

Meridian Rendering Software

Developer Guide

January 10, 2012

Version 2.0.1

Copyright © 2011 Sunfish Studio, LLC

All rights reserved.

Meridian™, MeridianSL™ and RXF™ are trademarks of Sunfish Studio, LLC.

Protected by one or more of the following patents, or by other patents in the United States or other countries, or by patents pending:

US 7949700

US 7554540

US 7250948

AU 2003294327

NZ 563047

NZ 540742

JP 4643271

Contents

Introduction	1
Parallel Rendering.....	2
Design Goals.....	3
Features	3
Advanced Capabilities.....	4
Roadmap.....	5
About This Document	5
Render Exchange Format (RXF)	6
Introduction	6
Mathematical Conventions.....	7
Coordinate Systems	7
Lexical and Logical Structure.....	9
Grammar Notation.....	9
Relationship to the Meridian Shading Language	11
Shaders	11
Light	12
Surface	13
Volume.....	15
Displacement	15
Shader Initialization	16
Construction.....	16
Dynamic Constants and Variables	17
Native Interfaces.....	19
Samplers.....	20
Ranges.....	20
Lexical Structure	21
Line Terminators.....	21

Input Elements and Tokens	21
White Space	22
Comments	22
Tokens	22
Identifiers	22
Literals	23
Boolean Literals	23
Integer Literals	23
Floating-Point Literals	24
String Literals	24
Data Types	26
Basic Types	26
Compound Types	26
Point	27
ControlPoint	27
Color	27
Matrix	28
Quaternion	28
Rect	28
Blocks	30
Program	30
Frame	31
Raster	32
NDC to Pixel Transformation	34
Projection	34
Ortho	35
Perspective	36
Stereo	37
Lattice	37
Tile	38
Model	39
Transform	39

Motion	40
ShadingGroup	42
Linker	42
Displays	43
Display Drivers	44
Dynamic Range and Quantization.....	44
Color Options	45
Depth Options.....	46
Attributes	47
Shading Attributes	47
Native Interface	48
Shader	49
Surface Parameter	51
Channel	52
Rendering Attributes.....	53
Brush	53
Displace.....	54
DoubleSided.....	54
Opposite.....	55
ShadingRate	55
Timebase.....	55
TransparencyScale	56
Geometric Primitives	57
Channels.....	57
Channel Type	57
Freeform Surfaces.....	58
BSpline	58
Patch	60
TrianglePatch	62
Polygons.....	63
Face	63
Quad.....	64

Lightweight Primitives	65
Particles.....	65
Ribbon.....	66
Tube	68
Miscellaneous Commands	72
Synchronization Commands	72
Finish	72

Introduction

Welcome to the world of parallel processing. Welcome to the world of Meridian.

Nathan T. Hayes

The semiconductor industry is changing, and the computer graphics industry is changing along with it. Gone are the days when processor speeds doubled every year and software programs enjoyed the benefit of improved application performance for free.

We are now entering the many-core era: on the horizon is a foreseeable future of computers with an ever increasing number of processing cores, not clock speeds. Commodity desktop computers of today already have two, four or eight processing cores. Tomorrow they will have dozens, perhaps even hundreds. In ten years, a chip with a thousand processing cores is feasible.

The dominant computing paradigm is shifting from sequential to parallel processing. In the past, single-threaded applications enjoyed the benefit of ever increasing processor clock speeds. But this is no longer true. To harness the full potential of new hardware, software needs to be parallel.

This is why we have developed Meridian rendering software. Parallel to its core, Meridian is a massively concurrent architecture designed to harness the full theoretical capacity of computers with hundreds of processor cores. Based on a new patented rendering method that uses interval arithmetic, Meridian analytically renders geometry like NURBS directly into perfectly anti-aliased pixels. Since there is no longer any need to tessellate geometry into dense polygon meshes of tiny pixel-sized polygons, each tile in an image can be processed using only a very small and constant amount of memory regardless of the scene size and complexity.

This makes the Meridian rendering method highly parallel and an ideal software configuration for modern many-core computers. As dozens or hundreds of processor cores are added to a single rendering machine, users can expect that Meridian will keep the cores busy even on the most demanding scenes.

Parallel Rendering

Meridian is an immediate-mode rendering engine based on a tile-centric streaming interface. This means an animated scene is partitioned into frames, and each frame is partitioned into a lattice of image tiles. Each tile of each frame is then streamed to Meridian and rendered concurrently “on the fly.”

Figure 1 depicts the method Meridian uses to manage all of the available processors on a machine for optimal parallel performance. Tiles are added to a queue and then concurrently rendered by active processors as older tiles in the data stream are completed. Even if tiles in one frame take a very long time to render, tiles from newer frames in the data stream may be queued and rendered as the number of active processors and rendering conditions permit.

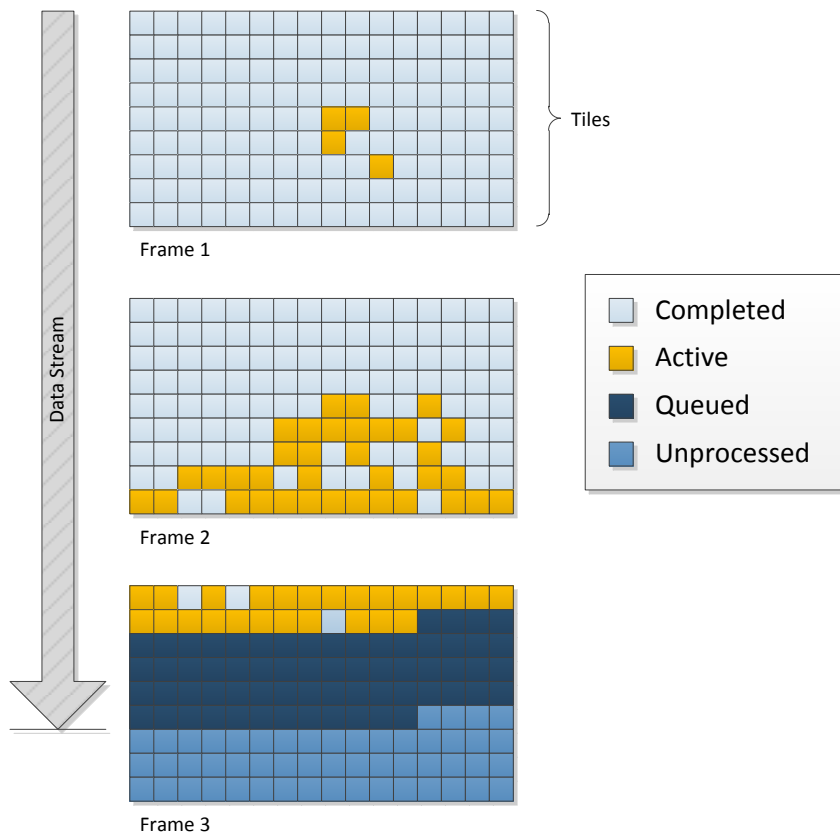


Figure 1. An example of several frames of animation being rendered concurrently on a computer with 64 processors.

Due to the special parallel properties of the Meridian interval arithmetic rendering engine, the number of concurrently rendered tiles is practically limitless. Even the most demanding scenes with features such as transparency and motion blur should not reduce the ability of Meridian to take advantage of a large number of processing cores.

Design Goals

Meridian has been designed specifically for the industrial production of high-quality computer generated animation. It is intended primarily to be used on large networks or “render farms” of computers as a rendering solution for heavy production jobs involving complex scenes and imagery.

Below is a list of design goals that inspired the development of the Meridian rendering software:

- **Stability.** In a production environment, a crashed computer is a liability measured in time, effort and revenue. The software must be dependable and reliable, even under the most demanding and challenging workloads.
- **Parallel performance.** We are now in the multi-core era. The software must be highly parallel and capable of scaling to machines with hundreds of processing cores.
- **Capacity.** The software should be able to process dense scenes with practically an unlimited number of primitives.
- **Image quality.** Display of rendered images on movie screens and high definition monitors requires the highest quality output.
- **Ease of use.** Interval algorithms are by nature adaptive, so the software should not require large amounts of esoteric knobs, settings or manual tweaking to run effectively even under a wide range of operating conditions.

Users should expect that Meridian will be a reliable workhorse in the most challenging production environments.

Features

Meridian offers standard rendering features:

- Hierarchical attribute state and active light lists
- Orthographic and perspective viewing transformations
- Depth-based hidden-surface elimination
- High-quality anti-aliased rendering
- Image levels and gamma correction before quantization
- Output of color, alpha and depth image files
- Support for a versatile list of geometric primitives, including freeform surfaces and polygons
- Shading calculations are defined by user-supplied programs written in Meridian Shading Language
- Highly efficient memory mapped files for texture, environment and shadow map access

Advanced Capabilities

In addition to standard rendering features, Meridian offers some advanced capabilities:

- Massively parallel architecture
- Support for lightweight primitives: ribbons, tubes and particles
- Truly nonlinear motion blur of rigid bodies
- Displacement shaders can compute true surface relief that modifies the shape of a geometric surface along its surface normal
- Arbitrary output variables
- Patent-pending Variance Shadow Maps using beta distributions provide pre-filtered, transparent and motion blurred shadows
- Patent-pending order independent, single-pass transparency using a constant memory footprint
- Stereographic viewing transformations for dual-paraboloid environment and shadow map generation

Meridian does not cheat when rendering effects such as motion blur. For example, some rendering engines tessellate moving objects into a dense polygon mesh. Shading is performed on the mesh only at the shutter-open time and the polygons in the mesh are then “dragged” across the screen to simulate a motion blur effect. This may cause strange artifacts in the rendered image, such as a shadow that sticks to a moving object even though by the end of the exposure the object should be fully illuminated.

Meridian is an analytical rendering engine. It does not tessellate objects into polygonal meshes, and each visible pixel of an object is always shaded at the actual time the object is visible in the pixel. This approach ensures the highest image quality.

On a single processor, it may take Meridian a longer time to render motion blur than other rendering engines that simply drag polygons across the screen. But as more processors are added to the computer (even dozens or hundreds), Meridian is able to keep the processors busy. By contrast, other rendering engines that use dense polygon meshes typically begin to suffer the effects of Amdahl’s Law, so no further gains in speed can be achieved after adding just a few processors.

Meridian has been designed to produce uncompromising image fidelity by harnessing the full theoretical potential of modern many-core computers that have dozens or even hundreds of processing cores. This makes Meridian an ideal solution for users that want the best image quality rendered in the fastest amount of time on computers with a very large number of processors.

Roadmap

The Sunfish team is always working to improve the Meridian software. A list of capabilities expected in future releases include:

- Depth of field
- Soft-body motion blur
- Global illumination
- Implicit “blobby” surfaces
- Volumetric rendering

Of course, we aim to please and always like to hear feature requests and suggestions from loyal customers.

About This Document

This document is a technical reference aimed at developers who wish to learn how to program and integrate Meridian into a custom production pipeline. The next chapter introduces Render Exchange Format (RXF), the basic rendering interface to Meridian. Subsequent chapters cover more detailed aspects of RXF, including block structure, geometric primitives and attributes. The basic relationship between RXF and Meridian Shading Language is covered in this document, but a full technical specification of the shading language is provided in the [Meridian Shading Language Specification](#).

Render Exchange Format (RXF)

Render Exchange Format (RXF) describes the contents of a scene to Meridian. In order to render a picture, Meridian must be supplied with valid RXF data.

Here are a few critical facts about RXF:

- Sunfish Studio, LLC is the author of RXF
- Permission is granted for anyone to write, develop or sell software that creates RXF data
- All rights to render RXF data are reserved by Sunfish Studio, LLC

Introduction

Meridian is an immediate-mode rendering engine based on a tile-centric streaming interface. This means an animated scene is partitioned into frames, and each frame is partitioned into a lattice of image tiles. Each tile of each frame is then streamed to Meridian and rendered “on the fly.”

RXF data is a stream of ASCII characters that encodes a tile-centric scene description comprised of logical elements called *blocks* and *commands*. Blocks define a hierarchical tree structure of nested elements and commands are leaf elements in the tree.

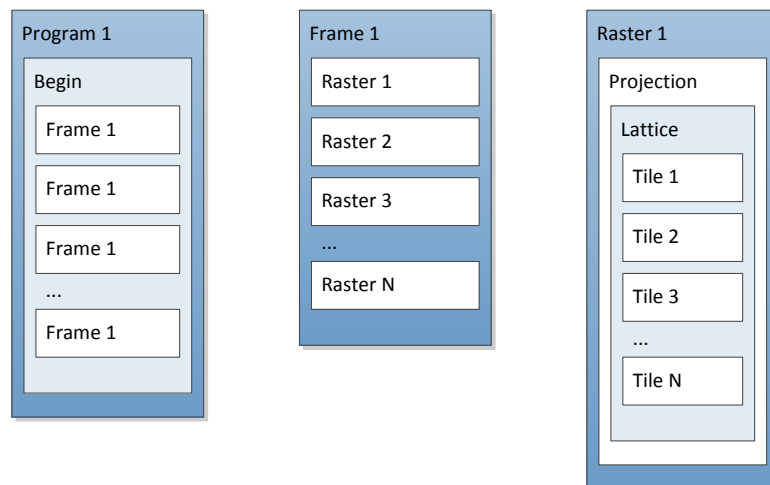


Figure 2. Block structure of RXF data: *programs*, *frames*, *rasters* and *tiles*.

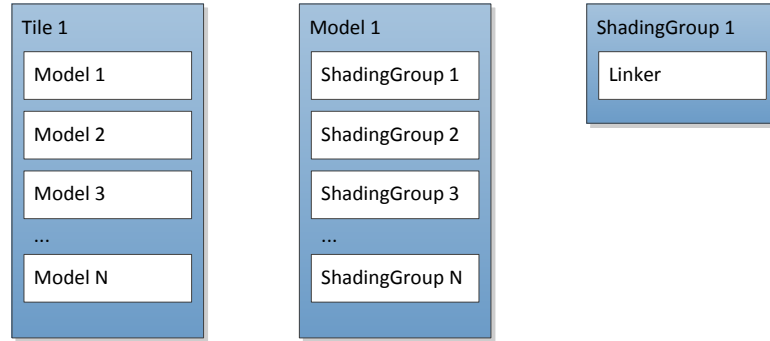


Figure 3. Block structure of tile data: *models*, *shading groups* and *linkers*.

Figure 2 depicts the block structure of RXF data. An RXF stream is comprised of *programs*. Each program has a *begin* block that contains *frames*. User-defined shading routines specified in a program are compiled at the start of the *begin* block and the shading language symbols are then available to all of the blocks nested inside the program. Each frame may contain *rasters*. Several shadow maps and a beauty pass may belong to a single frame, for example. Rasters are the actual rendered images and may be associated with output displays. Each raster contains a projection block that specifies a camera projection and a lattice block that contains all of the raster data in *tiles*.

Figure 3 depicts the block structure of tile data. Each tile contains *models*. Models establish a local coordinate system for an object in the scene and are comprised of *shading groups*. Each shading group delimits a logical subset of the model sharing common shading attributes. A *linker* in each shading group binds shading attributes to actual geometric primitives.

Mathematical Conventions

Meridian uses the right-hand rule for coordinate systems. Transformation matrices follow the standard conventions of most linear algebra textbooks, i. e., points are column vectors multiplied on the right side of the matrix. For example,

$$\begin{bmatrix} 1 & 0 & 0 & A \\ 0 & 1 & 0 & B \\ 0 & 0 & 1 & C \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} X + A \\ Y + B \\ Z + C \\ 1 \end{bmatrix}$$

is an affine translation.

Coordinate Systems

RXF defines several coordinate systems and a transformation pipeline, as depicted in Figure 4.

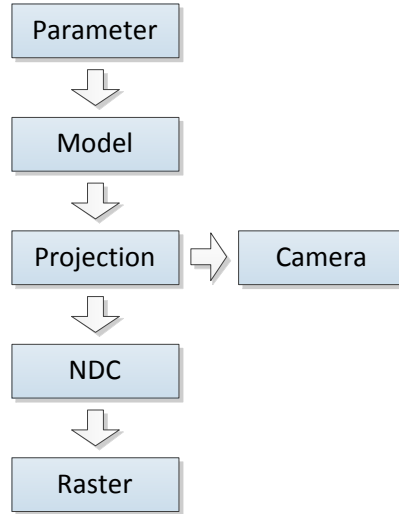


Figure 4. RXF coordinate systems and transformation pipeline.

Parameter space is a coordinate system that parameterizes a geometric object. Curves and surfaces have one and two-dimensional parameter spaces respectively. If a curve or surface is also parameterized in time, as in the case of defining motion, the respective parameter spaces are two and three-dimensional.

Model space is a three-dimensional Cartesian coordinate system that places a geometric object in the scene. Parameter space is mapped into the model coordinate system. Model space is sometimes called *local space*.

Camera space is another three-dimensional Cartesian coordinate system that places a camera in the scene. The viewing position of the camera is at the origin and the direction or line of sight is down the negative z -axis. The positive y -axis is up and the positive x -axis is to the right. Shading calculations often take place in camera space.

Projection space is a transformation of camera space into a convenient or canonical space defined by a particular type of camera projection. Unlike many traditional computer graphics pipelines, Meridian requires that model or local space is always transformed directly into projection space. At render time, points in projection space are transformed back into camera space and then shaded.

Normalized Device Coordinate (NDC) space is also a three-dimensional Cartesian coordinate system. Projection space is mapped to NDC space. A subset of NDC space called the *window* is the set of all NDC points with x and y coordinates in the interval $[-1,1]$. The window may be mapped to a display or a raster of image pixels. All other NDC points are clipped or discarded. The z coordinate of any point in the window is always the reciprocal of the camera-space distance between the nearest visible point in the scene and the center of projection.

Lexical and Logical Structure

RXF data is a stream of ASCII characters that encodes a structure of logical elements:

- The *lexical structure* of RXF is the translation of a stream of ASCII characters into a sequence of terminal symbols called *tokens*
- The *logical structure* of RXF is the translation of a sequence of tokens into programs

Subsequent chapters give detailed definitions of the lexical and logical structures of RXF data.

Grammar Notation

Terminal symbols are shown in `monospace` font, and nonterminal symbols are shown in *italic* type. A nonterminal symbol is defined by specifying the name of the symbol followed by a colon, followed by one or more alternative *productions* each on a separate line. For example

```
SourceFile:
  Source SourceName
```

defines the nonterminal symbol *SourceFile* to be a production that consists of the token `Source` followed by the nonterminal symbol *SourceName*.

Any symbol ending with an “opt” subscript is optional. For example

```
Program:
  Program SourceFilesopt Begin End
```

defines the nonterminal symbol *Program* to be a production that consists of the token `Program`, followed by an optional nonterminal symbol called *SourceFiles*, followed by a nonterminal symbol *Begin* and then followed by a token `End`.

Recursive definitions such as

```
SourceFiles:
  SourceFile
  SourceFiles SourceFile
```

state that *SourceFiles* may represent either a single *SourceFile* or a production that consists of *SourceFiles* followed by a *SourceFile*. The result is that *SourceFiles* may be comprised of one or more nonterminal symbols called *SourceFile*.

If the words “one of” follow the colon of a nonterminal symbol, this is a shorthand notation that allows alternate productions to be specified on a single line. For example

```
Sign: one of  
+ -
```

is shorthand notation for

```
Sign:  
+  
-
```

Productions are sometimes defined by a simple descriptive phrase. For example

```
InputCharacter:  
any ASCII character but not CR or LF
```

specifies any ASCII character that is not a carriage return or linefeed. Similarly, the example

```
MotionNum:  
  Int  
  
MotionOrder:  
  Int  
  
MotionKnots:  
  array of Float[ MotionNum + MotionOrder ]
```

specifies that *MotionKnots* is an array of nonterminal symbols called *Float* and that the number of elements in the array is the sum of the values represented by the two nonterminal symbols *MotionNum* and *MotionOrder*.

Relationship to the Meridian Shading Language

Meridian Shading Language is a full-featured programming language that resembles ANSI C in many respects but also offers some object-oriented features such as class methods and interfaces. The language also supports operator overloading for small vectors of packed data types such as `boolean`, `int` and `float`.

Users can write programs in Meridian Shading Language to extend the functionality of the Meridian rendering engine. RXF provides several mechanisms to pass user-defined data into shading language programs written by users. This includes an ability to attach user-defined data to geometric primitives that is then automatically interpolated by Meridian at render-time and provided on a per-sample basis to shading programs.

This chapter of the document describes the interface between RXF and Meridian Shading Language. Please see the [Meridian Shading Language Specification](#) for a detailed description of the language.



Note. Source code definitions for the Meridian Shading Language classes and interfaces defined in this chapter are part of the standard translation environment of the compiler. Users do not need to include or re-define them in their own programs.

Shaders

Shading is the process of computing the color of a point on a surface. Shading requires a specification of light sources, surface material properties and atmospheric or volumetric effects. *Shaders* are special programs written in the Meridian Shading Language. Shaders allow the entire shading process to be specified in Meridian Shading Language.

RXF defines four standard Meridian Shading Language class interfaces:

Light
Displacement

Surface

Volume

Each of the standard shader class interfaces defines a specific interface users may implement in Meridian Shading Language. The following subsections document these interfaces.

Light

```

struct differentialArea {
    float3 filterSize;
    float3 point;
    float3 normal;
}

struct intensity {
    float3 direction;
    float3 transmittance;
    float3 ambient;
    float3 diffuse;
    float3 specular;
}

interface Light {
    intensity illuminate( differentialArea da ) const;
}

```

Light shaders compute the amount of energy emitted from a light source and arrives at a point on a surface somewhere in the scene.

Users must derive their own light shader from the [Light](#) interface and provide an implementation for the `illuminate` method. The [differentialArea](#) structure is the only formal parameter to the `illuminate` method. All members of [differentialArea](#) are in camera space. The point to be shaded is specified by `point`, and the orientation of the differential area at `point` is provided by `normal`. The `normal` member can be used by the light shader, for example, to compute lookups into diffuse environment maps. The size of the differential area is centered at `point` and bounded in three-dimensional space by `filterSize`.

The `illuminate` method returns an [intensity](#) structure. A vector from the light source to the point being shaded is returned in `direction`. `transmittance` is the fractional amount of energy arriving at this point from the light source that is not occluded or absorbed by objects or participating media. The raw energy contributions of the light source are attenuated by the light shader and returned in the `ambient`, `diffuse` and `specular` members.

```

// Example light shader derived from the standard Light
// interface.
struct omniLight : Light {
    float3 origin;    // Light origin in camera space
    float3 color;     // Light color
    boolean emitSpecular; // Emit specular energy?
    intensity illuminate( differentialArea da ) const {
        intensity out;
        out.direction = da.point - origin;
        float atten = sqr( 1.0 / length( out.direction ) );
        out.transmittance = 1.0;
        out.ambient = 0.0;
        out.diffuse = color * atten;
        out.specular = emitSpecular ? out.diffuse : 0.0;
        return out;
    }
}

```

Surface

```

struct transform {
    float3x3 basis;
    float3 translation;
}

struct rect {
    int x;
    int y;
    int width;
    int height;
}

struct fragment {
    transform from;
    transform to;
    float3 filterSize;
    float3 filterSizeCamera;
    float3 filterSizeNDC;
    float3 point;
    float3 pointCamera;
    float3 pointNDC;
    float3 normal;
    float3 geometricNormal;
    float3 tangentU;
    float3 tangentV;
    float3 tangentTime;
    float3 incident;
    float3 incidentCamera;
    float u;
    float v;
    float time;
    float du;
    float dv;
    float dtime;
    boolean opposite;
    rect raster;
}

interface surface {
    // NOTE: derived classes must declare the standard float3
    // output variables "color" and "opacity" as class
    // members, i.e.,
    //
    // output float3 color;
    // output float3 opacity;
    //
    procedure evaluate(
        fragment in, const Light lights[], const int num ) const;
}

```

Surface shaders are attached to geometric primitives and compute the amount of energy reflected from a surface in a certain direction. Surface shaders often integrate all of the energy arriving at a point on the surface from light sources in the scene.

Users must derive their own surface shader from the [surface](#) interface and provide an implementation for the `evaluate` method. The formal parameter `in` is a [fragment](#) structure that contains state information provided by the renderer.

Table 1 provides a description of the `fragment` class members. An array of `Light` shaders containing `num` elements is also passed into the `evaluate` method.

Type	Member	Description
transform	from	Model to camera space transform
transform	to	Camera to model space transform
float3	filterSize	Filter size in model space
float3	filterSizeCamera	Filter size in camera space
float3	filterSizeNDC	Filter size in NDC space
float3	point	Shading point in model space
float3	pointCamera	Shading point in camera space
float3	pointNDC	Shading point in NDC space
float3	normal	Shading normal in model space
float3	geometricNormal	Geometric normal in model space
float3	tangentU	Tangent with respect to u in model space
float3	tangentV	Tangent with respect to v in model space
float3	tangentTime	Tangent with respect to time in model space
float3	incident	Incident vector in model space
float3	incidentCamera	Incident vector in camera space
float	u	Surface parameter in u
float	v	Surface parameter in v
float	time	Surface parameter in time
float	du	Change in u surface parameter
float	dv	Change in v surface parameter
float	dtime	Change in time surface parameter
boolean	opposite	Are we shading the back of the surface?
rect	raster	Raster origin and dimensions, in pixels

Table 1. Members of the `fragment` structure.

The `evaluate` method is declared as a procedure, so it has no return value. Shading results are passed back to the renderer via the following standard output variables that all user-defined shaders derived from the `Surface` interface must declare as class members:

```
output float3 color; // total reflected energy
output float3 opacity; // outgoing opacity
```

The total amount of energy reflected from the surface should be returned in `color`, and `opacity` should specify the outgoing opacity.

```
// Example surface shader derived from the standard Surface
// interface.
struct myLambert : Shader {
    float3 diffuseColor;
    output float3 color; // Standard color output
    output float3 opacity; // Standard opacity output
    procedure evaluate(
        fragment in, const Light lights[], const int num ) const {
        differentialArea da;
        da.filterSize = in.filterSizeCamera;
        da.point = in.pointCamera;
        da.normal = dot( in.normal, in.to.basis ); // camera space
        float3 N = unit( da.normal );
        float3 I = unit( -in.incidentCamera );
        float3 Kd = 0.0;
        for ( int i = 0; i < num; i += 1 ) {
            const intensity e = lights[i].illuminate( da );
            float3 L = unit( -e.direction );
            float dotNL = dot( N, L );
```

```

        Kd += e.diffuse * clamp( dotNL );
    }
    color = Kd * diffuseColor;
    opacity = 1.0;
}

```

A user-defined surface shader may also declare arbitrary output variables as class members. If an appropriate display is specified in the RXF, arbitrary output variables may be written to image files. See the [Displays](#) section of this document for more information.

Volume

Volume shaders compute the attenuation or modulating effects of a light beam passing through a volumetric primitive.



Note. Volume shaders are not yet documented.

Displacement

```

interface Displacement {
    interval evaluate( box2 coordinate, interval time ) const;
}

```

Displacement shaders compute a surface relief offset that modifies the shape of a geometric surface along its surface normal. If $f(x): \mathbf{R}^2 \rightarrow \mathbf{R}$ is a real continuous function, $X \subseteq \mathbf{R}^2$ is a two-dimensional interval box and

$$f(X) := \left[\min_{x \in X} f(x), \max_{x \in X} f(x) \right],$$

the displacement shader must compute an interval Y such that

$$f(X) \subseteq Y.$$

The interval box X is passed to the displacement shader as the [coordinate](#) parameter of the [evaluate](#) method.

Motion blur of a displacement shader can be achieved if the displacement function f is instead defined $f(x): \mathbf{R}^3 \rightarrow \mathbf{R}$ and the interval box $X \subseteq \mathbf{R}^3$ is three-dimensional. In this case the third dimension of X is the [time](#) parameter.

```

// Example surface shader derived from the standard Surface
// interface. This displacement shader uses the "uvChooser1"
// member variable as the input box instead of the default
// "coordinate" argument passed in to the evaluate() method. In
// order to work properly, ensure that "uvChooser1" is a channel
// in the RXF.
struct displacementShader1 : Displacement {
    box2 uvChooser1;
    range2D displacementMap;
    interval evaluate( box2 coordinate, interval time ) {
        return displacementMap.evaluate( uvChooser1 );
    }
}

```

Shader Initialization

RXF provides several mechanisms to pass user-defined data into shading language programs. See the [Attributes](#) and [Geometric Primitives](#) chapters later in this document for more detailed information on these topics.

Construction

RXF provides commands to construct concrete instances of user-defined Meridian Shading Language classes derived from one of the standard shader interfaces.

During construction, all members of the user-defined class are recursively exploded into basic Meridian Shading Language data types. For each basic type of the exploded class definition, a corresponding literal value is parsed out of the RXF data stream. Every class member of a user-defined shader is initialized in this manner by default values specified in the RXF.

Each default value in the RXF data stream appears in the same order the corresponding class members are declared in the Meridian Shading Language source code. Arrays, packed arrays and nested classes are recursively exploded in a depth-first traversal. Default values for exploded elements of arrays and packed arrays are initialized in array index order.

Class members declared in the Meridian Shading Language source code with an [output](#) qualifier are not part of the exploded class definition, so they are skipped and not initialized.

For example, a simple user-defined surface shader may look like this:

```
// Simple user-defined surface shader.
// Derives from the standard Surface interface.
struct simpleSurface : Surface {
    float3 emissionColor // Input emissionColor
    float3 transparency; // Input transparency
    output float3 color; // Standard color output
    output float3 opacity; // Standard opacity output
    procedure evaluate(
        fragment in, const Lights lights[], const int num ) const {
        color = emissionColor;
        opacity = 1.0 - transparency;
    }
}
```

The shader uses the inputs [emissionColor](#) and [transparency](#) to compute the standard [color](#) and [opacity](#) output variables. In RXF, a concrete instance of [simpleSurface](#) may be constructed with the [Surface](#) command:

```
# Example RXF.
#
# Construct an instance of "simpleSurface" and provide
# initialization data
Surface "simpleSurface" 1 0 0 0 0
```

In this example, the exploded class members `emissionColor` and `transparency` are respectively initialized to red and black.

Dynamic Constants and Variables

All class members of a user-defined Meridian Shading Language class are initialized to default values at construction. However, the class members can also be updated at render time with new values.

RXF provides commands to specify dynamic types, more specifically *dynamic constants* and *dynamic variables*. Dynamic constants are class members that may be updated once for each component of a geometric primitive, and dynamic variables are class members that may be updated once for each shading sample.

Components are typically logical subsets of the geometry specification and are defined by the geometric primitive type. `Particles`, for example, define each particle as a component, so a user-defined value can be uniquely attached to each individual particle as a dynamic constant.

On the other hand, some geometric primitives allow user-defined values to be interpolated over the domain of the geometric primitive and calculated dynamically for each shading sample. `Patch`, for example, can interpolate a set of user-defined control-points over the domain of the surface as a dynamic variable.

Meridian automatically updates dynamic class members at the appropriate time. For example, all dynamic constants are updated once for each component of a geometric primitive, and all dynamic variables are updated once for each shading sample.

Table 2 is a summary of dynamic types. RXF commands may specify a type in the left column, and the middle column identifies if the type is a dynamic constant or a dynamic variable. In the Meridian Shading Language source code, the actual declaration of the class member is expected to be of the type indicated in the right column.

RXF Type	Dynamic Type	MSL Type
float	Variable	interval
float1	Variable	box1
float2	Variable	box2
float3	Variable	box3
const boolean	Constant	boolean
const boolean1	Constant	boolean1
const boolean2	Constant	boolean2
const boolean3	Constant	boolean3
const int	Constant	int
const int1	Constant	int1
const int2	Constant	int2

const int3	Constant	int3
const float	Constant	float
const float1	Constant	float1
const float2	Constant	float2
const float3	Constant	float3

Table 2. Each RXF type in the left column specifies if the class member of a user-defined shader is a dynamic constant or a dynamic variable. In the Meridian Shading Language source code, the actual declaration of the class member is expected to be of the type indicated in the right column.

Meridian uses interval arithmetic to calculate all dynamic variables. This is the reason for the `interval` and `box` types that appear in the right column of Table 2 for the dynamic variables. In other words, any class member that is a dynamic variable must be declared in the Meridian Shading Language source code as an interval extension of the corresponding floating-point type specified in the RXF, as indicated in Table 2. This is so Meridian can pass the calculated interval range enclosure of the dynamic variable to the user-defined shader.

For example, a simple user-defined surface shader may look like this:

```
// Simple user-defined surface shader.
// Derives from the standard Surface interface.
struct uvSurface : Surface {
    box2 uvCoord;           // Dynamic variable
    output float3 color;    // Standard color output
    output float3 opacity;  // Standard output output
    procedure evaluate(
        fragment in, const Lights lights[], const int num ) const {
        float2 m = mid( uvCoord ); // Midpoint of box
        color = float3( m[0], m[1], 0.0 );
        opacity = 1.0;
    }
}
```

The shader uses `uvCoord` as a dynamic variable input and simply assigns the u and v coordinates to the red and green channel of the output color. In RXF, a concrete instance of `uvSurface` may be constructed with the `Surface` command:

```
# Example RXF.
#
# Construct an instance of "uvSurface" and provide
# initialization data
Surface "uvSurface"
0 0 0 0 # Initialize default values of uvCoord to all zeros
SurfaceParameter "float2" "uvCoord" "uv" # Specify dynamic variable
```

In this example, the exploded class member `uvCoord` is initialized to a default value of all zeros. The `SurfaceParameter` command then declares `uvCoord` as a dynamic variable (see the `Attributes` section of this document for more information about the `SurfaceParameter` command). According to Table 2 the actual declaration of the `uvCoord` class member in the Meridian Shading Language source code is a `box2` and receives the interval range enclosure of the dynamic variable.

Native Interfaces

Native interfaces are classes implemented by the run-time environment. Meridian Shading Language does not allow users to derive their own classes from a native interface or to create a concrete instance of a native interface. Only the run-time environment can implement a native interface or create a concrete instance of a native interface.

RXF manages the run-time environment and provides a mechanism to create concrete instances of native interfaces. RXF associates each concrete instance of a native interface with a user-supplied handle name. In Meridian Shading Language, any user-defined class member declared as a native interface type can be initialized in the RXF with a reference to some concrete instance of a native interface via this handle.

For example, suppose in the Meridian Shading Language a user writes a class called `myShader` that declares a member called `texture` as a native interface of the type `sampler2D`:

```
struct mySurface : Surface {
    sampler2D texture; // sampler2D is a native interface type
    output float3 color; // standard shader output
    output float3 opacity; // standard shader output
    procedure evaluate(
        fragment in, const Light lights[], const int num ) const {
        float4 tex = texture.evaluate( float2( in.u, in.v ), 0 );
        color = float3( tex[0], tex[1], tex[2] );
        opacity = 1.0;
    }
}
```

In RXF, a concrete instance of a `sampler2D` can be created by the run-time environment and associated with a user-supplied handle name by using the `Texture2D` command:

```
# Example RXF.
#
# Create a concrete instance of a sampler2D from a texture file
# and give it the handle name "mySurface_texture"
Texture2D "mySurface_texture" "//Klendathu/dog.txr"
```

Later in the RXF, the `texture` member of the `mySurface` class can then be initialized with a reference to this concrete instance of the native interface via the user-supplied handle name:

```
# Pass the handle associated with the concrete instance of
# the sampler2D into the user-defined surface shader "mySurface"
Surface "mySurface" "mySurface_texture"
```

Now the run-time environment initializes the `texture` member of the `mySurface` class with a valid reference to the actual texture map data contained in the file `//Klendathu/dog.txr` that was specified earlier in the `Texture2D` command.

For more detailed information about how to initialize the native interface class members of user-defined shaders, see the [Attributes](#) chapter later in this document.

Samplers

```
native sampler1D {
    float4 evaluate( float1 s, float1 ds ) const;
}

native sampler2D {
    float4 evaluate( float2 s, float2 ds ) const;
}

native sampler3D {
    float4 evaluate( float3 s, float3 ds ) const;
}
```

RXF defines three Meridian Shading Language native interfaces

`sampler1D` `sampler2D` `sampler3D`

for samplers. Shaders may use these sampler interfaces to access texture data and perform texture lookup. For any given input texture coordinate `s` the `evaluate` method filters the texture data over a region of radius `ds` and returns the filtered RGBA result as a `float4` value.

Ranges

```
native range1D {
    interval evaluateEntire() const;
    interval evaluate( box1 coordinate ) const;
}

native range2D {
    interval evaluateEntire() const;
    interval evaluate( box2 coordinate ) const;
}
```

RXF defines two Meridian Shading Language native interfaces

`range1D` `range2D`

for ranges. Shaders may use these range interfaces to access displacement map data and perform displacement map lookup. The `evaluate` method returns an interval range containing the minimum and maximum values of the displacement map over the input box specified by `coordinate`. The `evaluateEntire` method returns an interval range containing the minimum and maximum values of the entire displacement map.

Lexical Structure

This chapter describes the lexical structure of RXF.

RXF data is a stream of ASCII characters translated into a sequence of tokens by the following steps:

- The stream of ASCII characters is translated into a stream of line terminators and input characters
- The stream of line terminators and input characters is translated into a sequence of input elements: white space, comments and tokens
- The sequence of input elements is translated into a sequence of tokens by discarding the white space and comments

Tokens are identifiers and literals.

Line Terminators

RXF data is divided into lines. Line terminators are defined to support the conventions of commonly used systems.

LineTerminator:

- the ASCII LF character
- the ASCII CR character
- the ASCII CR character followed by the ASCII LF character

InputCharacter:

- any ASCII character but not CR or LF

Lines are terminated by the ASCII character CR, or LF, or CR LF.

Input Elements and Tokens

Input characters and line terminators are translated into a sequence of input elements.

InputElement:

- WhiteSpace*
- Comment*
- Token*

Input elements that are not white space or comments are tokens. Tokens are the terminal symbols of the logical structure of RXF data.

White Space

White space is defined as the ASCII space, horizontal tab, vertical tab and form feed characters, as well as line terminators.

```
WhiteSpace:
  the ASCII HT character
  the ASCII VT character
  the ASCII FF character
  the ASCII space character
LineTerminator
```

Comments

The ASCII number sign # delimits the start of a comment. All text from the ASCII character # to the end of the line is ignored.

```
Comment:
  # CharactersInLine

CharactersInLine:
  InputCharacter
  CharactersInLine InputCharacter
```

Tokens

Input elements that are not white space or comments are tokens. Tokens are the terminal symbols of the logical structure of RXF data.

```
Token:
  Identifier
  Literal
```

Identifiers

An identifier is a sequence of letters or digits, the first of which must be a letter.

```
Identifier:
  Letter
  Identifier LetterOrDigit

Letter: one of
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
Digit: one of
  0 1 2 3 4 5 6 7 8 9
```

```
LetterOrDigit:
  Letter
  Digit
```

Blocks and commands that comprise the logical structure of RXF data are reserved identifiers. A list of all the blocks and commands is given in subsequent chapters.

Literals

Literals are lexical elements that specify values for RXF data.

```
Literal:
  BooleanLiteral
  IntegerLiteral
  FloatingPointLiteral
  StringLiteral
```

Boolean Literals

A Boolean literal is a value that may be true or false.

```
BooleanLiteral: one of
  true false
```

Integer Literals

An integer literal is a signed decimal number.

```
IntegerLiteral:
  Sign Digits
  Digits

Sign: one of
  + -

Digits:
  Digit
  Digits Digit
```

The following are examples of integer literals:

```
0           -17          +1398          152
```

Floating-Point Literals

A floating-point literal is a decimal number that may have a fraction, an exponent or both.

```

FloatingPointLiteral:
  Sign FloatingPointPart
  FloatingPointPart

FloatingPointPart:
  Digits FractionOrExponentPart
  . Digits ExponentPartopt

FractionOrExponentPart:
  . DigitsOrNaNopt
  ExponentPartopt

DigitsOrNaN:
  Digitsopt ExponentPartopt
  NaN

NaN:
  # Letters

Letters:
  Letter
  Letters Letter

ExponentIndicator: one of
  e E

ExponentPart:
  ExponentIndicator IntegerLiteral

```

The following are examples of floating-point literals:

0.	0.003	-.0025	+12.7
0.128E+05	+ .33e-6	-17e13	+12.e-123
+0.#INF	-0.#INF	0.#IND	1.#QNAN

Floating-point literals with an ASCII number sign # are valid tokens. However the text after the # is ignored and the value of such floating-point tokens are implementation defined.

String Literals

Strings consist of zero or more string characters enclosed in double quotes.

```

StringLiteral:
  " StringCharactersopt "

```

StringCharacters:

StringCharacter

StringCharacters StringCharacter

StringCharacter:

InputCharacter but not " or \

EscapeSequence

EscapeSequence:

\ *EscapeChar*

EscapeChar: one of

*t n v f r " *

Data Types

Data types are logical elements of the RXF data. Blocks and commands depend on data types.

Basic Types

```
BasicType:  
  Boolean  
  Int  
  Float  
  String  
  
Boolean:  
  BooleanLiteral  
  
Int:  
  IntegerLiteral  
  
Float: one of  
  IntegerLiteral FloatingPointLiteral  
  
String:  
  StringLiteral
```

Basic types are matched to their literal counterparts. A *Boolean* is a Boolean literal. An *Int* is an integer literal, and so on. The exception is a *Float*, which may be a floating-point or integer literal.

Compound Types

```
CompoundType:  
  Point  
  ControlPoint  
  Color  
  Matrix  
  Quaternion  
  Rect
```

Compound types are comprised of basic types.

Point

```
Point:  
    PointX PointY PointZ  
  
PointX:  
    Float  
  
PointY:  
    Float  
  
PointZ:  
    Float
```

Point is a three dimensional coordinate.

ControlPoint

```
ControlPoint:  
    Point ControlWeightopt  
  
ControlWeight:  
    w PointW  
  
PointW:  
    Float
```

ControlPoint is a three dimensional coordinate with a weight. *PointW* should be a value greater than zero. If *ControlWeight* is not specified, *PointW* has an implied value of one (unity).

ControlPoint is not a homogenous coordinate: *PointX*, *PointY* and *PointZ* are not multiplied by *PointW*.

Color

```
Color:  
    ColorRed ColorGreen ColorBlue  
  
ColorRed:  
    Float  
  
ColorGreen:  
    Float  
  
ColorBlue:  
    Float
```

Color is a color with red, green and blue components.

Matrix

```
Matrix:  
array of MatrixRow[ 3 ]  
  
MatrixRow:  
array of Float[ 4 ]
```

Matrix is a 4x4 square matrix. The last row of elements is always [0 0 0 1] and is not stored in the RXF data.

Quaternion

```
Quaternion:  
QuaternionVector QuaternionScalar  
  
QuaternionVector:  
QuaternionX QuaternionY QuaternionZ  
  
QuaternionX:  
Float  
  
QuaternionY:  
Float  
  
QuaternionZ:  
Float  
  
QuaternionScalar:  
Float
```

Quaternion is a quadruple of *Floats*. *QuaternionVector* is the imaginary part and *QuaternionScalar* is the real part.

Rect

```
Rect:  
RectX RectY RectWidth RectHeight  
  
RectX:  
Int  
  
RectY:  
Int  
  
RectWidth:  
Int  
  
RectHeight:  
Int
```

Rect is a rectangle with dimensions *RectWidth* and *RectHeight*. The origin of the rectangle is at coordinates *RectX* and *RectY*.

Blocks

Blocks and commands are logical elements of the RXF data. Blocks define a hierarchical tree structure of nested elements and commands are leaf elements in the tree.

Program is a block, and the logical goal of RXF data is the translation of a sequence of tokens into *Programs*.

```

Goal:
    Programsopt EndOfData

Programs:
    Program
    Programs Program

EndOfData:
    .
  
```

The period after the last program delimits the end of RXF data. Any remaining characters or data are implementation defined.

Program

```

Program:
    Program SourceFilesopt Begin End

SourceFiles:
    SourceFile
    SourceFiles SourceFile

SourceFile:
    Source SourceName

SourceName:
    String

Begin:
    Begin Framesopt End
  
```

```

Frames:
    FrameOrFinish
    Frames FrameOrFinish

FrameOrFinish:
    Frame
    Finish

```

Program is a primary block of RXF data.

User-defined shading routines written in Meridian Shading Language are specified as *SourceFiles*. *SourceName* is the name of a source file containing Meridian Shading Language code. All source files in *Program* are compiled at the start of *Begin*, and the shading language symbols are then available to all of the blocks nested inside the program.

The *Begin* block of a *Program* may contain *Frames*.

FrameOrFinish is a logical element of an animated sequence. *Frame* represents a frame of animation, and *Finish* is a synchronization command that may be used to synchronize concurrent processing of frames.

```

# Example RXF program.
#
Program
Source "foo.txt"
Source "bar.txt"
Begin # All source files are compiled here
# Shading language symbols are available in this block
End # End of Begin
End # End of Program
.

```

Frame

```

Frame:
    Frame FrameNumber Rastersopt End

FrameNumber:
    Int

Rasters:
    RasterOrFinish
    Rasters RasterOrFinish

RasterOrFinish:
    Raster
    Finish

```

Frame is a logical element of an animated sequence.

FrameNumber identifies the frame. It is for informational purposes only and may have any implementation-defined value. In practice, *FrameNumber* is often a frame or sequence number.

A frame block may contain *Rasters*.

RasterOrFinish is a logical element of a frame. *Raster* represents an actual rendered image such as a shadow or beauty pass, and *Finish* is a synchronization command that may be used to synchronize concurrent processing of rasters.

```
# Example RXF program with a frame.
#
Program
Source "foo.txt"
Source "bar.txt"
Begin # All source files are compiled here
Frame 1
# Content for frame 1 goes here
End # End of Frame 1
End # End of Begin
End # End of Program
.
```

Raster

```
Raster:
    Raster RasterOptions Displaysopt Projectionopt End

RasterOptions:
    RasterRect TileExponent PixelExponent

RasterRect:
    Rect

TileExponent:
    Int

PixelExponent:
    Int
```

Raster is a rectangular array of pixels comprising a rendered image.

RasterOptions specify the raster dimensions, the number of pixels per tile and the number of subpixels per pixel. A raster block may have *Display* commands followed by a *Projection*.

A raster is partitioned into a lattice of tiles, and pixels in the raster are partitioned into subpixels. Tiles in a raster are always a square power of two, as is the partition of each pixel into subpixels. Integer pixel coordinate (0,0) is the center of the left-most and bottom-most pixel in the raster: pixels to the right have increasing column index and pixels above have increasing row index.

RasterRect specifies the width and height of the raster in pixels. The origin coordinates of *RasterRect* are not used by Meridian but may be passed to a display driver and saved in a final rendered image.

TileExponent and *PixelExponent* give the dimensions of tiles and pixels as a power of two. Tile dimensions are measured in pixels and pixel dimensions are measured in subpixels. Tiles along the top or right edge of a raster may contain unused pixels if the raster dimensions are not evenly divisible by the tile dimensions: any unused pixels are not part of the raster.

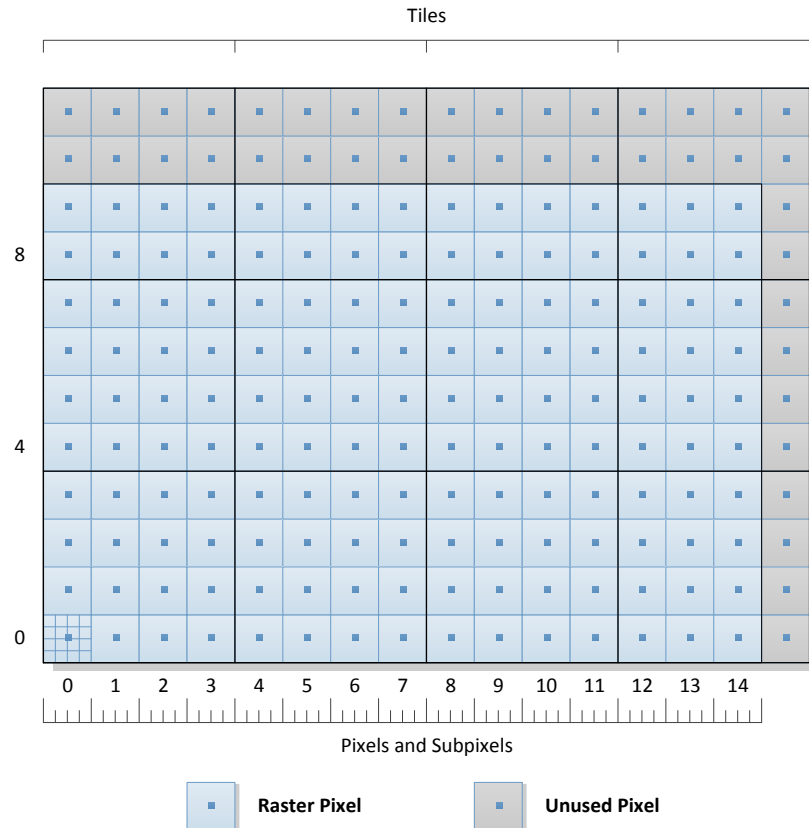


Figure 5. Raster is a rectangular array of pixels. The raster is also partitioned into a lattice of tiles, and pixels in the raster are partitioned into subpixels. Tiles along the top and right edges may contain unused pixels if the raster dimensions are not evenly divisible by the tile dimensions.

```
# Example RXF raster block (see Figure 5).
#
Raster 0 0 15 10 2 2
Color "color.tif" "tiff/rgba/8" 1 0 1 0 1 0.005882
Depth "depth.tif" "tiff/rgb/32" "min"
Alpha "alpha.tif" "tiff/rgb/8"
Perspective -1.33 1.33 -1 1 2 2.33 # Projection block
Lattice # Lattice block
# Tiles can go here
End # End of Lattice
End # End of Perspective
End # End of Raster
```

NDC to Pixel Transformation

Each pixel in a raster represents an area. Integer pixel coordinates specify centers of pixels, but continuous pixel coordinates can specify any point in the area of a pixel. If w and h are the width and height of a raster in pixels, NDC coordinates are mapped to continuous pixel coordinates within the raster by the formula:

$$x_{pixel} = \frac{w}{2} \cdot x_{ndc} + \frac{w-1}{2} \quad y_{pixel} = \frac{h}{2} \cdot y_{ndc} + \frac{h-1}{2}$$

In this way, a raster defines a one-to-one mapping between the NDC window and the area represented by all of the pixels in the rectangular array of pixels comprising the raster.



Note. If the raster dimensions are not evenly divisible by the tile dimensions, any unused pixels in tiles along the top and right edges of the raster do not actually belong to the raster: area represented by the unused pixels are points outside the NDC-space window. These points are clipped and discarded.

Projection

Projection:

ProjectionType ShutterTime Attributes_{opt} Lattice_{opt} End

ProjectionType:

Ortho

Perspective

Stereo

ShutterTime:

ShutterOpen ShutterClose

ShutterOpen:

Float

ShutterClose:

Float

Projection specifies a camera projection for a *Raster*. A projection block may contain *Attributes* followed by a *Lattice*.

The purpose of a camera projection is to define a transformation that maps a subset of camera space to the NDC-space window. *ProjectionType* specifies the actual mathematical projection.



Note. Meridian requires that model or local space is always transformed directly into projection space by the model block. Meridian only applies the

portion of the mathematical projection specified by *ProjectionType* that transforms points from projection space to points in NDC space. For more information about the model block, see the [Model](#) subsection of this document later in this chapter

ShutterTime is an interval of time that specifies a projection exposure.

ShutterOpen and *ShutterClose* specify the times at which the shutter opens and closes. *ShutterOpen* should be less-or-equal to *ShutterClose*.



Note. When processing a scene with motion blur, only the geometric transformations and deformations that are a subset of the interval specified by *ShutterTime* will be rendered.

Ortho

```

Ortho:
  Ortho OrthoWindow

OrthoWindow:
  OrthoLeft OrthoRight OrthoBottom OrthoTop

OrthoLeft:
  Float

OrthoRight:
  Float

OrthoBottom:
  Float

OrthoTop:
  Float

```

Ortho defines an orthographic projection for a *Raster*.

OrthoWindow specifies a transformation from camera space to projection space. If l , r , b , and t are the values of *OrthoLeft*, *OrthoRight*, *OrthoBottom* and *OrthoTop*, and if

$$A = \frac{2}{r-l} \quad B = \frac{2}{t-b} \quad C = \frac{r+l}{r-l} \quad D = \frac{t+b}{t-b}$$

then camera coordinates \vec{p} are transformed into projection coordinates \vec{q} by the formula:

$$\vec{q} = \begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \vec{p} + \begin{bmatrix} C \\ D \\ 0 \end{bmatrix}$$

Projection coordinates x , y and z are then mapped to NDC space by the formula:

$$x_{ndc} = x \quad y_{ndc} = y \quad z_{ndc} = -\frac{1}{z}$$

The subset of projection space $0 \leq z$ is clipped and discarded.

Perspective

```

Perspective:
    Perspective PerspectiveWindow

PerspectiveWindow:
    PerspectiveLeft PerspectiveRight PerspectiveBottom PerspectiveTop

PerspectiveLeft:
    Float

PerspectiveRight:
    Float

PerspectiveBottom:
    Float

PerspectiveTop:
    Float

```

Perspective defines a perspective projection for a *Raster*.

PerspectiveWindow specifies a transformation from camera space to projection space. If l , r , b , and t are the values of *PerspectiveLeft*, *PerspectiveRight*, *PerspectiveBottom* and *PerspectiveTop*, and if

$$A = \frac{2}{r-l} \quad B = \frac{2}{t-b} \quad C = \frac{r+l}{r-l} \quad D = \frac{t+b}{t-b}$$

then camera coordinates \vec{p} are transformed into projection coordinates \vec{q} by the formula:

$$\vec{q} = \begin{bmatrix} A & 0 & C \\ 0 & B & D \\ 0 & 0 & 1 \end{bmatrix} \cdot \vec{p}$$

Projection coordinates x , y and z are then mapped to NDC space by the formula:

$$x_{ndc} = -\frac{x}{z} \quad y_{ndc} = -\frac{y}{z} \quad z_{ndc} = -\frac{1}{z}$$

The subset of projection space $0 \leq z$ is clipped and discarded.

Stereo

Stereo:
Stereo *StereoAngle*

StereoAngle:
Float

Stereo defines a stereographic projection for a *Raster*.

The transformation from camera space to projection space is identity.

If x , y and z are projection coordinates and

$$\rho^2 = x^2 + y^2 + z^2 \quad x' = \frac{x}{\rho} \quad y' = \frac{y}{\rho} \quad z' = \frac{z}{\rho}$$

then projection coordinates are mapped to NDC space by the formula:

$$\varphi \cdot x_{ndc} = \frac{x'}{1 - z'} \quad \varphi \cdot y_{ndc} = \frac{y'}{1 - z'} \quad z_{ndc} = \frac{1}{\rho}$$

The mapping is undefined for all $z = \rho$, and φ is a scaling factor described below.

StereoAngle defines a cone in projection space with axis on the negative z -axis and apex at the origin. *StereoAngle* is the angle from the negative z -axis to the side of the cone in degrees. For the angle

$$\theta = \frac{\pi}{180} \cdot (90 - \textit{StereoAngle})$$

measured in radians, the scaling factor

$$\varphi = \frac{\cos(\theta)}{1 + \sin(\theta)}$$

ensures all projection-space points inside the cone are mapped to points inside the NDC-space window. If *StereoAngle* is exactly 90 degrees, every point in projection space with a negative z coordinate is mapped to a point somewhere inside the NDC unit circle. Larger and smaller values of *StereoAngle* are also allowed and map the appropriate subsets of projection space to NDC space.

Lattice

Lattice:
Lattice *Tiles_{opt}* End

Tiles:
Tile
Tiles Tile

Lattice is a rectangular array of non-overlapping tiles in a raster. A lattice block may contain *Tiles*.

Tiles in a lattice are always a square power of two. All tiles are the same size, and tile dimensions are measured in pixels. The number of tiles and the dimensions of tiles are specified by the raster.

Each tile in a lattice has *x* and *y* integer coordinates. Lattice coordinate (0,0) is the left-most and bottom-most tile in the lattice: tiles to the right have increasing column index and tiles above have increasing row index.

Each tile in a lattice has a unique *identification number* composed by interleaving the bits of the *x* and *y* lattice coordinates into a two-dimensional Morton code. Figure 6 is an example. Lattice coordinates and identification numbers do not specify the order tiles must appear within a lattice block. Tiles may appear in any order.

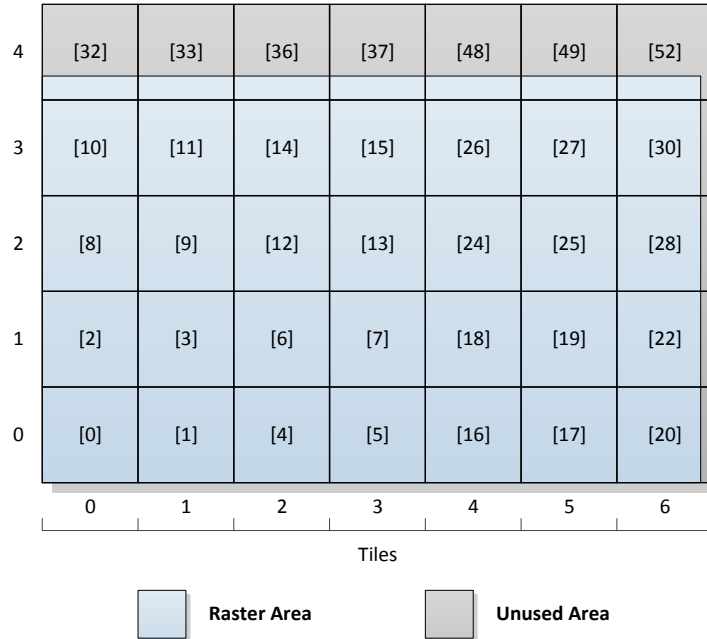


Figure 6. Lattice of tiles within a raster. Each tile has a unique identification number shown in square brackets. The identification number is composed by interleaving the bits of the *x* and *y* lattice coordinates into a two-dimensional Morton code.

Tile

Tile:

Tile TileNumber Models_{opt} End

TileNumber:

Int

Models:

Model

Models Model

Tile is an element of a lattice in a raster. A tile block may contain *Models*.

A tile is always a square power of two. Tile dimensions are measured in pixels and specified by the raster.

TileNumber is a unique identification number that specifies the spatial location of the tile within a lattice by interleaving the bits of the x and y lattice coordinates of the tile into a two-dimensional Morton code.



Note. Meridian decodes lattice coordinates of a tile from *TileNumber* and depends on this information to render images correctly.

Model

Model:

ModelType AttributesOrShadingGroups_{opt} End

ModelType:

Transform

Motion

AttributeOrShadingGroup:

Attribute

ShadingGroup

AttributesOrShadingGroups:

AttributeOrShadingGroup

AttributesOrShadingGroups AttributeOrShadingGroup

Model delimits a model or local coordinate system for geometric objects in the scene. A model block may contain *AttributesOrShadingGroups*.

Geometry is specified relative to the model coordinate system and transformed into projection space. *ModelType* specifies the actual mathematical transformation from model space to projection space.

Transform

Transform:

Transform Matrix

Transform defines a fixed coordinate system.

Matrix specifies a transformation from model space to projection space.

```
# Example RXF transform block.
#
Transform
2 0 0 -3      # MatrixRow[0]
0 1 0 -17    # MatrixRow[1]
0 0 4 -2     # MatrixRow[2]
# AttributesOrShadingGroups go here
End # End of Transform
```

Motion

```
Motion:
    Motion MotionPath

MotionPath:
    MotionControl MotionKnots MotionControlPoints

MotionControl:
    MotionNum MotionOrder

MotionNum:
    Int

MotionOrder:
    Int

MotionKnots:
    array of Float[ MotionNum + MotionOrder ]

MotionControlPoints:
    MotionPosition MotionRotation MotionScale MotionShear

MotionPosition:
    array of Point[ MotionNum ]

MotionRotation:
    array of Quaternion[ MotionNum ]

MotionScale:
    array of Point[ MotionNum ]

MotionShear:
    array of Point[ MotionNum ]
```

Motion defines an animated coordinate system. *MotionPath* specifies the animated coordinate system as an affine transformation parameterized in time as a b-spline.

MotionControl specifies a number of control points and an order for a b-spline. The number of control points and order are provided by *MotionNum* and *MotionOrder* respectively. The relations

$$MotionNum \geq MotionOrder \geq 2$$

must all be true.

MotionKnots is an array of *MotionNum* + *MotionOrder* knots. *MotionPath* is parameterized as a function of *u* for all

$$u \in [MotionKnots_{MotionOrder-1}, MotionKnots_{MotionNum}]$$

that are also elements of the shutter time interval specified by the current *Projection*.

MotionControlPoints specifies control point data. *MotionPosition*, *MotionRotation*, *MotionScale* and *MotionShear* are each arrays of *MotionNum* elements that define, in respective order, b-spline curves parameterized in *u* by *MotionKnots*:

$$\vec{p}(u) = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} \quad \vec{q}(u) = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \quad \vec{s}(u) = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} \quad \vec{h}(u) = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}$$

Curve $\vec{q}(u)$ parameterizes a rotation matrix

$$R(u) = \frac{2}{q_0^2 + q_1^2 + q_2^2 + q_3^2} \begin{bmatrix} q_0^2 + q_3^2 & q_0q_1 - q_2q_3 & q_2q_0 + q_3q_1 \\ q_0q_1 + q_2q_3 & q_1^2 + q_3^2 & q_1q_2 - q_3q_0 \\ q_2q_0 - q_3q_1 & q_1q_2 + q_3q_0 & q_2^2 + q_3^2 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Since $R(u)$ performs normalization, quaternions defining the curve $\vec{q}(u)$ do not need to be of unit length.

Curves $\vec{s}(u)$ and $\vec{h}(u)$ parameterize matrices

$$S(u) = \begin{bmatrix} s_0 & 0 & 0 \\ 0 & s_1 & 0 \\ 0 & 0 & s_2 \end{bmatrix} \quad H(u) = \begin{bmatrix} 0 & h_0 & h_2 \\ h_0 & 0 & h_1 \\ h_2 & h_1 & 0 \end{bmatrix}$$

The affine transformation of the animated coordinate system specified by *MotionPath* is then defined

$$\vec{y} = A(u) \cdot \vec{x} + \vec{p}(u)$$

where $A(u)$ is the linear transformation matrix

$$A(u) = R(u) \cdot (S(u) + H(u))$$

and model coordinates \vec{x} are transformed into projection coordinates \vec{y} .

```
# Example RXF motion block.
#
Motion 3 3      # MotionNum and MotionOrder
0 0 0 1 1 1    # MotionNum + MotionOrder = 6 knots
1 3 4          # MotionPosition[0]
2 6 8          # MotionPosition[1]
```

```

3 2 4      # MotionPosition[2]
0.3 0.1 0.6 1.2 # MotionQuaternion[0]
0.4 0.3 0.5 1.5 # MotionQuaternion[1]
0.5 0.2 0.3 1.7 # MotionQuaternion[2]
1 1 1      # MotionScale[0]
1 2 3      # MotionScale[1]
2 1 2      # MotionScale[2]
0 0 0      # MotionShear[0]
0 0 0      # MotionShear[1]
1 0 0      # MotionShear[2]
# AttributesOrShadingGroups go here
End # End of Motion

```

ShadingGroup

ShadingGroup:

```
ShadingGroup Attributesopt Linkeropt End
```

ShadingGroup delimits a logical subset of a model sharing common shading attributes.

A shading group block may contain *Attributes* followed by a *Linker*.

Linker

Linker:

```
Linker GeometricPrimitivesopt End
```

Linker binds attributes of a shading group to actual geometric primitives.

A linker block may contain *GeometricPrimitives*.



Note. *Linker* is the only block that may contain *GeometricPrimitives*.

Displays

```

Displays:
  Display
  Displays Display

Display:
  DisplayColor
  DisplayDepthOrAlpha

DisplayColor:
  Color DisplayOutput DisplayDriver DisplayColorOptions

DisplayDepthOrAlpha:
  Depth DisplayOutput DisplayDriver DisplayDepthOptions
  Alpha DisplayOutput DisplayDriver

DisplayOutput:
  String

DisplayDriver:
  String

```

Display is a logical association between a raster, an *output device* and a *display driver*. An output device is usually a file on a hard drive. A display driver specifies how raster data is sent to an output device.

Internal to the Meridian software, a raster may have color, depth and alpha buffers. A raster always has one depth buffer and one alpha buffer. They are created automatically by Meridian. Color buffers must be created explicitly. Each color buffer in a raster has its own set of RGB pixels, and any number of color buffers may be created.

DisplayColor creates a color buffer for the raster and associates the color buffer pixels with *DisplayOutput* and *DisplayDriver*.

DisplayDepthOrAlpha does not create a depth or alpha buffer but associates an existing depth or alpha buffer with *DisplayOutput* and *DisplayDriver*.

DisplayOutput specifies the name of the output. This is typically the name of a file to be created on a hard drive.

DisplayDriver is the name of a supported display driver.

```
# Example RXF.
#
Color "color.tif" "tiff/rgba/8" 1 0 1 0 1 0.005882
Color "glow.tif" "tiff/rgb/8" "glow" 1 0 1 0 1 0.005882
Depth "depth.tif" "tiff/rgb/32" "min"
Alpha "alpha.tif" "tiff/rgb/8"
```

Display Drivers

A *display driver* defines a *data type* and a set of *channels* that are used to create and store pixel data in the *DisplayOutput* of a *Display*.

Table 3 is a list of supported display drivers. The data type and channels describe the type of pixel output produced by each display driver listed in the table. Byte and short are 8-bit and 16-bit unsigned integers. Float and double are 32-bit and 64-bit IEEE 754 floating-point values. Beta is an undocumented format used in Sunfish shadow maps.

Driver Name	Data Type	Channels	Description
"tiff/a/8"	byte	A	Adobe TIFF
"tiff/a/16"	short	A	Adobe TIFF
"tiff/a/32"	float	A	Adobe TIFF
"tiff/a/64"	double	A	Adobe TIFF
"tiff/rgb/8"	byte	RGB	Adobe TIFF
"tiff/rgb/16"	short	RGB	Adobe TIFF
"tiff/rgb/32"	float	RGB	Adobe TIFF
"tiff/rgb/64"	double	RGB	Adobe TIFF
"tiff/rgba/8"	byte	RGBA	Adobe TIFF
"tiff/rgba/16"	short	RGBA	Adobe TIFF
"tiff/rgba/32"	float	RGBA	Adobe TIFF
"tiff/rgba/64"	double	RGBA	Adobe TIFF
"shadow/vsm/a"	beta	A	Sunfish VSM
"shadow/vsm/rgb"	beta	RGB	Sunfish VSM

Table 3. Supported display drivers and their properties.

Dynamic Range and Quantization

Meridian calculates all color, depth and alpha buffers in floating-point precision. This ensures high-fidelity representation of image data during the entire rendering process. Floating-point pixels are always provided as input to a display driver.

All display drivers that produce integer pixel output expect floating-point input values to be in the range [0,1]. The display driver may clip any values outside this range, or even of a subset of this range. All of the floating-point input values which have not been clipped are then quantized by the formula

$$value_{out} = \text{round}(M \cdot value_{in})$$

where the constant M is the maximum value in the dynamic range of the data type of the display driver. For the byte and short data types M is 255 and 65535 respectively.

Display drivers that produce floating-point output usually perform no clipping or quantization and simply store the floating-point input values “as is,” even if the input values are not in the range $[0,1]$.

Color Options

```

DisplayColorOptions:
    DisplayColorSymbolopt DisplayColorTransformations
DisplayColorSymbol:
    String
DisplayColorTransformations:
    DisplayExposure DisplayNormalization DisplayDitherAmplitude
DisplayExposure:
    DisplayGain DisplayOffset DisplayGamma
DisplayNormalization:
    DisplayZero DisplayOne
DisplayGain:
    Float
DisplayOffset:
    Float
DisplayGamma:
    Float
DisplayZero:
    Float
DisplayOne:
    Float
DisplayDitherAmplitude:
    Float

```

DisplayColorOptions are display options that specify a sequence of standard transformations applied to floating-point color pixels *before* the pixels are provided as input to a display driver. The source of the color pixels may also be specified.

DisplayColorSymbol specifies the name of an output variable of a user-defined shader derived from the [Surface](#) interface. At render-time, the value of the

output variable is used to draw pixels in the color buffer. If *DisplayColorSymbol* is not specified, Meridian automatically uses the default name “color” for the standard color output variable of the shader.

DisplayExposure, *DisplayNormalization* and *DisplayDitherAmplitude* specify exposure, normalization and dithering transformations respectively. If $value_{in}$ is a value of a color pixel in a color buffer, $value_{out}$ is the corresponding value actually provided to a display driver:

$$\begin{aligned} v_1 &= DisplayGain \cdot value_{in} + DisplayOffset \\ v_2 &= sign(v_1) \cdot |v_1|^{1/DisplayGamma} \\ v_3 &= (DisplayOne - DisplayZero) \cdot v_2 + DisplayZero \\ value_{out} &= DisplayDitherAmplitude \cdot random + v_3 \end{aligned}$$

The transformations are applied component-wise to each of the red, green and blue channels of a color pixel. The value *random* is a uniformly distributed random number selected from the interval [-1,1].

After the transformations are applied, the floating-point color pixels are provided as input to a display driver.



Note. The *DisplayZero* and *DisplayOne* values are passed to the display driver, so the display driver may choose to clip the floating-point input values to some subset of the interval [*DisplayZero*,*DisplayOne*].

Depth Options

DisplayDepthOptions:

DisplayDepthFilter

DisplayDepthFilter:

String

DisplayDepthOptions are display options that specify standard transformations applied to floating-point depth pixels *before* the pixels are provided as input to a display driver.

DriverDepthFilter specifies a depth filter transformation. Table 4 is a list of supported depth filters.

Depth Filter	Description
“min”	Minimum depth value
“max”	Maximum depth value
“moment1”	Mean depth value
“moment2”	Second moment of depth values
“beta”	Beta distribution

Table 4. Supported depth filters.

Attributes

```
Attributes:  
  Attribute  
  Attributes Attribute  
  
Attribute:  
  ShadingAttribute  
  RenderingAttribute
```

Meridian maintains an *attribute state*. The attribute state is a set of *Attributes*.

Attribute state is preserved by RXF block structure: entering a block saves a copy of the attribute state and leaving a block restores the attribute state to the saved copy. If an attribute is changed inside a block, the changes are lost when the block scope ends.

ShadingAttribute specifies an attribute related in some way to Meridian Shading Language. *RenderingAttribute* specifies an attribute that controls some aspect of the Meridian rendering engine.

Shading Attributes

```
ShadingAttribute:  
  NativeInterface  
  Shader  
  SurfaceParameter  
  Channel
```

Attribute state contains a set of *shading attributes*. All shading attributes are related in some way to Meridian Shading Language.

Many of the RXF shading attributes require the name of a shading language symbol. Meridian Shading Language source files are specified by [Source](#) commands in an RXF program and then compiled at the start of the begin block. All of the compiled shading language symbols are then available in the scope of the blocks nested inside the program. If the name of a shading language symbol specified by an attribute does not exist in this scope, a syntax error may be generated.

Native Interface

```

NativeInterface:
    NativeInterfaceType NativeHandleName NativeFile

NativeInterfaceType:
    Displacement1D
    Displacement2D
    Texture1D
    Texture2D
    Texture3D
    Shadow

NativeHandleName:
    String

NativeFile:
    String

```

Native interfaces are Meridian Shading Language classes implemented by the run-time environment.

Meridian Shading Language does not allow users to derive their own classes from a native interface or to create a concrete instance of a native interface. Only the run-time environment can implement a native interface or create a concrete instance of a native interface.

RXF manages the run-time environment and provides a mechanism to create concrete instances of native interfaces. RXF associates each concrete instance of a native interface with a user-supplied handle name. In Meridian Shading Language, any user-defined class member declared as a native interface type can be initialized in the RXF with a reference to some concrete instance of a native interface via this handle.

NativeInterface causes the run-time environment to create a new concrete instance of a native interface of a certain type. The concrete instance is created from a disk file specified by *NativeFile*. The run-time environment associates the concrete instance of the native interface with a handle name specified by *NativeHandleName*.

Shader attributes may use the handle name specified by *NativeHandleName* as a *ShaderDefaultValue* in *ShaderData*. This initializes any Meridian Shading Language class member of an appropriate type with a reference to the concrete instance associated with *NativeHandleName*.

Table 5 is a summary of the supported values for *NativeInterfaceType*.

NativeInterfaceType specifies the expected contents of *NativeFile* as well as the required Meridian Shading Language type that can be successfully initialized by

the run-time environment with a reference to a concrete instance of a native interface without generating a run-time error.

RXF <i>NativeInterfaceType</i>	Shading Language Type	Typical File Extension	Expected File Contents
Displacement1D	range1D	.txd	1D displacement file
Displacement2D	range2D	.txd	2D displacement file
Texture1D	sampler1D	.txr	1D texture file
Texture2D	sampler2D	.txr	2D texture file
Texture3D	sampler3D	.txr	3D texture file
Shadow	sampler3D	.vsm	Shadow map file

Table 5. Supported values for *NativeInterfaceType*.

```
# Example RXF.
#
# Assume the following user-defined shader has been specified
# for the RXF program and compiled in the begin block
#
# // Meridian Shading Language source code for
# // user-defined class "userSurface"
#
# struct userSurface : Surface {
#     sampler2D texture; // sampler class member
#     ... // the rest of the class is elided
# }
#
# Create a concrete instance of a sampler2D from a texture file
# and give it a handle name "dog_texture":
Texture2D "dog_texture" "//k|endathu/dog.txr"
#
# Create an instance of the userSurface shader and initialize
# the "texture" class member with a reference to the concrete
# sampler:
Surface "userSurface" "dog_texture"
```

Shader

```
Shader:
    ShaderType ShaderName ShaderDataopt

ShaderType:
    Displacement
    Light
    Surface
    Volume

ShaderData:
    ShaderDefaultValue
    ShaderData ShaderDefaultValue

ShaderDefaultValue:
    BasicType
```

Shader specifies a new shader of type *ShaderType*.

ShaderName specifies the name of *Shader*. *ShaderName* must be the name of an available shading language symbol that is a user-defined class derived from the *ShaderType* interface.

The attribute state contains a set of shaders consisting of one displacement shader, one surface shader, one volume shader and a list of light shaders. The initial attribute state for each shader type is empty.

If *ShaderType* is a displacement, surface or volume shader, *Shader* replaces the current shader of the same type in the attribute state with the shader specified by *ShaderName*. If *ShaderType* is a light shader, *Shader* adds the shader specified by *ShaderName* to the list of light shaders in the attribute state.

For the user-defined Meridian Shading Language class specified by *ShaderName*, all members of the class are recursively exploded into basic Meridian Shading Language data types. *ShaderData* specifies a *ShaderDefaultValue* for each basic type of the exploded class definition in the following manner.

Each *ShaderDefaultValue* of *ShaderData* in the RXF data stream appears in the same order the corresponding class members are declared in the Meridian Shading Language source code. Arrays, packed arrays and nested classes are recursively exploded in a depth-first traversal. Exploded elements of arrays and packed arrays appear in the *ShaderData* in array index order.

The *BasicType* of each *ShaderDefaultValue* in *ShaderData* must match the actual type specified in the Meridian Shading Language source code, and there must be one such *ShaderDefaultValue* for each item of the exploded class definition of *ShaderName*.

Class members declared in the Meridian Shading Language source code with an `output` qualifier are not part of the exploded class definition, so they are skipped and do not appear in *ShaderData*.

```
# Example RXF.
#
# Assume the following user-defined shader has been specified
# for the RXF program and compiled in the begin block
#
# // Meridian Shading Language source code for
# // user-defined class "userSurface"
#
# struct userSurface : Surface {
#   float3 ambientColor;    // input ambientColor
#   float3 diffuseColor;   // input diffuseColor
#   output float3 color;    // standard shader output
#   output float3 opacity; // standard shader output
#   ... // the rest of the class is elided
# }
#
# Create an instance of the userSurface shader and initialize
# all of the non-output class members so that ambientColor
# is red and diffuseColor is green:
Surface "userSurface" 1 0 0 0 1 0
```

Surface Parameter

```

SurfaceParameter:
    SurfaceParameter SurfaceVariable SurfaceAssignment

SurfaceVariable:
    SurfaceVariableType SurfaceVariableName

SurfaceVariableType:
    String

SurfaceVariableName:
    String

SurfaceVariableAssignment:
    String

```

SurfaceParameter declares a dynamic variable that is assigned the value of a standard surface variable known to the rendering engine.

SurfaceVariable specifies the declaration of the dynamic variable.

SurfaceVariableType is the declared data type of the dynamic variable and *SurfaceVariableName* is the variable name.

SurfaceVariableAssignment specifies a standard surface variable known to the rendering engine that is assigned to the dynamic variable specified by *SurfaceVariable*. Table 6 is a tabulation of the supported values for *SurfaceVariableAssignment*.

SurfaceVariable Type	SurfaceVariable Assignment	Description
float3	"P"	Surface position in local space
float	"u"	Surface u-coordinate
float	"v"	Surface v-coordinate
float2	"uv"	Surface uv-coordinate
float	"time"	Exposure time

Table 6. Standard surface variables known by the rendering engine that may be assigned to the dynamic variable specified by *SurfaceVariableName*.

The attribute state maintains a set of surface variable declarations.

SurfaceParameter adds the declaration specified by *SurfaceVariable* to the set.

The initial attribute state contains an empty set of declarations.

```

# Example RXF.
#
# Assume the following user-defined shader has been specified
# for the RXF program and compiled in the begin block
#
# // Meridian Shading Language source code for
# // user-defined class "userSurface"
#
# struct userSurface : Surface {
#     box3 referencePoint; // input reference point
#     box2 texCoord;      // input texture coordinate

```

```
# ... // the rest of the class is elided
# }
#
# Create an instance of the userSurface shader and initialize
# the class members with default values:
Surface "userSurface" 0 0 0 0 0 0 0 0 0 0
#
# Create the surface parameter attributes. This causes
# the "referencePoint" and "texCoord" class members of the
# userSurface shader to receive interpolated surface data
# on a per-sample basis:
SurfaceParameter "float3" "referencePoint" "P"
SurfaceParameter "float2" "texCoord" "uv"
```

Channel

```
Channel:
    Channel ChannelVariable

ChannelVariable:
    ChannelType ChannelName

ChannelType:
    String

ChannelName:
    String
```

Channel declares a dynamic variable attached to a geometric primitive.

ChannelVariable specifies the declaration. *ChannelType* is the declared data type of the dynamic variable and *ChannelName* is the variable name.

ChannelType must be a string that specifies one of the values in the left column of Table 2. This also identifies *ChannelVariable* as a dynamic constant or a dynamic variable.

The attribute state maintains a list of channel declarations. *Channel* adds the declaration specified by *ChannelVariable* to the end of the list, so all of the declarations in the list are stored in the order they appear in the RXF data. The initial attribute state contains an empty list of channel declarations.

For each channel declaration in the attribute state, geometric primitives must provide a corresponding channel of data. Channel data syntax is defined by each geometric primitive type. See the documentation for each geometric primitive type for a specification of the channel data syntax.

```
# Example RXF.
#
# Assume the following user-defined shader has been specified
# for the RXF program and compiled in the begin block
#
# // Meridian Shading Language source code for
# // user-defined class "userSurface"
#
```

```

# struct userSurface : Surface {
#     box3 referencePoint; // input reference point
#     box2 texCoord; // input texture coordinate
#     ... // the rest of the class is elided
# }
#
# Create an instance of the userSurface shader and initialize
# the class members with default values:
Surface "userSurface" 0 0 0 0 0 0 0 0 0 0
#
# Create the channel attributes. This causes the "referencePoint"
# and "texCoord" class members of the userSurface shader to
# receive interpolated surface data on a per-sample basis:
Channel "float3" "referencePoint"
Channel "float2" "texCoord"
#
# Note that subsequent geometric primitives must explicitly
# provide channel data in the RXF stream for both
# "referencePoint" and "texCoord"
Quad
0 0 0 # Required point data
1 0 0
0 1 0
1 1 0
0 0 1 # Required normal data
0 0 1
0 0 1
0 0 1
2 1 3 # "float3" "referencePoint" channel data
3 2 7
1 1 2
7 4 3
3 4 # "float2" "texCoord" channel data
6 8
4 5
9 9

```

Rendering Attributes

RenderingAttribute:

Brush
Displace
DoubleSided
Opposite
ShadingRate
Timebase
TransparencyScale

Attribute state contains a set of *rendering attributes*. All rendering attributes control some aspect of the Meridian rendering engine.

Brush

Brush:

Brush BrushColor BrushOpacity

BrushColor:

Color

BrushOpacity:

Color

Brush specify a default color and opacity for geometric primitives. *BrushColor* and *BrushOpacity* specify the respective default values.

Default color and opacity values are used to render geometric primitives only if the surface shader in the attribute state is empty.

The initial attribute state for *BrushColor* and *BrushOpacity* is white.

```
# Example RXF.
#
# Specify a red and opaque brush:
Brush 1 0 0 1 1 1
```

Displace

Displace:

Displace DisplaceValue

DisplaceValue:

Boolean

Displace specifies if displacement shaders actually deform surface geometry or just the surface normal. If *DisplaceValue* is true, displacement shaders deform surface geometry. Otherwise only surface normals are modified, creating a bump-mapping effect.

The initial attribute state for *DisplaceValue* is true.

```
# Example RXF.
#
Displace false # Enable bump-mapping
```

DoubleSided

DoubleSided:

DoubleSided DoubleSidedValue

DoubleSidedValue:

Boolean

DoubleSided specifies if both sides of a geometric primitive are rendered or only the parts facing the camera.

If \vec{N} is the surface normal of a geometric primitive at a point and \vec{I} is the camera incident vector, the geometric primitive faces the camera when

$$-\vec{I} \cdot \vec{N} \geq 0$$

If *DoubleSidedValue* is true, then both sides of a geometric primitive are rendered. Otherwise only the parts of a geometric primitive that are facing the camera are rendered.

The initial attribute state for *DoubleSidedValue* is true.

```
# Example RXF.  
#  
DoubleSided false
```

Opposite

```
Opposite:  
Opposite OppositeValue  
  
OppositeValue:  
Boolean
```

Opposite specifies if the surface orientation of a geometric primitive is reversed at render time by negating the surface normal.

If *OppositeValue* is true, the surface normal is negated. Otherwise the surface normal is not reversed.

The initial attribute state for *OppositeValue* is false.

```
# Example RXF.  
#  
Opposite true
```

ShadingRate

```
ShadingRate:  
ShadingRate ShadingRateValue  
  
ShadingRateValue:  
Float
```

ShadingRate is the primary quality control for Meridian.

ShadingRateValue specifies a multiplier value expressed in pixel area. A value of 1.0 has no effect. Larger or smaller values improve rendering speed or image quality respectively.

The initial attribute state for *ShadingRateValue* is 1.0.

```
# Example RXF.  
#  
ShadingRate 0.8
```

Timebase

```
Timebase:  
Timebase TimebaseValue
```

TimebaseValue:

Float

Timebase is a secondary quality control for Meridian that is used only when rendering geometric primitives with motion blur.

TimebaseValue is a multiplier of *ShadingRateValue*. A value of 1.0 has no effect. Larger or smaller values respectively improve rendering speed or image quality of geometric primitives with motion blur.



Note. When rendering geometric primitives with motion blur, Meridian automatically multiplies *ShadingRateValue* by an area of sixteen pixels. So a *TimebaseValue* of 1.0 is actually equal to an effective *ShadingRateValue* scaled by this amount.

The initial attribute state for *TimebaseValue* is 1.0.

```
# Example RXF.
#
Timebase 1.2
```

TransparencyScale

TransparencyScale:

TransparencyScale *TransparencyScaleValue*

TransparencyScaleValue:

Int

TransparencyScale controls the quality of transparency effects.

Unlike many rendering engines, Meridian uses a unique method for rendering transparency that operates in a single pass and uses a constant memory footprint. This allows highly complex scenes with transparent objects to render efficiently and never run out of memory. A consequence of this method is that only a fixed dynamic range is available for the transparency effect.

TransparencyScaleValue is a scaling value that adjusts the dynamic range of the transparency effect based on the depth-buffer values of geometric primitives in the scene. *TransparencyScaleValue* must be an integer in the interval [0,63].

The initial attribute state for *TransparencyScaleValue* is 32.

```
# Example RXF.
#
TransparencyScale 24
```

Geometric Primitives

```
GeometricPrimitives:  
  GeometricPrimitive  
  GeometricPrimitives GeometricPrimitive  
  
GeometricPrimitive:  
  FreeformSurface  
  Polygon  
  LightweightPrimitive
```

All of the currently supported geometric primitives are parametric surfaces. Parametric surfaces define a mapping from a (u, v) parameter space to three-dimensional model space.

Channels

Attribute state contains a list of channel declarations, and each channel declaration in the list requires that all subsequent geometric primitives must parse a corresponding channel of data out of an RXF data stream.

Channel data syntax is specified by each geometric primitive type. Geometric primitives must parse an appropriate channel of data for each channel declaration in the attribute state. For example, if the attribute state contains three channel declarations, each geometric primitive must parse an appropriate channel of data for each declaration in the attribute state.

Channel declarations in the attribute state list are always stored in the order they were specified in the RXF data. Geometric primitives must parse channels in this order.

A syntax error is generated if the channel data expected by a geometric primitive is not present in the RXF data.

Channel Type

```
ChannelType:  
  Float  
  array of Float[ 1 ]
```

```

array of Float[ 2 ]
array of Float[ 3 ]
Boolean
array of Boolean[1]
array of Boolean[2]
array of Boolean[3]
Int
array of Int[1]
array of Int[2]
array of Int[3]

```

ChannelType is defined lexically as one of the productions in the grammar above, but semantically *ChannelType* is defined by the data type of a corresponding channel declaration in the attribute state.

This means the actual production allowed for *ChannelType* depends on the channel declaration in the attribute state that corresponds to the channel of data actually being parsed by a geometric primitive. For example, if the channel declaration in the attribute state specifies “float2” as the data type, it is a semantic error for *ChannelType* to be any production other than an array of two floats.

Freeform Surfaces

```

FreeformSurface:
  BSpline
  Patch
  TrianglePatch

```

Freeform surfaces are polynomial or rational surfaces that are defined by a set of control points.

BSpline

```

BSpline:
  BSpline BSplineSurface BSplineChannelsopt
BSplineSurface:
  BSplineControl BSplineKnots BSplineControlPoints
BSplineControl:
  BSplineNumU BSplineOrderU BSplineNumV BSplineOrderV
BSplineNumU:
  Int

```

```

BSplineNumV:
    Int

BSplineOrderU:
    Int

BSplineOrderV:
    Int

BSplineKnots:
    BSplineKnotsU BSplineKnotsV

BSplineKnotsU:
    array of Float[ BSplineNumU + BSplineOrderU ]

BSplineKnotsV:
    array of Float[ BSplineNumV + BSplineOrderV ]

BSplineControlPoints:
    array of ControlPoint[ BSplineNumU * BSplineNumV ]

BSplineChannels:
    BSplineChannel
    BSplineChannels BSplineChannel

BSplineChannel:
    BSplineControl BSplineKnots BSplineChannelData

BSplineChannelData:
    array of ChannelType[ BSplineNumU * BSplineNumV ]

```

BSpline is a *BSplineSurface* followed by optional *BSplineChannels*.

BSplineSurface defines a b-spline surface that maps a rectangular subset of a (u, v) parameter space into model space. An order, a number of control points and an array of knots are specified for each of the u and v parameter dimensions, and an array of control points is specified for the entire surface.

BSplineControl specifies a number of control points and an order for each of the u and v parameter dimensions of a b-spline surface. The number of control points and order are provided by *BSplineNumU*, *BSplineOrderU*, *BSplineNumV* and *BSplineOrderV* respectively. The relations

$$\begin{aligned}
 BSplineNumU &\geq BSplineOrderU \geq 2 \\
 BSplineNumV &\geq BSplineOrderV \geq 2
 \end{aligned}$$

must all be true. *BSplineControl* implicitly defines the number of piece-wise segments for each of the u and v parameter dimensions:

$$\begin{aligned}
 segments_u &= (BSplineNumU - BSplineOrderU) + 1 \\
 segments_v &= (BSplineNumV - BSplineOrderV) + 1
 \end{aligned}$$

BSplineKnots specifies an array of knots for each of the u and v parameter dimensions. *BSplineKnotsU* is an array of $BSplineNumU + BSplineOrderU$ knots and *BSplineKnotsV* is an array of $BSplineNumV + BSplineOrderV$ knots.

BSplineSurface is parameterized in u and v as a function of all

$$u \in [BSplineKnotsU_{BSplineOrderU-1}, BSplineKnotsU_{BSplineNumU}]$$

$$v \in [BSplineKnotsV_{BSplineOrderV-1}, BSplineKnotsV_{BSplineNumV}]$$

BSplineControlPoints specifies control point data as an array of *ControlPoint* elements. The size of the array is $BSplineNumU * BSplineNumV$, and elements are specified in row-major order. If any *ControlPoint* element in the array has non-unity weight the surface is rational.

BSplineChannel specifies a new set of *BSplineControl* and *BSplineKnots* for *BSplineChannelData*. *BSplineChannelData* is a b-spline surface that is parameterized by the new *BSplineControl* and *BSplineKnots*. The number of segments defined for *BSplineChannel* must equal the number of segments defined for *BSplineSurface* in both u and v parameter dimensions. *ChannelType* can only specify dynamic variables.

```
# Example RXF.
#
# This example assumes channel attribute:
# Channel "float2" "texCoord"
# is in the attribute state.
BSpline
5 4 # Number of control points and order in u
5 4 # Number of control points and order in v
0 0 0 0.5 1 1 1 1 # Knots in u
0 0 0 0 0.5 1 1 1 # Knots in v
-4 -4 0 -2.6 -4 0 0 -4 0 2.6 -4 0 4 -4 0
-4 -2.6 0 -2.6 -2.6 0 0 -2.6 0 2.6 -2.6 0 4 -2.6 0
-4 0 0 -2.6 0 0 0 0 2.6 0 0 4 0 0
-4 2.6 0 -2.6 2.6 0 0 2.6 0 2.6 2.6 0 4 2.6 0
-4 4 0 -2.6 4 0 0 4 0 2.6 4 0 4 4 0
3 2 # Number of control points and order in u for channel
3 2 # Number of control points and order in v for channel
0 0 0.5 1 1 # Knots in u for channel
0 0 0.5 1 1 # Knots in v for channel
0 0 0.5 0 1 0
0 0.5 0.5 0.5 1 0.5
0 1 0.5 1 1 1
```

Patch

Patch:

Patch PatchSurface PatchChannels_{opt}

PatchSurface:

PatchOrder PatchControlPoints

PatchOrder:

PatchOrderU PatchOrderV

```

PatchOrderU:
    Int

PatchOrderV:
    Int

PatchControlPoints:
    array of ControlPoint[ PatchOrderU * PatchOrderV ]

PatchChannels:
    PatchChannel
    PatchChannels PatchChannel

PatchChannel:
    PatchOrder PatchChannelData

PatchChannelData:
    array of ChannelType[ PatchOrderU * PatchOrderV ]

```

Patch is a *PatchSurface* followed by optional *PatchChannels*.

PatchSurface defines a Bezier surface. A Bezier surface is parameterized in u and v and maps the set of all

$$u \in [0,1] \quad v \in [0,1]$$

into model space. An order is specified for each of the u and v parameter dimensions, and an array of control points is specified. A Bezier surface always has the same number of control points along a parameter dimension as the order of that dimension.

PatchOrder specifies an order for the u and v parameter dimensions of a Bezier surface, provided by *PatchOrderU* and *PatchOrderV* respectively.

$$\begin{aligned} \text{PatchOrderU} &\geq 2 \\ \text{PatchOrderV} &\geq 2 \end{aligned}$$

must be true.

PatchControlPoints specifies control point data as an array of *ControlPoint* elements. The size of the array is $\text{PatchOrderU} * \text{PatchOrderV}$, and elements are specified in row-major order. If any *ControlPoint* element in the array has non-unity weight the surface is rational.

PatchChannel specifies a new *PatchOrder* and *PatchChannelData*.

PatchChannelData is a Bezier surface that is parameterized by the new *PatchOrder*. *ChannelType* can only specify dynamic variables.

```

# Example RXF.
#
# This example assumes channel attribute:
#   Channel "float2" "texCoord"
# is in the attribute state.

```

```
Patch
5 5 # order in u and v
-4 -4 0 -2.6 -4 0 0 -4 0 2.6 -4 0 4 -4 0
-4 -2.6 0 -2.6 -2.6 0 0 -2.6 0 2.6 -2.6 0 4 -2.6 0
-4 0 0 -2.6 0 0 0 0 2.6 0 0 4 0 0
-4 2.6 0 -2.6 2.6 0 0 2.6 0 2.6 2.6 0 4 2.6 0
-4 4 0 -2.6 4 0 0 4 0 2.6 4 0 4 4 0
2 2 # order in u and v for channel
0 0 1 0
0 1 1 1
```

TrianglePatch

```
TrianglePatch:
    TrianglePatch TrianglePatchSurface TrianglePatchChannelsopt

TrianglePatchSurface:
    TriangleOrder TriangleControlPoints

TriangleOrder:
    Int

TriangleControlPoints:
    array of ControlPoint[ ( TriangleOrder * ( TriangleOrder + 1 ) ) / 2 ]

TrianglePatchChannels:
    TrianglePatchChannel
    TrianglePatchChannels TrianglePatchChannel

TrianglePatchChannel:
    TriangleOrder TriangleChannelData

TriangleChannelData:
    array of ChannelType[ ( TriangleOrder * ( TriangleOrder + 1 ) ) / 2 ]
```

TrianglePatch is a *TrianglePatchSurface* followed by optional *TrianglePatchChannels*.

TrianglePatchSurface defines a triangular Bezier surface. A triangular Bezier surface is parameterized in u and v and maps the set of all

$$u \in [0,1] \quad v \in [0,1] \quad 1 - u - v \geq 0$$

into model space. An order and an array of control points specify the triangular Bezier surface.

TriangleOrder specifies an order for a triangular Bezier surface, and

$$\textit{TriangleOrder} \geq 2$$

must be true.

TriangleControlPoints specifies control point data as an array of *ControlPoint* elements. The size of the array is $(\textit{TriangleOrder} * (\textit{TriangleOrder} + 1)) / 2$, and

the elements are specified in order of nested triangles. If any *ControlPoint* element in the array has non-unity weight the surface is rational.

TrianglePatchChannel specifies a new *TriangleOrder* for *TriangleChannelData*. *TriangleChannelData* is a triangular Bezier surface that is parameterized by the new *TriangleOrder*. *ChannelType* can only specify dynamic variables.

```
# Example RFX.
#
# This example assumes channel attribute:
# Channel "float2" "texCoord"
# is in the attribute state.
#
# Control points are specified in order of nested triangles:
#
#     9
#     5 8
#     2 4 7
#     0 1 3 6
#
TrianglePatch
4 # Order
0 0 0
1 0 0 0 1 0
2 0 0 1 1 0 0 2 0
3 0 0 2 1 0 1 2 0 0 3 0
2 # Order for channel
0 0
1 0 0 1
```

Polygons

Polygon:

Face

Quad

Polygon primitives are provided mainly for compatibility with polygon-based modeling and animation software packages.

Face

Face:

Face FaceSurface FaceChannels_{opt}

FaceSurface:

FacePoints FaceNormals

FacePoints:

array of *Point*[3]

FaceNormals:

array of *Point*[3]

```

FaceChannels:
  FaceChannel
  FaceChannels FaceChannel

FaceChannel:
  array of ChannelType[ 3 ]

```

Face is a *FaceSurface* followed by optional *FaceChannels*.

FaceSurface defines a triangular Barycentric surface. A triangular Barycentric surface is parameterized in u and v and maps the set of all

$$u \in [0,1] \quad v \in [0,1] \quad 1 - u - v \geq 0$$

into model space.

FacePoints and *FaceNormals* specify an array of three points and an array of three surface normals respectively. Elements \vec{e}_0 , \vec{e}_1 and \vec{e}_2 in each array are interpolated over the parameter domain by the formula:

$$\vec{S}(u, v) = \vec{e}_0 + u \cdot (\vec{e}_1 - \vec{e}_0) + v \cdot (\vec{e}_2 - \vec{e}_0)$$

FaceChannel specifies optional channel data that, if present, is similarly interpolated. *ChannelType* can only specify dynamic variables.

```

# Example RXF.
#
# This example assumes channel attribute:
# Channel "float2" "texCoord"
# is in the attribute state.
Face
0 0 0 1 0 0 0 1 0 # Points
0 0.1 1 0.1 0 1 0.1 0.1 1 # Normals
0 0 1 0 0 1 # Channel

```

Quad

```

Quad:
  Quad QuadSurface QuadChannelsopt

QuadSurface:
  QuadPoints QuadNormals

QuadPoints:
  array of Point[ 4 ]

QuadNormals:
  array of Point[ 4 ]

QuadChannels:
  QuadChannel
  QuadChannels QuadChannel

```

```

QuadChannel:
  array of ChannelType[ 4 ]

```

Quad is a *QuadSurface* followed by optional *QuadChannels*.

QuadSurface defines a bilinear surface. A bilinear surface is parameterized in u and v and maps the set of all

$$u \in [0,1] \quad v \in [0,1]$$

into model space.

QuadPoints and *QuadNormals* specify an array of four points and an array of four surface normals respectively. The array elements are specified in row-major order and bilinearly interpolated over the parameter domain.

QuadChannel specifies optional channel data that, if present, is similarly interpolated. *ChannelType* can only specify dynamic variables.

```

# Example RXF.
#
# This example assumes channel attribute:
#   Channel "float2" "texCoord"
# is in the attribute state.
Quad
0 0 0 1 0 0 0 1 0 1 1 0 # Points
0 0.1 1 0.1 0 1 0.1 0.1 1 0 0 1 # Normals
0 0 1 0 0 1 1 1 # Channel

```

Lightweight Primitives

```

LightweightPrimitive:
  Particles
  Ribbon
  Tube

```

Lightweight primitives are parametric surfaces optimized for speed and quantity. Meridian is designed to efficiently render scenes consisting of millions or even billions of these lightweight primitives.

Particles

```

Particles:
  Particles ParticlesSurface ParticlesChannelsopt

ParticlesSurface:
  ParticlesNum ParticlesPoints ParticlesSizes

ParticlesNum:
  Int

ParticlesPoints:
  array of Point[ ParticlesNum ]

```

ParticlesSizes:array of *Float*[*ParticlesNum*]**ParticlesChannels:***ParticleChannel**ParticleChannels ParticleChannel***ParticleChannel:**array of *ChannelType*[*ParticlesNum*]

Particles is a *ParticlesSurface* followed by optional *ParticlesChannels*.

ParticlesSurface is actually a collection of many surfaces: each surface in the collection is a sphere parameterized for all $u, v \in [0,1]$ by the formula

$$\vec{S}(u, v) = radius \cdot \begin{bmatrix} \cos(2\pi u) \cdot \cos(\pi v - \pi/2) \\ \sin(2\pi u) \cdot \cos(\pi v - \pi/2) \\ \sin(\pi v - \pi/2) \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$$

The coordinates x_0, y_0 and z_0 are the origin of the sphere and *radius* is the radius of the sphere.

ParticlesNum specifies the number of spheres in *ParticlesSurface*. *ParticlesPoints* and *ParticlesSizes* specify the origin and radius of each sphere respectively.

ParticlesChannel specifies optional channel data as an array of *ChannelType* elements. The size of the array is *ParticlesNum*, and there is one element in the array for each sphere in *ParticlesSurface*. *ChannelType* can only specify dynamic constants.

```
# Example RXF.
#
# This example assumes channel attribute:
# Channel "float3" "particleColor"
# is in the attribute state.
Particles 3
3 2 1 6 2 7 5 2 1 # Points
1.2 2.4 2.1 # Sizes
1 0 0 0 1 0 0 0 1 # Channel
```

Ribbon

Ribbon:*Ribbon RibbonSurface RibbonChannels_{opt}***RibbonSurface:***RibbonControl RibbonKnots RibbonControlPoints***RibbonControl:***RibbonNum RibbonOrder***RibbonNum:***Int*

```

RibbonOrder:
    Int

RibbonKnots:
    array of Float[ RibbonNum + RibbonOrder ]

RibbonControlPoints:
    RibbonPosition RibbonRotation RibbonScale

RibbonPosition:
    array of Point[ RibbonNum ]

RibbonRotation:
    array of Quaternion[ RibbonNum ]

RibbonScale:
    array of Float[ RibbonNum ]

RibbonChannels:
    RibbonChannel
    RibbonChannels RibbonChannel

RibbonChannel:
    RibbonControl RibbonKnots RibbonChannelData

RibbonChannelData:
    array of ChannelType[ RibbonNum ]

```

Ribbon is a *RibbonSurface* followed by optional *RibbonChannels*.

RibbonSurface defines a ribbon as a swept surface. The $[-1,1]$ interval of an x -axis is swept along a coordinate frame that is specified by a sequence of control points and parameterized as a b-spline.

RibbonControl specifies a number of control points and an order for a b-spline. The number of control points and order are provided by *RibbonNum* and *RibbonOrder* respectively. The relations

$$RibbonNum \geq RibbonOrder \geq 2$$

must all be true. *RibbonControl* implicitly defines the number of piece-wise segments of the b-spline:

$$segments = (RibbonNum - RibbonOrder) + 1$$

RibbonKnots is an array of $RibbonNum + RibbonOrder$ knots. *RibbonSurface* is parameterized in v as a function of all

$$v \in [RibbonKnots_{RibbonOrder-1}, RibbonKnots_{RibbonNum}]$$

RibbonControlPoints specifies control point data. *RibbonPosition*, *RibbonRotation* and *RibbonScale* are each arrays of *RibbonNum* elements that define, in respective order, b-spline curves parameterized in v by *RibbonKnots*:

$$\vec{p}(v) = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} \quad \vec{q}(v) = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \quad \vec{s}(v) = [s_0]$$

Curve $\vec{q}(v)$ parameterizes a vector

$$\vec{r}(v) = \frac{2}{q_0^2 + q_1^2 + q_2^2 + q_3^2} \begin{bmatrix} q_0^2 + q_3^2 \\ q_0q_1 + q_2q_3 \\ q_2q_0 - q_3q_1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Since $\vec{r}(v)$ performs normalization, quaternions defining the curve $\vec{q}(v)$ do not need to be of unit length.

The ribbon is also parameterized in u for all

$$u \in [0,1]$$

and the actual surface is defined

$$\vec{S}(u, v) = \vec{p}(v) + \vec{r}(v) \cdot s_0 \cdot (2u - 1)$$

RibbonChannel specifies a new set of *RibbonControl* and *RibbonKnots* for *RibbonChannelData*. *RibbonChannelData* is interpolated as a b-spline curve that is parameterized by the new *RibbonControl* and *RibbonKnots*. The number of segments defined for *RibbonChannel* must equal the number of segments defined for *RibbonSurface*. *ChannelType* can only specify dynamic variables.

```
# Example RXF.
#
# This example assumes channel attribute:
# Channel "float3" "ribbonColor"
# is in the attribute state.
Ribbon
4 4 # Number and order
0 0 0 1 1 1 1 # Knots
0 0 0 2 7 3 5 6 1 7 2 8 # Points
-1 0 1 -1 -1 0 1 -0.2 -2 2 1 -0.8 -0.3 2 1 -1.5 # Quaternions
2 1 0.8 1 # Scale
2 2 # Number and order of channel
0 0 1 1 # Knots for channel
1 1 0 0.8 0.2 1
```

Tube

Tube:

Tube TubeSurface TubeChannels_{opt}

TubeSurface:

TubeControl TubeKnots TubeControlPoints

```

TubeControl:
    TubeNum TubeOrder

TubeNum:
    Int

TubeOrder:
    Int

TubeKnots:
    array of Float[ TubeNum + TubeOrder ]

TubeControlPoints:
    TubePosition TubeRotation TubeScale

TubePosition:
    array of Point[ TubeNum ]

TubeRotation:
    array of Quaternion[ TubeNum ]

TubeScale:
    array of Float[ TubeNum ][2]

TubeChannels:
    TubeChannel
    TubeChannels TubeChannel

TubeChannel:
    TubeControl TubeKnots TubeChannelData

TubeChannelData:
    array of ChannelType[ TubeNum ]

```

Tube is a *TubeSurface* followed by optional *TubeChannels*.

TubeSurface defines a tube as a swept surface. The unit circle in an *xy*-plane is swept along a coordinate frame that is specified by a sequence of control points and parameterized as a b-spline.

TubeControl specifies a number of control points and an order for a b-spline. The number of control points and order are provided by *TubeNum* and *TubeOrder* respectively. The relations

$$TubeNum \geq TubeOrder \geq 2$$

must all be true. *TubeControl* implicitly defines the number of piece-wise segments of the b-spline:

$$segments = (TubeNum - TubeOrder) + 1$$

TubeKnots is an array of *TubeNum* + *TubeOrder* knots. *TubeSurface* is parameterized in v as a function of all

$$v \in [TubeKnots_{TubeOrder-1}, TubeKnots_{TubeNum}]$$

TubeControlPoints specifies control point data. *TubePosition*, *TubeRotation* and *TubeScale* are each arrays of *TubeNum* elements that define, in respective order, b-spline curves parameterized in v by *TubeKnots*:

$$\vec{p}(v) = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} \quad \vec{q}(v) = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \quad \vec{s}(v) = \begin{bmatrix} s_0 \\ s_1 \end{bmatrix}$$

Curve $\vec{q}(v)$ parameterizes a matrix

$$R(v) = \frac{2}{q_0^2 + q_1^2 + q_2^2 + q_3^2} \begin{bmatrix} q_0^2 + q_3^2 & q_0q_1 - q_2q_3 \\ q_0q_1 + q_2q_3 & q_1^2 + q_3^2 \\ q_2q_0 - q_3q_1 & q_1q_2 + q_3q_0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Since $R(v)$ performs normalization, quaternions defining the curve $\vec{q}(v)$ do not need to be of unit length.

Curve $\vec{s}(v)$ parameterizes a matrix

$$S(v) = \begin{bmatrix} s_0 & 0 \\ 0 & s_1 \end{bmatrix}$$

The tube is also parameterized in u for all

$$u \in [0,1]$$

and the actual surface is defined

$$\vec{S}(u, v) = \vec{p}(v) + R(v) \cdot S(v) \cdot \begin{bmatrix} \cos(2\pi u) \\ \sin(2\pi u) \end{bmatrix}$$

TubeChannel specifies a new set of *TubeControl* and *TubeKnots* for *TubeChannelData*. *TubeChannelData* is interpolated as a b-spline curve that is parameterized by the new *TubeControl* and *TubeKnots*. The number of segments defined for *TubeChannel* must equal the number of segments defined for *TubeSurface*. *ChannelType* can only specify dynamic variables.

```
# Example RXF.
#
# This example assumes channel attribute:
# Channel "float3" "tubeColor"
# is in the attribute state.
Tube
4 4 # Number and order
0 0 0 1 1 1 1 # Knots
0 0 0 2 7 3 5 6 1 7 2 8 # Points
-1 0 1 -1 -1 0 1 -0.2 -2 2 1 -0.8 -0.3 2 1 -1.5 # Quaternions
2 -2 1 -2 0.8 -1.3 1 -1 # Scale
2 2 # Number and order of channel
```

```
0 0 1 1 # knots for channel  
1 1 0 0.8 0.2 1
```

Miscellaneous Commands

This chapter documents a few miscellaneous RXF commands.

Synchronization Commands

SynchronizationCommand:

Finish

Meridian is highly parallel software, and most RXF data is processed concurrently. Synchronization commands provide a mechanism to specify synchronization points within an RXF data stream.

Finish

Finish:

Finish

Finish waits for all of the threads in a multi-processing environment to become idle before parsing the next token. This ensures all concurrent processing of all prior RXF data is completed before continuing.