SML3d: 3D Graphics for Standard ML

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Abstract

The SML3d system is a collection of libraries designed to support real-time 3D graphics programming in Standard ML (SML). This paper gives an overview of the system and briefly highlights some of the more interesting aspects of its design and implementation.

1. Introduction

SML3d is a collection of libraries designed to support real-time 3D graphics programming in Standard ML (SML). At its core, SML3d provides an SML interface to the industry standard OpenGL API [14], which is supported on workstations, PCs, and most mobile devices. SML3d also provides support for the common linear algebra types and operations found in computer graphics applications, as well as libraries for images, meshes, audio, and other common components of real-time 3D graphics applications. The SML3d libraries are implemented on top of the MLton implementation of SML [10] and include a type safe interface to OpenGL, various utilities to support graphics applications, and SML interfaces to related libraries. This paper gives a high-level overview of the design and implementation of the SML3d libraries.

2. OpenGL

OpenGL is an industry standard library for programming direct-style 3D graphics that was originally introduced by SGI in 1991 [4]. Since that time, computer graphics hardware has undergone a tremendous evolution to the point where a modern graphics card provides super-computer performance, but only costs a few hundred dollars.[1] The OpenGL API has evolved to track the changes in graphics hardware, with 16 revisions of the OpenGL specification released since 1991. Prior to Version 3.0 (released in August 2008), the standard programming model for OpenGL was the so-called *fixed pipeline*. This model relied on a complicated *state machine* to define how primitives should be rendered on the screen.[2] When rendering a scene, one would specify properties, such as the way that lighting would be computed, using a series of API calls. Once the state machine was properly configured, primitives would be streamed through the pipeline using other API calls to submit vertex information to the GL. This model was inflexible, difficult to evolve, and did not match the hardware of modern GPUs, which had evolved into fully programmable devices.

As of OpenGL Version 3.2 (released in August 2009), the old state-machine model has be deprecated and a new programmable pipeline model (called the *Core Profile*) has been made standard. This model is organized around two main mechanisms: memory buffers of various kinds that hold the inputs (and outputs) of rendering, and various programmable pipeline stages for processing vertices, primitives, and fragments. To maximize performance, applications must minimize the overhead of transferring data to the GPU and shift as much computation to the GPU as possible.

Both of these trends make it much more practical to use high-level functional languages to program real-time graphics and to take advantage of features such as higher-order functions and polymorphic type systems. The use of data buffers to specify geometry, *etc.* plays to the strength of functional languages for efficiently dealing with aggregate data using iteration combinators. Since much of the computational load is borne by the GPU, we can exploit the ability of functional languages to support embedded DSLs and staged computation to both raise the level of the programming model and deliver high performance.

SML3d was originally implemented on top of OpenGL Version 2.1, but once the Version 3.2 core API became widely available, we decided to remove support for the old fixed pipeline and do a complete rewrite of the library to support modern graphics programming techniques. This paper describes this new version of the SML3d libraries.

3. Design goals

There are a number of different approaches that one can take when designing a high-level-language API for a C API, such as OpenGL. One approach is to match the low-level C API as closely as possible. This is the approach followed by the Haskell OpenGLRaw package [13]. At the other end of the spectrum, once could define a high-level library, such as Apple's SceneKit [2], which provides a scene-graph API that largely hides the OpenGL API.

For our taste, we prefer an API that *fits* the SML style of programming, provides low overhead, and does not preclude full use of the OpenGL API. We believe that we have achieved these goals in the design of SML3d. It supports the full OpenGL Core Profile API, but does so in a way that respects the strongly-typed nature of SML. Using MLton's efficient foreign-interface (FI) mechanisms minimizes overhead. For example, the OpenGL C API defines a single type `glenum` to cover several thousand symbolic constants. In practice, however, these constants are logically organized into a few hundred distinct conceptual types (note that some constants appear in multiple types). In SML3d, we introduce distinct abstract types (represented by the native `glenum` type) for each conceptual type in the OpenGL API. This approach leads to better static error checking and better documented interfaces.

Over time, we plan to develop higher-level libraries that build on top of the OpenGL Core API. Examples include support for triangle meshes, 2D rendering, text rendering, *etc.*. We have also experimented with embedded DSLs that provide high-level abstractions for dynamically generated shader code [7] in earlier versions of SML3d and we plan to further explore examples of the DSL approach to graphics in the future.

[1] For comparison, the top-of-the-line graphics hardware of the early 1990's cost four orders of magnitude more and provided significantly less performance.

[2] In graphics terminology, a *primitive* is a piece of planar geometry, such as a point, line segment, or triangle. Primitives are defined by vertices and are rasterized to produce *fragments*.

4. The SML3d libraries

SML3d consists of over 45,000 lines of SML code organized as a collection of libraries using MLton's MLB mechanism. While space does not permit a detailed description of these, we briefly describe the most important components and survey the others.

Raw data This library provides an abstraction layer over various MLton-specific FI features as well as providing a home for types that are common across the other SML3d libraries. It defines structures for dealing with the native integer and floating-point types used in OpenGL (and other APIs), and utilities for interfacing with C code.

A key feature of the Raw Data library are *data buffers*, which are arrays of 1, 2, 3, or 4-element, integer or floating-point tuples. They can effectively represent any of the basic OpenGL buffer types, used to hold image data, vertex attributes, and other types of graphical data. Data buffers are allocated on the C heap and can be explicitly deallocated. As a backup, we use MLton's finalization mechanism to ensure that the buffer storage is recovered in case the buffers become garbage.

SML3d The SML3d library is the core of the system. It provides support for linear algebra types and operations, geometric types, and, most importantly, it provides an interface to the OpenGL API. The interface is structured as a single module (`GL`) with substructures for each of the main API components (vertex specification, drawing, blending, textures, *etc.*). Our current focus is on the Version 3.2 Core API, but we plan to support the more recent versions of the API too. The the desired implementation is specified as a build option (see Section 7).

Image I/O Image data is used to specify textures as well as other forms of graphical data, such as height fields and noise. SML3d provides a library for loading image data from a number of standard image file formats. These currently include the various Netpbm formats [11], PNG files [12], TGA files [17], and TIFF files [16].[3] Both input and output operations are provided for each format. There is also support for generating synthetic images, such as gradients. The Image I/O library is independent of OpenGL (and the SML3d library), so that it can be used in image processing applications.

GLFW OpenGL is an window-system independent API that does not provide any mechanism for creating or managing rendering contexts or for receiving user input. Instead, there are various extensions, such as GLX for X Windows and GLW for Microsoft Windows, that provide system-specific mechanisms for specifying, creating, and managing rendering contexts, and user input is left to the host system APIs. Since developing applications that directly use these APIs to manage windows is a significant porting effort, a number of libraries have been developed that provide an abstraction layer over the system-specific APIs. One such library is GLFW [5], which is a lightweight C library that supports display and window management and various input devices.

The interface to GLFW requires some additional glue code to properly handle callback functions. In GLFW, callbacks for handling user input events are defined per-window. Unfortunately, MLton's support for exporting ML functions to C only allows one instance of the function to exist. Thus, we have to keep an ML-side registry of windows and their callback functions, which is used to multiplex the the single instance of the callback.

Other libraries Graphics applications often need other services, such as audio, model loading, *etc.*. We support some of these with the following components:

- SML3d provides support for using the OpenAL [1] library for audio playback.
- OpenCL is an language for general-purpose computing on GPUs [6] coupled with a supporting library for managing OpenCL programs on the host CPU. Because OpenCL is useful in its own right, we also distribute the OpenCL library as a standalone library (it incorporates the Raw Data library described above).
- There are dozens (if not hundreds) of different file formats for representing 3D objects.
- A long-term goal of the SML3d project is to provide a platform for experimenting with higher-level programming models for graphics. One early example of this is an embedded DSL for specifying particle-system effects [7].

5. Type safety

The OpenGL API is rife with the potential for uncaught type errors; some of these errors can be caught by the GL, but others can result in undefined behavior and even crashes. One of the main goals of SML3d is to provide as much type safety for OpenGL programming as possible. We use three basic techniques to improve the type safety of the OpenGL API. First, we introduce distinct abstract types for the various conceptual types used in OpenGL. For example, the type of OpenGL primitives has the following SML specification:

```
structure PrimitiveType :> sig
    eqtype t

    val LINES : t
    val LINES_ADJACENCY : t
    val LINE_LOOP : t
    val LINE_STRIP : t
    val LINE_STRIP_ADJACENCY : t
    val PATCHES : t
    val POINTS : t
    val TRIANGLES : t
    val TRIANGLES_ADJACENCY : t
    val TRIANGLE_FAN : t
    val TRIANGLE_STRIP : t
    val TRIANGLE_STRIP_ADJACENCY : t

  end (* PrimitiveType *)
```

Second, we specialize the types of generic functions, such as `glGetFloatv`, for their specific uses. Lastly, we make heavy use of phantom types [9] to constrain the types of polymorphic operations. For example, the function for loading data into a 2D texture constrains the internal texture format, the external format, and the data source to all match:

```
val image2D : {
    target : target2D, level : int, border : int,
    internalFmt : 'a internal_format,
    format : 'a Pixel.format,
    img : 'a Image.image2D
  } -> unit
```

6. Foreign interfaces

A major part of SML3d's implementation is the interface between SML code and the underlying C APIs. The MLton implementation of SML has convenient mechanisms for interfacing to C libraries. It supports native base types (e.g., 32-bit integers) efficiently and syntax for importing C functions into ML and exporting ML functions to C. Furthermore, MLton supports passing ML arrays and vectors

[3] The reader might be surprised that JPEG is not included in this list, but it is a lossy format that is known to have issues with texture mapping.

```
val glBufferData   = _import "glBufferData" stdcall
                       : (glenum * glsizeiptr * ptr * glenum) -> unit;
val glBufferDataus = _import "glBufferData" stdcall
                       : (glenum * glsizeiptr * glushort vector * glenum) -> unit;
val glBufferDataf  = _import "glBufferData" stdcall
                       : (glenum * glsizeiptr * glfloat vector * glenum) -> unit;
```

Figure 1. Binding `glBufferData` at multiple SML types

of base types directly to C, and provides support for directly manipulating C pointers. With these features, it is possible to directly interface most of the OpenGL API in a low-overhead way.

One major challenge is that OpenGL is a very large API with over 500 functions and several thousand symbolic constants. In addition to the core API, OpenGL has over one hundred extensions and new extensions are added frequently.[4] Writing a foreign interface to OpenGL by hand would be extremely time consuming. Therefore, to handle the the scale of the OpenGL API, we use a custom-built program-generation toolchain. We start with the OpenGL API specifications that are published by Khronos. Originally, these were in a undocumented format that had to be reverse engineered, but they are now represented as XML files with a well-documented schema [8]. We convert these files to our own XML-format database. Our toolchain supports merging updates to the Khronos specifications into the database. The database is then used to generate glue code.

In addition to the data extracted from the Khronos specifications, our database includes additional information about how to bind C functions to ML code. Specifically, it is not always possible to automatically generate the correct SML types for a C function from the information in the Khronos specifications. Furthermore, we have found it useful to bindmultiple SML variables with different types to a single C function. For example, consider the OpenGL function

```
void glBufferData (GLenum target,
  GLsizeiptr size, const void *data, GLenum usage);
```

which is used to load data into a buffer object in the GL. Since buffers can hold signed and unsigned integer data of various sizes, as well as floating-point data, we bind the C function to multiple SML variables at different types as is shown in Figure 1. In this code, we are defining versions of `glBufferData` that accept C pointers (`ptr`), vectors of 16-bit words (`glushort vector`), and vectors of 32-bit floats (`glfloat vector`).

The generated glue code is not directly exported to the user, but is instead used to implement the SML3d library. The code wrapping the generated C function bindings is mainly concerned with imposing additional type constraints. For example, we define the user-level interface:

```
eqtype face
val FRONT : face
val BACK  : face
val BOTH  : face

val cullFace : face -> unit
```

which is implemented as

```
type face = glenum
val FRONT : face = GLConsts.GL_FRONT
val BACK  : face = GLConsts.GL_BACK
val BOTH  : face = GLConsts.GL_BOTH

val cullFace = GLFuncsBase.glCullFace
```

where `GLConsts` and `GLFuncsBase` are generated from the database. SML's opaque signature matching makes `face` an abstract type, but its machine representation is the same as the C APIs, so there is no additional overhead for the additional type safety.

In addition to OpenGL, we also build on other C APIs, such as GLFW [5], libpng [12], libtiff [16], and OpenAL [1]. For these libraries, the glue code is a mix of C and SML.

7. Build support

Building an SML3d application requires many system-dependent configuration choices. Examples include

- Different FI mechanisms for binding to OpenGL's API: on Microsoft Windows we must use a dynamic linking scheme for most (but not all) of the OpenGL API.[5]
- Different linking options are required to link with the OpenGL API (*e.g.*, `--framework OpenGL` on MacOS X versus `-lgl` on Linux).
- Libraries, such as image I/O, depend on C libraries and, in some cases, glue code, which also need to be included in the linking command.

To avoid shouldering the user with this complexity, we provide a shell script that generates the necessary MLton command-line options for a given program's requirements. For example, an SML3d application that targets OpenGL 3.2 Core API and uses GLFW and the Image I/O library can be compiled with the following shell commands:

```
FLGS=`sml3d-config gl32 glfw image-io`
C_OBJS=`sml3d-config objs gl32 glfw image-io`
mlton $FLGS -output myprog myprog.mlb $C_OBJS
```

The first call to the `sml3d-config` script returns the MLton command-line flags needed to compile and link the program, which include specifying the MLton path map so that MLton can find mlb files, specifying the directory of the requested OpenGL version, and specifying the C linker options needed to include the necessary C libraries. The second call returns a list of extra object files needed for the program (in this case, these would be the image I/O glue code object files).

8. Status and future work

SML3d is a work in progress. Previous versions of SML3d supported the OpenGL 2.1 API. Once the OpenGL 3.2 core API became widely available, we decided to remove support for the old fixed pipeline and do a complete rewrite of the library to support modern graphics programming techniques. This rewrite was a major undertaking, but a first usable version is almost in place (we are still doing some fine tuning of the texture interface).

Many other parts of SML3d in place, such as the Raw Data, Image I/O, GLFW, OpenAL, and OpenCL libraries. The other libraries, such as mesh loading and our particle-system embedded

[4] OpenGL's extensions provide a mechanism for hardware and system vendors to define new features.

[5] This requirement is because Microsoft does not support OpenGL beyond version 1.1; the GPU drivers provide the more recent parts of the API.

DSL, and the various test and example programs have yet to be ported to the new design.

We plan a number of additional libraries in the future, such as support for immediate-mode rendering of simple geometry (something that was lost in the transition from the fixed to the programmable pipeline) and support for font rendering (based on the FreeType2 library [3]). At some point, we would also like to port SML3d to other implementations of SML.

SML3d is distributed as Open Source and information about it can be found at

`http://sml3d.cs.uchicago.edu`

and suggestions, bug reports, and contributions are all welcome.

Acknowledgments

Pavel Krajcevski and Sam Quinan have made significant contributions to earlier versions of the SML3d code; Matt Rice has provided a number of fixes; and Matthew Fluet has significant support in the understanding the best way to handle various problems in MLton.

References

[1] `http://openal.org`.

[2] Apple, Inc. *Scene Kit Programming Guide*, July 2012.

[3] `http://freetype.org`.

[4] `http://opengl.org`.

[5] `http://www.glfw.org`.

[6] Khronos OpenCL Working Group. *The OpenCL Specification (Version 1.2)*, November 2011. Available from `http://www.khronos.org/opencl`.

[7] Pavel Krajcevski and John Reppy. A declarative API for particle systems. In *PADL '11*, volume 6539 of *LNCS*, pages 130–144, New York, NY, January 2011. Springer-Verlag.

[8] Jon Leech. *The Khronos XML API Registry*, September 2013. Available at `https://cvs.khronos.org/svn/repos/ogl/trunk/doc/registry/public/api/readme.pdf`.

[9] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL '99*, pages 109–122, New York, NY, USA, January 2000. ACM.

[10] `http://mlton.org`.

[11] `http://netpbm.sourceforge.net`.

[12] `http://www.libpng.org`.

[13] `http://hackage.haskell.org/package/OpenGLRaw`.

[14] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. *OpenGL Programming Guide*. Addison Wesley, Upper Saddle River, NJ, 8th edition, 2013.

[15] `http://www.pllab.riec.tohoku.ac.jp/smlsharp`.

[16] `http://www.libtiff.org`.

[17] Truevision, Inc., Indianapolis, IN. *Truevision TGA FILE FORMAT SPECIFICATION*, version 2.0 edition, 1991.