text{SCENE\_MIN\_W} < l.\text{position}.x < \text{SCENE\_MAX\_W}) \wedge (\text{SCENE\_MIN\_D} < l.\text{position}.z$   
 $< \text{SCENE\_MAX\_D})$

$\text{obsvInScene}(o : \text{Observer}) \equiv (\text{SCENE\_MIN\_H} < o.\text{position}.y < \text{SCENE\_MAX\_H}) \wedge$   
 $(\text{SCENE\_MIN\_W} < o.\text{position}.x < \text{SCENE\_MAX\_W}) \wedge (\text{SCENE\_MIN\_D} < o.\text{position}.z$   
 $< \text{SCENE\_MAX\_D})$

## 19 MIS of VecMath

??

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.

### 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:  $\text{exc} :=$

$\text{sclMult}(v1 : \text{Vector}, r : \mathbb{R})$ :

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

- exception:

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

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

- exception:

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

- 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:

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

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

### 19.4.5 Local Functions

N/A

## 20 MIS of Shader

??

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, Object	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:

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

| 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  $\cup (\forall start, end, p : Point3D \mid start, end, p \in o.Mesh.pointsOnMesh() \wedge start, end \in o.Mesh.Vertices() \wedge p \notin o.Mesh.Vertices() \rightarrow (p, interpolate((start, ), (end, ), p)))$*

*end*

)

- exception: —



## 21 MIS of LightingModel

??

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  $((l.direction() \bullet o.intersects(l.direction()))(I_{Lip}) \cdot (l.colour()))$   
| 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  $\implies \lambda \in [0, 1]$ ,  $\alpha \rightarrow$  a function where  $i(p, p_0) \cdot \text{o.ka} + \text{o.kd} \cdot \max(0, (N \bullet l.direction)) \cdot i(p, p_0) + \text{o.ks} \cdot \max(0, (N \bullet H))^{\alpha.alpha} \cdot i(p, p_0)$  }

#### 21.4.5 Local Functions

N/A

## References

## 22 Appendix