
MetaSL

Shading Language Tutorial and Reference

Andy Kopra and Lutz Kettner

Excerpt:

Chapter 1: MetaSL — Strategy and scope

Preliminary draft — subject to change.

For latest version, see <http://www.metasl.org>

Copyright Information

Copyright © 1986-2011 mental images GmbH, Berlin, Germany.

All rights reserved.

This document is protected under copyright law. The contents of this document may not be translated, copied or duplicated in any form, in whole or in part, without the express written permission of mental images GmbH.

The information contained in this document is subject to change without notice. mental images GmbH and its employees shall not be responsible for incidental or consequential damages resulting from the use of this material or liable for technical or editorial omissions made herein.

mental images®, mental ray®, mental matter®, mental mill®, mental queue®, mental cloudTM, mental mesh®, RealityServer®, RealityPlayer®, RealityDesigner®, MetaSL®, Metanode®, Phenomenon®, neuray®, iray®, DiCETM, imatter®, Shape-By-Shading®, SPM®, and rendering imagination visibleTM are trademarks or, in some countries, registered trademarks of mental images GmbH, Berlin, Germany.

Other product names mentioned in this document may be trademarks or registered trademarks of their respective companies and are hereby acknowledged.

Chapter 1

MetaSL — Strategy and scope

MetaSL is a special-purpose programming language for the rendering of three-dimensional scenes in computer graphics. Special-purpose languages are often called *domain-specific* languages — they have been designed with the expressive purposes of a particular discipline in mind.

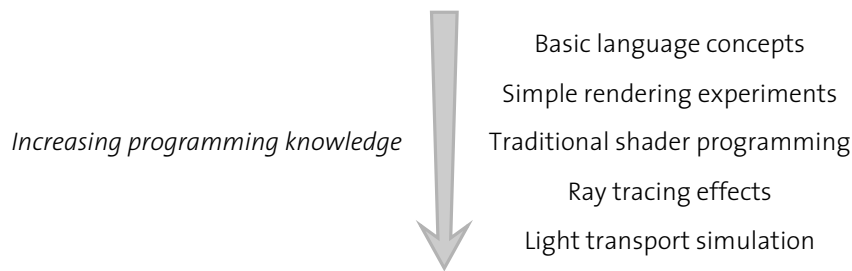
But all programming languages are designed for a purpose, though some may be broader in intent than others. The names of two of the oldest programming languages still in widespread use, Fortran and Lisp, are based on what programs in those languages represent. Fortran — “formula translation” — and Lisp — “list processing” — implemented the most productive means of expression for the users of the language. Those users are also “domain-specific” — communities of researchers and engineers employing a common programming language that in turn influenced the language’s future development.

All the members of the MetaSL community employ software rendering systems to create imagery, but their purposes can vary widely. Entertainment content production, architectural and industrial design, scientific and historical visualization — each field employs different methods and has different requirements for the qualities that produce a “successful” picture. MetaSL’s goal is to supply a general language framework to these disparate disciplines that also simplifies the implementation of the requirements unique to each.

This is, of course, an ambitious goal. To address it, MetaSL provides a range of possibilities for what MetaSL programs can do, how MetaSL programs are organized, and how MetaSL programs are installed and become part of a production environment. Presenting these three facets of MetaSL — language, organization, and implementation — is the purpose of this book.

1.1 Language

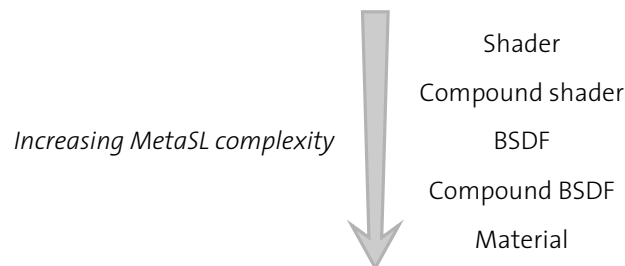
MetaSL simplifies a variety of common software concepts required by a rendering system. The basic MetaSL software unit, the *shader*, is a plugin module defining inputs and outputs to a process that the shader implements. The language provides the programmer with common-sense arithmetic operations on higher-level data, like colors and geometric elements. Rendering state and functions operating on that state can be used to implement in simple terms a variety of traditional rendering algorithms. Such simplifications are typical among programming languages used in renderer plugins. However, MetaSL also allows the programmer to simulate the physics of light transport using functional definitions for surface properties, the *bidirectional scattering distribution function* (BSDF).



This book begins with a description of MetaSL as a programming language separate from issues of rendering. Beginning programmers should read Chapter 2, “Elements of the MetaSL language,” for an overview of the basics of MetaSL. More advanced programmers should review the “MetaSL Reference” in the Appendices to see how software concepts with which they may be familiar are implemented in MetaSL.

1.2 Organization

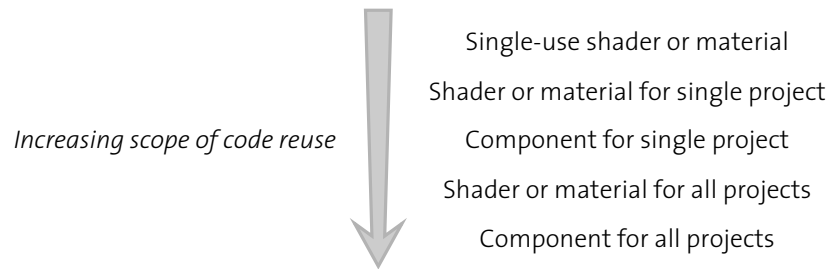
An individual MetaSL shader can specify the appearance or shape of an object, the effect of the environment on that appearance, or manipulate the simulated camera of the rendering system. MetaSL also uniquely includes as part of the language hierarchical combinations of shader units. These combined forms — *compound shaders*, *compound BSDFs*, and *materials* — can also be used in other combinations, allowing hierarchical structures of arbitrary complexity. The MetaSL programmer can think of shader programming as the construction of a set of small, well tested modules — the creation of a vocabulary of reusable, elemental units.



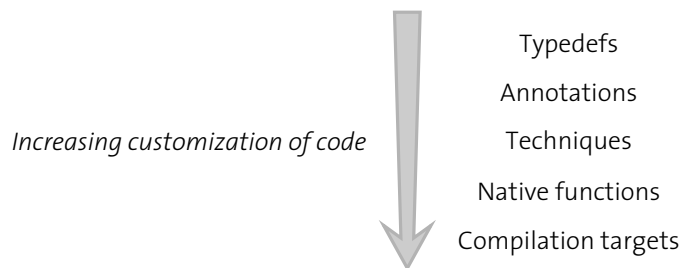
Combined forms are described in Part 2, “Organization,” and build upon the development of individual shaders in the previous chapters. Creating these structures follows in a straightforward way from shader programming; their syntax is essentially a subset of shader syntax.

1.3 Implementation

A MetaSL shader can be designed for a very limited role in a rendering project. Some shaders may only be used once, some even written in an emergency to solve an unforeseen problem. But a shader may also be designed to be used throughout a project, or across many projects, based on production planning that takes current and future projects into account. Creating cross-project shaders may be simplified by writing shaders intended to be used only as components in combined forms. Developing a library of reusable components can be an important part of MetaSL development in a larger visualization department or production facility.



The combined structures in Part 2, “Organization,” are useful tools for the management of MetaSL code across projects. Part 3, “Implementation,” presents the additional mechanisms available for implementers of MetaSL-based systems for effective shader implementation on different hardware and software platforms.



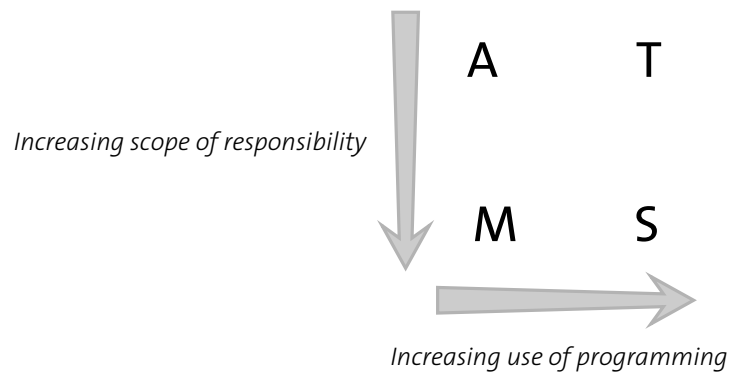
A MetaSL programmer can make the intended use of datatypes clearer to other programmers through *typedefs*, customized names for pre-existing MetaSL types. *Annotations* provide additional information about shader parameters that can be presented to the user in a graphical interface. The application user will think of a MetaSL shader in terms of these parameters and the effect the shader has on the rendering process. MetaSL code represents this effect as an abstraction of the implementation details of shader programming. Typically, the shader programmer writes a single shader; the abstraction MetaSL provides allows it to be used in a wide variety of hardware and software environments through *compilation* of the shader to a target language or hardware-specific executable.

However, the intended use of such a shader may vary, for example, from real-time display for scene design to off-line rendering of final frames of an animation. To provide a logically similar interpretation of the shader in these different scenarios, the MetaSL programmer creates a set of shaders that are *parametrically equivalent* — all shaders in the set have the same parameters. Each shader in this *equivalence set* implements that shader’s intended effect in the most appropriate way for a given scenario. To make use of scenario-specific code in other languages, the MetaSL programmer can include *native functions* within shaders. Taken together, the set of shaders for a single scenario is called a *technique*.

Technique definitions allow the MetaSL programmer to preserve the shader’s role as a “high level” description even as different scenarios require different implementations. Thus, an application user may select the “real-time display” technique for scene construction, while the production of the final imagery uses the “high-quality rendering” technique.

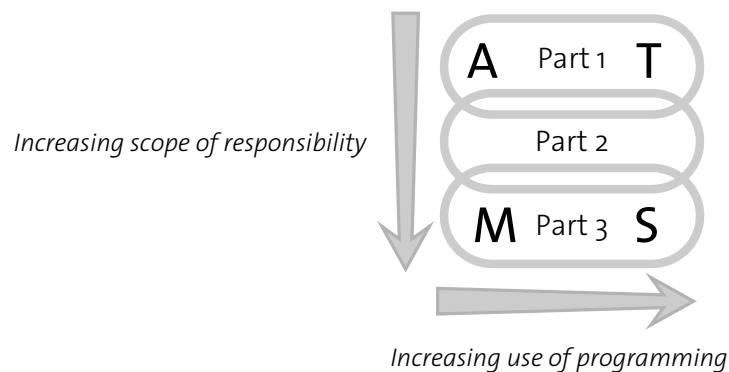
1.4 Audience

The ranges of programming knowledge and the scope of MetaSL’s use described above can characterize some typical roles in a production facility. For example, application users, technical assistants, project managers, and system programmers vary in their programming skill and the scope of their responsibilities.



- A** Application user working on a single unit of a project, typically using software through a graphical user interface.
- T** Technical assistant writing software required by a single unit of a project.
- M** Manager responsible for the software developed by the programming staff for use across all projects in the facility.
- S** Systems programmer designing and implementing the software used across all projects in the facility.

The three major parts of this book are designed to address this wide variety of skills and job requirements.



Part 1, “Language,” presents the background needed by application users and technical assistants who want to learn shader programming in MetaSL. A discussion of basic programming concepts as they are implemented in MetaSL is followed by the presentation of a series of shaders of increasing visual complexity.

Part 2, “Organization,” describes the creation of more complex MetaSL structures from the basic shader components described in Part 1. These structures are important both for project programmers creating specific effects as well as for system programmers and facility managers who want to develop and control their rendering assets effectively.

Part 3, “Implementation,” will provide system programmers and managers with the information they need to customize MetaSL programs for specific hardware and software platforms, preserving the beneficial abstraction that MetaSL provides while making the most effective use of their production resources.