Proceedings, ACM Multimedia 97 Conference, Seattle, WA, November 1997.

Flavor: A Language for Media Representation

Alexandros Eleftheriadis
www.ee.columbia.edu/~eleft

Department of Electrical Engineering
and Columbia New Media Technology Center
Columbia University
New York, NY 10027, USA

Abstract
We present the design and implementation of a new programming language for media-intensive applications called Flavor (Formal Language for Audio-Visual Object Representation). It is an extension of C++ and Java in which the typing system is extended to incorporate bitstream representation semantics. This allows to describe in a single place both the in-memory representation of data as well as their bitstream-level (compressed) representation as well. We have developed software tools (www.ee. columbia.edu/flavor) that automatically generate standard C++ and Java code from the Flavor source code, so that direct access to compressed multimedia information by application developers can be achieved with essentially zero programming.

1. Introduction

Flavor originated from the need to simplify and speed up the development of software that processes coded audiovisual or general multi-media information. This includes encoders and decoders as well as applications that manipulate such information. Examples include editing tools, content creation tools, multimedia indexing and searching engines, etc.

Such information is invariably encoded in a highly efficient form, to minimize the cost of storage and transmission. This source coding [1] operation is almost always performed in a bitstream-oriented fashion: the data to be represented is converted to a sequence of binary values of arbitrary (and typically variable) lengths, according to a specified syntax. The syntax itself can have various degrees of sophistication. One of the simplest forms is the GIF87a format [2], consisting of essentially two headers and blocks of coded image data using Lempel-Ziv-Welch compression. Much more complex formats include JPEG, MPEG-2 [3] and the forthcoming MPEG-4 specifications [4, 5, 10], among others.

General-purpose programming languages such as C++ [6] and Java [7] do not provide native facilities for coping with such data. Software codec or application developers need to build their own facilities, involving two components. First, they need to develop software that deals with the bitstream-oriented nature of the data, as general-purpose microprocessors are strictly byte-oriented. Secondly, the need to implement parsing and generation code that complies with the syntax of the format at hand (be it proprietary or standard). These two tasks represent a significant amount of the overall development effort. They also have to be duplicated by everyone who requires access to a particular compressed representation within their application. Furthermore, they can also represent a substantial percentage of the overall execution time of the application.

Flavor addresses these problems in an integrated way. First, it allows the formal description of the bitstream syntax. Formal here means that the description is based on a well-defined grammar, and as a result is amenable to software tool manipulation. In the past such descriptions were using ad-hoc conventions involving tabular data or pseudo-code.

A second and key aspect of our approach is that this description has been designed as an extension of C++ and Java, both heavily used object-oriented languages in multimedia applications development. This ensures seamless integration of Flavor code with both C++ and Java code and the overall architecture of an application.

Flavor was designed as an object-oriented language, anticipating an audiovisual world comprised of audiovisual objects, both synthetic and natural, and combining it with well-established paradigms for software design and implementation. Its object-oriented facilities go beyond the mere duplication of C++ and Java features, and introduce several new concepts that are pertinent for bitstream-based media representation.

In order to validate the expressive power of the language, several existing bitstream formats have already been described in Flavor, including sophisticated structures such as MPEG-2 Video and Audio. A translator has also been developed for translating Flavor code to C++ and Java.

In the following, we first present a brief overview of the language in terms of its history and technical approach. We then describe each of its features, including declarations and constants, expressions and statements, classes, scoping rules, and maps. We also briefly describe the translator and its simple run-time API. We conclude with an overview of the benefits of using the Flavor approach for media representation. More detailed information and publicly available software can be found in the Flavor Web site at: www.ee. columbia.edu/flavor.

2. Overview

2.1 A Brief History

Flavor has its origins in a Perl script (`mkvlc`) that the author developed in early 1994 in order to automate the (laborious)

generation of C code declarations for variable-length code tables of the MPEG-2 Video specification. In November 1995, the ideas behind `mkvlc` took a more concrete shape in the form of a "syntactic description language," [8, 9, 11] i.e., a formal way to describe not just variable length codes, but the entire structure of a bitstream. Such a facility was proposed to the MPEG-4 standardization activity, which at that time had started to consider flexible, even programmable, audiovisual decoding systems. The language subsequently underwent a series or revisions benefiting from input from several participants in the MPEG-4 standardization activity, and its specification is now fairly stable. We should note that Flavor is currently used in MPEG-4 for the description of the bitstream syntax.

2.2 Technical Approach

Flavor provides a formal way to specify how data is laid out in a serialized bitstream. It is based on a *principal of separation* between bitstream parsing operations and encoding, decoding and other operations. This separation acknowledges the fact that the same syntax can be utilized by different tools, but also that the same tool can work unchanged with a different bitstream syntax. For example, the number of bits used for a specific quantity can change without modifying any part of the application program.

Past approaches for syntax description utilized a combination of tabular data, pseudo-code, and textual description to describe the format at hand. Taking MPEG as an example, both MPEG-1 and MPEG-2 specifications were described using a C-like pseudo-code syntax (originally introduced by Milt Anderson, Bellcore), coupled with explanatory text and tabular data. Several of the lower and most sophisticated layers could only be handed by explanatory text. The text had to be carefully crafted and tested over time for ambiguities. Other specifications (e.g., JPEG, GIF) use similar bitstream representation schemes, and hence share the same limitations.

Other formal facilities already exist for representing syntax. One important example is ASN.1 (ISO International Standards 8824 and 8825). A key difference, however, is that ASN.1 was not designed to address the intricacies of source coding operations, and hence cannot cope with, for example, variable length coding. In addition, ASN.1 tries to hide the bitstream representation from the developer by using its own set of binary encoding rules, whereas in our case the binary encoding is the actual target of the description.

There is also some remote relationship between syntax description and "marshalling," a fundamental operation in distributed systems where consistent exchange of typed data is ensured. Examples in this category include Sun's ONC XDR (External Data Representation) and the `rpcgen` compiler which automatically generates marshalling code, as well as CORBA IDL, among others. These ensure, for example, that even if the native representation of an integer in two systems is different (big versus little endian), they can still exchange typed data in a consistent way. Marshalling, however, does not constitute bitstream syntax description because: 1) the programmer does not have control over the data representation (the binary representation for each data type is predefined), 2) it is only concerned with the representation of simple serial structures (lists of arguments to functions, etc.). As in ASN.1, the binary representation is "hidden" and is not amenable to customization by the developer. One could parallel Flavor and marshalling by considering the Flavor source as the XDR layer. A better parallelism would be to view Flavor as a parser-generator like `yacc`, but for bitstream representations.

It is interesting to note that all prior approaches to syntactic description where concerned only with the definition of message structures typically found in communication systems. These tend to have a much simpler structure compared with coded representations of audio-visual information (compare the IP header with the baseline JPEG specification, for example).

Flavor was designed to be an intuitive and natural extension of the typing system of object-oriented languages like C++ and Java. This means that the bitstream representation information is placed together with the data declarations in a single place. In C++ and Java, this place is where a class is defined.

Flavor has been explicitly designed to follow a declarative approach to bitstream syntax specification. In other words, the designer is specifying how the data is laid out on the bitstream, and does not detail a step-by-step procedure that parses it. This latter procedural approach would severely limit both the expressive power as well as the capability for automated processing and optimization, as it would eliminate the necessary level of abstraction. As a result of this declarative approach, Flavor does not have functions or methods.

A related example from traditional programming is the handling of floating point numbers. The programmer does not have to specify how such numbers are represented or how operations are performed; these tasks are automatically taken care of by the compiler in coordination with the underlying hardware or run-time emulation libraries.

An additional feature of combining type declaration and bitstream representation is that the underlying object hierarchy of the base programming language (C++ or Java), becomes quite naturally the object hierarchy for bitstream representation purposes as well. This is an important benefit for ease of application development, and it also allows Flavor to have a very rich typing system itself.

"Hello Bits"

The following trivial example indicates how the integration of type and bitstream representation information is accomplished. Consider a simple object called `HelloBits` with just a single pixel, represented using 8 bits. Using the MPEG-1/2 methodology, this would be described as follows.

HelloBits () {	**No. of Bits**	**Mnemonic**
Bits	8	uimsbf
}		

Example 1: HelloBits using MPEG-1/2.

A C++ description of this single pixel object would include a method to read its value, and have a form similar to the one shown in Example 2. Here `getuint()` is assumed to be a function that reads bits from the bitstream (here 8) and returns them as an unsigned integer (by default with the most significant bit first). When `HelloBits:: get()` is called, the bitstream is read and the resultant quantity is placed in the data member `Bits`.

```
class HelloBits {
    unsigned int Bits;
    void get() {
        Bits=::getuint(8);
    }
};
```

Example 2: HelloBits using C++ (a similar construct would be used for Java as well).

In Flavor, the same description would be done as follows.

```
class HelloBits {
    unsigned int(8) Bits;
}
```

Example 3: HelloBits using Flavor.

As we can see, the bitstream representation is integrated with the type declaration. The above should be read as: `Bits` is an unsigned integer quantity represented using 8 bits in the bitstream. Note that there is no implicit encoding rule as in ASN.1: the rule here is embedded in the type declaration ans indicates that, when the system has to parse a `HelloBits` data type, it will just read the next 8 bits as an unsigned integer and assign them to the variable `Bits`.

These examples, although trivial, demonstrate the differences between the various approaches. In Example 1, we just have a tabulation of the various bitstream entities, grouped into syntactic units (here `HelloBits`). This style is sufficient for straightforward representations, but fails when more complex structures are used (e.g., variable length codes).

In Example 2, the syntax is incorporated into hand-written code embedded in a `get()` or equivalent method. As a result, the syntax becomes an integral part of the decoding method even though the same decoding mechanism could be applied to a large variety of similar syntactic constructs. Also, it quickly becomes overly verbose.

Flavor provides a wide range of facilities to define sophisticated bitstreams, including `if-else`, `switch`, `for`, and `while` constructs. In contrast with regular C++ or Java, these are all included in the data declaration part of the class, so they are completely disassociated from code that belongs to class methods. This is in line with the declarative nature of Flavor, where the focus is on defining the structure of the data, not operations on them.

In order to be usable in actual programs, Flavor source is translated to regular C++ or Java code with each Flavor class creating an equivalent C++/Java class. Two methods are automatically generated by the translator for each class: a `get()` method that will read data from a bitstream and load it to the class variables, and a `put()` method which will take the values from these variables and place them in the bitstream using the specified syntax.

In the following we describe each of the language features in more detail, emphasizing the differences between C++ and Java. In order to ensure that Flavor semantics are in line with both C++ and Java, whenever there was a conflict a common denominator approach was used.

3. Declarations and Constants

3.1 Literals

All traditional C++ and Java literals are supported by Flavor with the exception of strings. This includes integers, floating point numbers, and character constants (e.g., `'a'`). In addition, Flavor defines a special binary number notation using the prefix `0b`. In addition to specifying the actual value, binary literals also convey their length. For example, one can write `0b011` to denote the number 3 represented using 3 bits. For readability, a bitstring can include periods every four digits, e.g., `0b0010.01`. Hexadecimal or octal constants used in a context of a bitstring also convey their length in addition to their value. Whenever the length of a bitstring literal is irrelevant, it is treated as a regular integer literal.

3.2 Comments

Both multi-line (`/**/`) and single-line (`//`) comments are allowed. Multi-line comment delimiters cannot be nested.

3.3 Names

Variable names follow the C++ and Java conventions (e.g., variable names cannot start with a number). Several keywords that are used in C++ and Java are considered reserved in Flavor.

3.4 Types

Flavor supports the common subset of C++ and Java built-in or fundamental types. This includes `char`, `int`, `float`, and `double` including all appropriate modifiers. In addition, Flavor defines a new type called `bit`. This is to accommodate bitstring variables.

In addition, it allows the declaration of new types in the form of classes (see Section 5).

Flavor does not support pointers, references, casts, or C++ operators related to pointers. It also does not support structures or enumerations.

3.5 Declarations

Regular variable declarations can be used in Flavor in the same way as in C++ and Java. As Flavor follows a declarative approach, constant variable declarations with specified values are allowed everywhere (there is no constructor to set the initial values). This means that the declaration '`const int a=1;`' is valid anywhere (not just in global scope). The two major differences are the declaration of parsable variables and arrays.

3.5.1 Parsable Variables

Parsable variables are the core of Flavor's design; it is the proper definition of these variables that defines the bitstream syntax.

Parsable variables include a parse length specification immediately after their type declaration, as shown in Figure 1. *length* can be an integer constant, a non-constant variable of type compatible to `int`, or a `map` (discussed later on) with the same

type as the variable. This means that the parse length of a variable can be controlled by another variable.

`aligned`(*length*) *type*(*length*) *variable*;

Figure 1: Parsable variable declaration.

In addition to the parse length specification, parsable variables can also have the modifier `aligned`. This signifies that the variable begins at the next integer multiple boundary of the *length* specified within the alignment expression. If this length is omitted, an alignment size of 8 is assumed (byte boundary). Only multiples of 8 are allowed. For parsing, any intermediate bits are ignored, while for output bitstream generation the bitstream is padded with zeros.

As we will see later on, parsable variables cannot be assigned to. This ensures that the syntax is preserved regardless if we are performing an input or output operation. However, parsable variables *can be redeclared*, as long as their type remains the same, only the parse size is changed, and the original declaration was not as a `const`. This allows one to select the parse size depending on the context (see Exressions and Statements, Section 4). In addition, the obey special scoping rules as we will see later on.

In general, the parse size expression must be a non-negative value. The special value 0 can be used when, depending on the bitstream context, a variable is not present in the bitstream but obtains a default value. In this case no bits will be parsed or generated, however the semantics of the declaration will be preserved.

Finally, variables of type `float`, `double`, and `long double` are only allowed to have a parse size equal to the fixed size that their standard representation requires (32 and 64 bits).

3.5.2 Look-Ahead Parsing

In several instances it is desirable to examine the immediately following bits in the bitstream, without actually removing the bits. To support this behavior, a '`*`' character can be placed after the parse size parentheses to modify the parse size semantics. Note that for bitstream output purposes this has no effect.

3.5.3 Parsable Variables with Expected Values

Very often, certain parsable variables in the syntax have to have specific values (markers, start codes, reserved bits, etc.). These are specified as initialization values for parsable variables. Figure 2 shows an example.

```
int(3) a = 2;
```

Figure 2: Example of declaration of parsable variable with expected value.

This is interpreted as: `a` is an integer represented with 3 bits, and must have the value 2. The keyword `const` may be prepended in the declaration, to indicate that the parsable variable will have this constant value and, as a result, cannot be redeclared.

As both parse size and initial value can be arbitrary expressions, we should note that the order of evaluation is parse expression first, followed by the initializing expression.

3.5.4 Arrays

Arrays have special behavior in Flavor, due to its declarative nature but also due to the desire for very dynamic type declarations. For example, we want to be able to declare a parsable array with different array sizes depending on the context. In addition, we may need to load the elements of an array one at a time (this is needed when the retrieved value indicates indirectly if further elements of the array should be parsed). These concerns are only relevant for parsable variables.

The array size, then, does not have to be a constant expression (as in C++ and Java), but it can be a variable as well. The following is allowed in Flavor.

```
int a = 2;
int(2) A[a++];
```

Figure 3: Array declaration with dynamic size specification.

An interesting question is how to handle initialization of arrays, or parsable arrays with expected values. As the size of the array may not be known until run-time, the usual brace expression initialization (e.g., '`int a[2] = {1, 2};`') cannot be used. The mechanism we provided involves the specification of a single expression as the initializer. For example, we can write:

```
int A[3]= 5;
```

Figure 4: Array declaration with initialization.

This means that all elements of `A` will be initialized with the value 5. In order to provide more powerful semantics to array initialization, Flavor considers the parse size and initializer expressions as executed *per each element of the array*. The array size expression, however, is only executed once, and before the parse size expression or the initializer expression. Let's look at a more complicated example.

```
int a=1;
int(a++) A[a++]=a++;
```

Figure 5: Array declaration initialization with dynamic array and parse sizes.

Here `A` is declared as an array of 2 integers. The first one is parsed with 3 bits and is expected to have the value 4, while the second is parsed with 5 bits and is expected to have the value 6. After the declaration, `a` is left with the value 7.

This probably represents the largest deviation of Flavor's design from C++ and Java declarations. On the other hand it does provide significant flexibility in constructing sophisticated declarations in a very compact form, and it is also in line with the dynamic nature of variable declarations that Flavor provides.

3.5.5 Partial Arrays

An additional refinement of array declaration is partial arrays. These are declarations of parsable arrays in which only a subset of the array needs to be declared (or, equivalently, parsed from or written to a bitstream). Flavor introduces a double brace notation for this purpose. The following examples demonstrate its use.

```
int(2) A[[3]]=1;
int(4) B[[2]][3];
```

Figure 6: Partial arrays.

In the first line, we are declararing the 4-th element of `A` (array indices start from 0). The array size is unknown at this point, but of course it will be considered at least 4. In the second line, we are declaring a two-dimensional array, and in particular only its third

column (assuming the first index corresponds to a row). The array indices can, of course, be expressions themselves. Partial arrays can only appear on the left-hand side of declaration and are not allowed in expressions.

4. Expressions and Statements

Flavor supports all of the C++ and Java arithmetic, logical, and assignment operators. However, parsable variables cannot be used as lvalues. This ensures that they always represent the bitstream's contents, and allow consistent operation for the translator-generated `put()` and `get()` methods.

Flavor also supports all the familiar flow control statements: `if-else`, `do-while`, `while`, and `switch`. In contrast with C++ and Java, variable declarations are not allowed within the arguments of these statements (i.e., '`for(int i=0; ; );`' is not allowed. This is because in C++ the scope of this variable will be the enclosing one, while in Java it will be the enclosed one. To avoid confusion, we opted for the exclusion of both alternatives at the expense of a slightly more verbose notation. Scoping rules are discussed in detail in Section 6.

The following is an example of the use of these flow control statements.

```
if (a==1){
    int(3) b;
}
else {
    int(4) b;
}
```

Figure 7: Example of conditional expression.

The variable `b` is declared with a parse size of 3 if `a` is equal to 1, and with a parse size of 4 otherwise. Observe that this construct would not be meaningful in C++ or Java as the two declarations would be considered as being in separate scopes. This is the reason why parsable variables need to obey slightly different scoping rules than regular variables. The way to approach this to avoid confusion is to consider that Flavor is designed so that these parsable variables can be properly defined at the right time and position. All the rest of the code is there to ensure that this is the case. We can consider the parsable variable declarations as "actions" that our system will perform at the specified times. Then this difference in scoping rules becomes very natural.

5. Classes

Flavor uses the notion of classes in exactly the same way as C++ and Java do. It is the fundamental structure in which object data is organized. Keeping in line with the support of both C++ and Java-style programming, classes in Flavor cannot be nested, and only single inheritance is supported. In addition, due to the declarative nature of Flavor, methods are not allowed (this includes constructors and destructors).

The following is an example of a simple class declaration with just two parsable member variables.

```
class SimpleClass {
    int(3) a;
    unsigned int(4) b;
};  // trailing ';' optional
```

Figure 8: A simple class declaration.

The trailing '`;`' character is optional accommodating both C++ and Java-style class declarations. This class defines objects which contain two parsable variables. They will be present in the bitstream in the same order they are declared. After this class is defined, we can declare objects of this type:

```
SimpleClass a;
```

Figure 9: A simple class variable declaration.

A class is considered parsable if it contains at least one variable that is parsable. Declaration of parsable class variables can be prepended by the `aligned` modifier in the same way as simple parsable variables.

Class member variables in Flavor do not require access modifiers (`public`, `protected`, `private`). In essence, all such variables are considered public.

5.1 Parameter Types

As Flavor classes cannot have constructors, it is necessary to have a mechanism to pass external information to a class. This is accomplished using *parameter types*. These act the same way as formal arguments in function or method declarations do. They are placed after the name of the class.

```
class SimpleClass(int i[2]) {
    int(3) a=i[1];
    unsigned int(4) b=i[2];
};  // trailing ';' optional
```

Figure 10: A simple class declaration with parameter types.

When declaring variables of parameter type classes, it is required that actual arguments are provided in place of the formal ones:

```
int(2) v[2];
SimpleClass a(v);
```

Figure 11: A simple class declaration with parameter types.

Of course the types of the formal and actual parameters must match. For arrays, only their dimensions are relevant; their actual sizes are not significant as they can by dynamically varying. Note that class types are allowed in parameter declarations as well.

5.2 Inheritance

As we mentioned earlier, Flavor supports single inheritance so that compatibility with Java is maintained. Although Java can "simulate" multiple inheritance through the use of interfaces, Flavor has no such facility (it would be meaningless since methods do not exist in Flavor). However, for media representation purposes, we have not found any instance where multiple inheritance would be required, or be even desirable. It is interesting to note that all existing representation standards today are not truly object-based. The only exception, to our knowledge, is the MPEG-4 specification which explicitly addresses the representation of audio-visual objects. It is, of course, possible to describe existing structures in an object-oriented way but it does not truly map one-to-one with the notion of objects. For example, MPEG-2 Video slices can be considered as separate objects of the same type, but of course their semantic interpretation (horizontal stripes of macroblocks) is not very useful.

Derivation in C++ and Java is accomplished using a different syntax (`extends` versus ':'). Here we opted for the Java notation (also ':' is used for object identifier declarations as explained below). Unfortunately, it was not possible to satisfy both.

```
class A {
    int(2) a;
}

class B extends A {
    int(3) b;
}
```

Figure 12: Derived class declaration.

In Figure 12 we show a simple example of a derived class declaration. Derivation from a bitstream representation point of view means that `B` is an `A` with some additional information. In other words, the behavior would be almost identical if we just copied the statements between the braces in the declaration of `A` in the beginning of `B`. We say almost here because scoping rules of variable declarations also come into play here, as discussed in Section 6.

Note that if a class is derived from a parsable class, it is considered parsable as well.

5.3 Polymorphic Parsable Classes

The concept of inheritance in object-oriented programming derives its power from its capability to implement polymorphism. In other words, the capability to use a derived object in a place where an object of the base class is expected. Although the mere structural organization is useful as well, it could be accomplished equally well with containment (a variable of type `A` is the first member of `B`).

Polymorphism in traditional programming languages is made possible via vtable structures, which allow the resolution of operations during run-time. Such behavior is not pertinent for Flavor, as methods are not allowed.

A more fundamental issue, however, is that Flavor describes the bitstream syntax: the information with which the system can detect which object to select *must be present in the bitstream*. As a result, traditional inheritance as defined in the previous section *does not* allow the representation of polymorphic objects. Considering Figure 11, there is no way to figure out by reading a bitstream if we should read an object of type `A` or type `B`.

Flavor solves this problem by introducing the concept of *object identifiers* or IDs. The concept is rather simple: in order to detect which object we should parse/generate, there must be a parsable variable that will identify it. This variable must have a different expected value for any class derived from the originating base class, so that object resolution can be uniquely performed in a well-defined way (this is checked by the translator). As a result, object ID values must be constant expressions.

In order to signify the importance of ID variables, they are declared immediately after the class name (including any derivation declaration) and before the class body. They are separated from the class name declaration using a colon (':'). We could rewrite the example of Figure 12 with IDs as shown in Figure 13

The name and the type of the ID variable is irrelevant, and can be anything that the user chooses. It cannot, however, be an array, or a class variable (only built-in types are allowed). Also, the name, type, and parse size must be identical between the base and derived classes.

```
class A : int(1) id=0 {
    int(2) a;
}

class B extends A
    : int(1) id=1 {
    int(3) b;
}
```

Figure 13: Derived class declaration with object identifiers.

The semantics of the object identifiers in Figure 13 are the following. Upon reading the bitstream, if the next 1 bit has the value 0 an object of type `A` will be parsed; if the value is 1 then an object of type `B` will be parsed. For output purposes, and as will be discussed in Section 8, it is up to the user to set up the right object type in preparation for output.

Object identifiers are not required for all derived classes of a base class that has a declared ID. This allows, for example, to have the following inheritance tree.

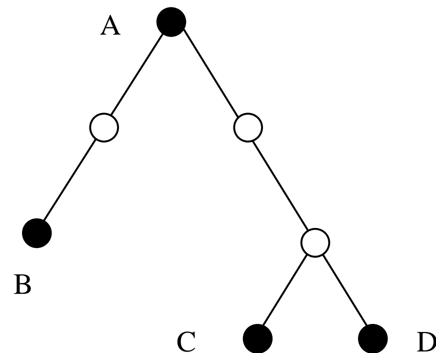

Figure 14: Class inheritance tree; not all classes have to have object identifiers.

Here only the classes represented by the black circles have IDs. As a result, only classes A, B, C, and D can be used wherever an A can appear; the intermediate classes cannot.

This type of polymorphism is already used in the MPEG-4 Systems specification, and in particular the Binary Format for Scenes (BIFS) [12]. This is a VRML-derived set of nodes that represent objects and operations on them, thus forming a hierarchical description of a scene.

ID variables are always considered constant, i.e., they cannot be redeclared within the class. This is the same as if the keyword `const` was prepended in their declaration.

6. Scoping Rules

The scoping rules that Flavor uses are identical with C++ and Java with the exception of parsable variables.

As in C++ and Java, a new scope is introduced with curly braces ({ }). Since Flavor does not have functions or methods, a scope can either be the global one or a scope within a class declaration.

The global scope cannot contain any parsable variable, since it does not belong to any object. As a result, global variables can only be constants.

Within a class, all parsable variables are considered as class member variables, regardless of the scope they are encountered in. This is essential in order to allow conditional declarations of variables which will almost always require that the actual declarations occur within compound statements (see Figure 7). Non-parsable variables that occur in the top-most class scope are also considered class member variables. The rest live within their individual scopes.

This distinction is important in order to understand which variables are accessible to a class variable that is contained in another class. The issues are illustrated in Figure 15. Looking at class `A`, the initial declaration of `i` occurs in the top-most class scope; as a result `i` is a class member. `a` is declared as a parsable variable, and hence it is automatically a class member variable. The declaration of `j` occurs in the scope enclosed by the `if` statement; as this is not the top-level scope, `j` is not a class member. The following declaration of `i` is acceptable; the original one is hidden within that scope. Finally, the declaration of the variable `a` as a non-parsable would hide the parsable version. As parsable variables do not obey scoping rules, this is not allowed (hiding parsable variables of a base class, however, is allowed).

```
class A  {
  int i=1;
  int(2) a;
  if (a==2) {
    int j=i;
    int i=2; // hides i, ok
    int a;   // hides a, error
  }
}

class B {
  A a;
  a.j=1; // error, j not a
         //class member
  int j=a.a+1; // ok
  j=a.i+2; // ok
  int(3) b;
}
```

Figure 15: Scoping rules example.

Looking now at the declaration of class `B` which contains a variable of type `A`, it becomes clear which variables are available as class members.

In summary, the scoping rules have the following two special considerations. Parsable variables do not obey scoping rules and are always considered class members. Non-parsable variables obey the standard scoping rules and are considered class members only if the are at the top-level scope of the class.

Note that parameter type variables are considered as having the top-level scope of the class. Also, they are not allowed to hide the object identifier, if any.

7. Maps

Up to know, we have only considered fixed-length representations, either constant or parametric. A wide variety of representation schemes, however, rely heavily on entropy coding, and in particular Huffman codes [1]. These are variable length codes (VLCs) which are uniquely decodable (no codeword is the prefix of another). Flavor provides extensive support for variable length coding through the use of maps. These are declarations of tables in which the correspondence between codewords and values is described. The following is a simple example of a map declaration.

```
map A(int) {
  0b0, 1,
  0b01, 2
}
```

Figure 16: A simple map declaration.

The `map` keyword indicates the declaration of a map named `A`. The declaration also indicates that the map converts from bitstring values to values of type `int`. The type indication can be a fundamental type, a class type, or an array. Map declarations can only occur in global scope. As a result, an array declaration will have to have a constant size (no non-constant variables are visible at this level).

The map contains a series of entries. Each entry starts with a bitstring that declares the codeword of the entry, followed by the value to be assigned to this codeword. If a complex type is used for the mapped value, then the values have to be enclosed in curly braces.

After the map is properly declared, we can now define parsable variables that use it by indicating the name of the map where we would put the parse size expression. For example:

```
int(A) i;
```

Figure 17: Declaring a variable with a variable length code table.

As we can see, the use of VLCs is essentially identical to fixed-length variables. All the details are hidden away in the map declaration.

The translator can check that the VLC table is uniquely decodable, and also generate optimized tables for extremely fast decoding using lookup tables.

As Huffman codeword lengths tend to get very large when their number increases, it is typical to specify “escape codes,” signifying that the actual value will be subsequently represented using a fixed-length code. To accommodate these as well as more sophisticated constructs, Flavor allows the use of parsable type indications in map values. This means that, using the example of Figure 16, we can write:

```
map A(int) {
  0b0,   1,
  0b01,  2,
  0b001, int(5)
}
```

Figure 18: Map declaration with extension.

This indicates that, when the bitstring `0b001` is encountered in the bitstream, the actual return value for the map will be obtained by parsing 5 more bits. The parse size for the extension can itself

be a map, thus allowing the cascading of maps in sophisticated ways. Although this facility is efficient when parsing, the bitstream generation operation can be costly when complex map structures are designed this way. None of today's specifications that we are aware of require anything beyond a single escape code.

8. The Flavor Translator

Designing a language like Flavor would be an interesting but academic exercise, unless it was accompanied by software that can put its power into full use. We have developed a translator that evolved concurrently with the design of the language. When the language specification became stable, the translator was completely rewritten. The most recent release (Version 2.0, Interim) is publicly available for downloading (www.ee.columbia.edu/flavor). As it is an interim release, it currently supports only Windows NT/95 and C++ code generation, with support for Java and more operating systems expected in the near future. Earlier versions support a large variety of UNIX systems, but do not support all features.

8.1 Run-Time API

The translator reads a Flavor source file and generates a single `.h` file that contains declarations of all Flavor classes as C++ classes with the appropriate class members. All such members are declared public. More importantly, the translator also generates for each class a `put()` and a `get()` method. The `get()` method is responsible for reading a bitstream and loading the class variables with their appropriate values, while the `put()` method does the reverse.

The translator makes minimal assumptions about the operating environment for the generated code. It requires that a class called `Bitstream` is defined, which provides a small and well-defined set of methods for bitstream I/O. A `Bitstream` reference is passed as an argument to the `get()` and `put()` methods. The Flavor run-time library includes a fast and simple implementation supporting file-based I/O. It is easy to design much more sophisticated, application-specific I/O structures; the only requirement is that the interface exposed is compatible with what the translator expects.

If parameter types are used in a class, then they are also required arguments in the `get()` and `put()` methods as well.

The translator also requires that a function (`flerror`) is available to receive calls when expected values or VLC lookups fail. The function name can be selected by the user; a default implementation is included in the run-time library.

For efficiency reasons, Flavor arrays are converted to fixed size arrays in the translated code. This is necessary in order to allow developers to access Flavor arrays without needing special techniques. Whenever possible, the translator automatically detects and sets the maximum array size; it can also be set by the user using a command-line option. Finally, the run-time library (and the translator) only allow parse sizes of up to the native integer size of the host processor (except for double values). This enables fast implementation of bitstream I/O operations.

For parsing operations, the only task required by the programmer is to declare an object of the class type at hand, and then call its `get()` method with an appropriate bitstream. While the same is also true for the `put()` operation, the application developer must also load all class member variables with their appropriate values before the call is made.

8.2 Verbatim Code

In order to further facilitate integration of Flavor code with C++/Java user code, the translator supports the notion of verbatim code. Using special delimiters, code segments can be inserted in the Flavor source code, and copied verbatim at the correct places in the generated C++/Java file. This allows, for example, the declaration of constructors/destructors, user-specified methods, pointer member variables for C++, etc. Such verbatim code can appear wherever a Flavor statement or declaration is allowed.

The delimiters `%{` and `%}` can be used to introduce code that should go to the class declaration itself (or the global scope). The delimiters `%p{` and `%p}`, and `%g{` and `%g}` can be used to place code at exactly the same position they appear in the `put()` and `get()` methods respectively. Finally, the delimiters `%*{` and `%*}` can be used to place code in both `put()` and `get()` methods.

The Flavor 2.0 release includes several samples on how to integrate user code with Flavor-generated code, including a simple GIF parser.

8.3 Tracing Code Generation

We are also including the option to generate bitstream tracing code within the `get()` method. This will allow one to very quickly examine the contents of a bitstream for development and/or debugging purposes.

9. Concluding Remarks

Flavor's design was motivated by our belief that content creation, access, manipulation, and distribution, will become increasingly important for end-users and developers alike. New media representation forms will continue to be developed, providing richer features and more functionalities for end-users. In order to facilitate this process, it is essential to bring syntactic description on par with modern software development practices and tools. Flavor can provide significant benefits in the area of media representation and multimedia application development at several levels.

First, it can be used as a media representation documentation tool, substituting ad-hoc ways of describing a bitstream's syntax with a well-defined and concise language. This by itself is a substantial advantage for defining specifications, as a considerable amount of time is spent to ensure that such specification are unambiguous and bug-free.

Secondly, a formal media representation language immediately leads to the capability of automatically generating software tools, ranging from bitstream generators and verifiers, as well as a substantial portion of an encoder or decoder.

Third, it allows immediate access to the content by any application developer, for such diverse use as editing, searching, indexing, filtering, etc.

With appropriate translation software, and a bitstream representation written in Flavor, obtaining access to such content is as simple as cutting and pasting the Flavor source code from the specification into an ASCII file, and running the translator.

Flavor, however, does not provide facilities to specify how full decoding of data will be performed as it only addresses bitstream syntax description. For example, while the data contained in a GIF file can be fully described by Flavor, obtaining the value of a particular pixel requires the addition of LZW decoding code that must be provided by the programmer. In several instances, such access is not necessary. For example, a number of tools have been developed to do automatic indexing, search, and retrieval of visual content directly in the compressed domain for JPEG and MPEG content (see [13-15] and references therein). Such tools only require parsing of the coded data so that DCT coefficients are available, but do not require full decoding. Also, emerging techniques, such as MPEG-7 [16], will provide a wealth of information about the content without the need to decode it. In all these cases, parsing of the compressed information may be the only need for the application at hand.

Finally, Flavor can also be used to redefine the syntax of content in both forward and backward compatible ways. The separation of parsing from the remaining coding/decoding operations allows its complete substitution as long as the interface (the semantics of the previously defined parsable variables) remain the same. Old decoding code will simply ignore the new variables, while newly written encoders and decoders will be able to use them. Use of Java in this respect is very useful; its capability to download new class definitions opens the door for such downloadable content descriptions that can accompany the content itself (similar to self-extracting archives). This can eliminate the rigidity of current standards, where even a slight modification of the syntax to accommodate new techniques or functionalities render the content useless in non-flexible but nevertheless compliant decoders.

References

[1] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, Wiley, 1991.

[2] Graphics Interchange Format, CompuServe Inc., 1987, 1989.

[3] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An Introduction to MPEG-2*, Chapman and Hall, 1997.

[4] ISO/IEC JTC1/SC29/WG11 N1643, MPEG-4 Video Working Draft Version 3.0, Bristol, April 1997.

[5] ISO/IEC JTC1/SC29/WG11 N1692, MPEG-4 Systems Working Draft Version 4.0, Bristol, April 1997.

[6] B. Stroustrup, *The C++ Programming Language*, 2nd ed., Addison Wesley, 1993.

[7] K. Arnold, J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[8] A. Eleftheriadis, “A Syntactic Description Language for MPEG-4,” Contribution ISO/IEC JTC1/SC29/WG11 M546, Dallas, November 1995.

[9] Y. Fang and A. Eleftheriadis, “A Syntactic Framework for Bitstream-Level Representation of Audio-Visual Objects,” *Proc., 3rd IEEE Int’l Conf. on Image Processing*, Lausanne, September 1996, pp. II.429–II.432.

[10] ISO/IEC JTC1/SC29WG11 N1631, MPEG-4 Audio Working Draft Version 3.0, Bristol, April 1997.

[11] O. Avaro, P. Chou, A. Eleftheriadis, C. Herpel, and C. Reader, “The MPEG-4 System and Description Languages: A Way Ahead in Audio Visual Information Representation,” *Signal Processing: Image Communication*, Special Issue on MPEG-4, Vol. 9, No. 4, May 1997, pp. 385–431.

[12] ISO/IEC JTC1/SC29/WG11 N1693, MPEG-4 Systems Verification Model Version 4.0, Bristol, April 1997.

[13] J. R. Smith and S. -F. Chang, “VisualSEEk: a fully automated content-based image query system,” *Proc. ACM Intern. Conf. Multimedia*, Boston, MA, November 1996 `http://disney.ctr.columbia.edu/SaFe`.

[14] S. W. Smoliar and H. Zhang, “Content-Based Video Indexing and Retrieval,” *IEEE Multimedia Magazine*, Summer 1994.

[15] J. R. Smith and S. -F. Chang, “Searching for Images and Videos on the World Wide Web,” *IEEE Multimedia Magazine* (to appear). `http://www.ctr.columbia.edu/webseek`.

[16] ISO/IEC JTC1/SC29/WG11 MPEG97/N1678, MPEG-7 Context and Objectives Version 3.0, Bristol, UK, April 1997.