Appendix A

Embedded Interpretation of Domain-Specific Languages for Scientific Volume Visualization

1 ViSlang: Language Design

1.1 Data Types

The ViSlang language employs a dynamic type system (duck typing) to weakly enforce type compatibilities. When a value is assigned to a variable or when a function is called with a certain set of arguments, types are automatically downcasted if possible. Otherwise a run time error is generated. Variables and functions are typed and type compatibility is enforced during run-time. This guarantees the programmer that methods are only called with arguments of compatible types. Therefore, a method does not need to check types but can always assume that correct types are provided. The special type *var* can be used when run time inference of the actual type is desired.

The well known integral types *boolean*, *string*, *integer*, and *float* are supported. Parameters of visualization algorithms are often semantically constrained to a certain range. Therefore, the types *integer* and *float* are implemented as constrained types, and these constraints are enforced at run-time. The range of variables of these types can be specified explicitly. An assignment to a constrained variable outside its range will result in a run-time error presented to the user. This ensures not only type safety but also range safety for parameters that are semantically bound to a certain range.

The result of algorithms is often a collection of variables. ViSlang supports lists that are collections of elements of arbitrary types. By returning a list, an algorithm can effectively return multiple results. ViSlang offers built-in types that are specifically targeting volume visualization algorithms. Among them are the types *volume*, *voxel*, *vset*, *vlabel*, and *image*. A *vset* is a volume of booleans (each voxel being true or false). A *vlabel* is a volume where each voxel gets a numeric label. ViSlang natively supports these concepts to accommodate the frequent use of binary masks and labeling in algorithms. Internally, the *vset* and *vlabel* data structures are resources that are stored using memory management on the graphics card. This specialization of the more general (unmanaged) *volume* type leads to an efficient storage layout of this common concept, and the ability to quickly share these resources among different software modules.

In an interpreted language environment, it is important to allow quick exploration of algorithms and objects. Reflection can be used to interactively query objects for types, variables, types of variables, functions and signature of functions. Since scientists are usually not fluent in object oriented programming, ViSlang does not allow the definition of objects at run-time. Scoping is provided by inbuilt objects and for extensions. This is important to facilitate different name spaces and to reduce ambiguities between function and variable names. Finally, ViSlang supports *slang*s, a specialization of objects, that offer extended functionality. Slangs and objects in general are registered with the current run-time environment of the ViSlang language.

1.2 Objects and Slangs

An object typically encapsulates a set of variables and functions. Each object has its own name-space and can register variables and functions at compile- and run-time. A function call as well as the access to variables of an object are done with the typical 'Object.Identifier' syntax. A slang additionally offers a programming interface to the user via the ViSlang language. A parser and execution interface are implemented that are called by the main language at run-time. The slang *VolumeGenerator* might for example implement a DSL that generates 3D volumes sampled from user provided functions. Listing 1 shows an example of the *VolumeGenerator* slang. Here it is used to sample the Marschner-Lobb test signal on a 40x40x40 volumetric grid.

```
1 //generating a volume
2 volume v;
3 using VolumeGenerator:
4   x=[-1:1], y=[-1:1], z=[-1:1]
5   v[40,40,40](x,y,z) = 0.4*(1-sin(pi*z*0.5)+
6   0.25*(1+cos(12*pi*cos(0.5*pi*sqrt(x^2+y^2)))))
```

Listing 1: Calling the *VolumeGenerator* slang's DSL interface with the *using* keyword to generate a 3D dataset.

The programming interface of slangs goes beyond the capabilities of most user interface elements. It allows the user to specify certain aspects of the algorithm in a domain-specific language that is tailored to the domain of the algorithm. To allow different languages for different domains it is not required for a slang to follow any syntactic rules of the main language. The ViSlang parser will forward parts of the program that are marked with the *using* keyword to the specified slang. The slang's syntax is completely independent from the syntax of the main language, with the one exception that it must not (and cannot) re-purpose the keyword *using*. With this technique it is possible to offer a lot of flexibility for the implementation of slangs, hence making the overall language more expressive.

1.3 Control Flow

The language supports a small set of instructions including variable declaration and assignment, function declaration and calls, conditionals, loops, as well as boolean and arithmetic expressions. Additionally, it supports syntactic constructs that are specialized to allow the seamless integration into existing user interfaces, and the combination of slangs in novel ways.

Linking: The linking (and unlinking) instruction allows to couple (and if desired decouple) two variables. In addition to an assignment, the linking is persistent and updates one variable whenever the value of the other variable changes (and vice versa). For instance, Listing 2 shows how two variables are linked and unlinked again. The assignment on line 4 will cause to also assign the new

```
1 integer a = 0;  //decl. and assign. of a
2 integer b = 0;  //decl. and assign. of b
3 a <=> b;        //linking of a and b
4 b = 1;          //assignment of 1 to b
5 a >=< b;        //unlinking of a and b
```

Listing 2: Linking of two variables: After the assignment *b=1* the variable *a* will automatically be updated to the same value.

value of *b* to variable *a*. This construct is especially useful to link variables of different objects that shall update each other.

Triggers: A trigger is assigned to a variable to execute a function every time the value of the variable changes. This is useful to trigger execution of functions or update of slangs that depend on values of

other variables. If a parameter of object A depends on the parameter of object B, a trigger is assigned to a function call that updates the parameter of object A and if necessary converts the parameters first.

A simple example that allows the user to specify the behavior of the program is shown in Listing 3. Every time the user clicks on the image the intersection of the viewing ray with the bounding box of the volume is computed and used as a position for the axis aligned slicing plane.

An object *Intersector* implements an algorithm that takes 2D image coordinates *imagePosition* as input and calculates the intersection point between a ray and the bounding box of a given volume. The intersection is stored as 3D coordinates in a variable called *volumePosition*, where each co-ordinate is between zero and the dimension of the volume. Another object *Mouse* with variable *position* stores the position of the mouse cursor on the image plane. After the initial setup that ensures that the *Intersector* and *Slicer* objects use the same volume and camera settings, the user can write a program that controls the position of the slicing plane with the coordinates of the intersection of the ray with the bounding box of the volume. In the example of Listing 3, it is (for the sake of simplic-

```
1  //function declaration
2  void setSlicePos(integer u, integer v){
3    //call intersection computation
4    list pos = Intersector.intersect(u, v);
5    //convert parameter range
6    float w = Slicer.volume.getWidth();
7    Slicer.position = pos.get(0) / w;
8  }
9  //assign trigger
10 Mouse.clickPosition ->
11 setSlicePos(Mouse.getX(), Mouse.getY());
```

Listing 3: Specifying the behavior of the program by triggering function calls.

ity) assumed that the range of the *Mouse.getX()*, and *Mouse.getY()* fit the range of the arguments of the function *Intersector.intersect*. If this is not the case the range could be converted with a similar setup. Lines 2 to 8 declare a function that calculates the conversion between the different ranges of the variables and assigns the result to the axis aligned slice position. Line 10 assigns a trigger to the variable *Mouse.clickPosition* that executes the function *setSlicePos*.

Display Functionality: In ViSlang, each valid statement ends with either a block of code (enclosed by curly brackets) or with one of the terminal symbols ':' or ';'. The *semicolon* symbol terminates the statement. The *colon* symbol triggers a call to all registered display functions with compatible signature. The type of the result of a statement is matched against all display functions. For instance the result of an assignment is the assigned value. If called with the *colon* symbol at the end of the statement the ViSlang system tries to display the result. If the value is of type *integer* an inbuilt function will be called to display the integer number. If the value is of type volume the ViSlang system finds all objects that have a method with a matching signature that can display volume objects. If there are more objects with a matching signature (for instance a volume rendering and a slicing algorithm) the user can decide which objects are enabled for display. This implicit interface mechanism is referred to as duck typing and extensively used by modern programming languages like the *go* language.

2 Grammar of Presented Slangs

2.1 Volume Predicate Slang

We give a loose description of the grammar of predicates in Listing 4:

```
predicate pred-name [ voxel-declaration ]
  (arg-list)  { statement-list }
```

Listing 4: Volume predicate grammar

where the arg-list is a list of user specified arguments, and the statement-list is a list of statements including declarations, assignments, *if* statements, predicate calls, function calls, for loops, and return statements. The voxel-declaration is specified as in Listing 5:

```
voxel voxel-name in volume-name
```

Listing 5: voxel-declaration grammar

Inside the statement-list variables from the user-specified arguments and the voxel-declaration can be used. The voxel-declaration is syntactically separated from the arg-list to emphasize its different semantics. The predicate is evaluated in parallel for each voxel of the volume defined in the voxel-declaration. The arguments of the arg-list are constant for all voxels during execution. The voxel defined in the voxel-declaration is the current voxel during evaluation of the predicate. Therefore, the member variables (x, y, z, and *value*) can be used in the statement-list and depend on the current voxel's values. Listing 6 shows a simple example of a user-defined predicate:

```
1  predicate valueAbove[voxel vox in v](float x)
2  { return vox.value > x; }
```

Listing 6: Volume predicate example

When the user defines such a predicate, the slang registers a new function with the signature *vset valueAbove(float)*. This function can subsequently be called by the user and is evaluated over volume v in the volume predicate slang. The evaluation is done on the GPU. This allows the user to specify logical predicates that are efficiently evaluated on the graphics hardware. There is no need for the user to deal with low-level interfaces, and the semantics of parallel execution.

2.2 Vlabel Visualization Slang

The *vlabel* visualization slang implements a declarative language that takes data of type *volume* and a corresponding *vlabel* and performs ray-casting. It allows the user to specify different visualization styles and to assign particular labels to these styles. A style is specified using a very concise syntax controlling the weights of different colors, and other visual properties.

```
1  focus{
2    color:50%(1,1,0,1)50%label,
3    bordercolor:75%(0,0,0,1),
4    shading:100%}
5  context{
6    color:100%(1,1,1,1),
7    shading:25%}
```

Listing 7: Example of declaring two visualization styles with the vlabel visualization slang.

For instance, Listing 7 shows the declaration of the two visualization styles *focus* and *context*. The *focus* style is defined to use 50% of the user-defined color (yellow in this case) and 50% of a randomly generated color using the label id. The border color is set to black and applied with 75% opacity. Shading is applied with 100% opacity. The vlabel visualization slang parses these descriptions and translates them to weights and colors that are made available to the volume rendering algorithm.

2.3 Map-Reduce Slang

The grammar consists of the *mapping function declaration* and *reduction function call*. The grammar of the *mapping function declaration* accepts a subset of the main language's function declaration. Statements like linking and unlinking as well as triggers are not permitted in the mapping function's body. In Listing 8, a short description of the *reduction function call* grammar is shown.

```
reduction-operation voxel-declaration-list
function-identifier (arg-list)
```

Listing 8: Reduction call grammar

The *reduction-operation* is one of the keywords *sum*, *min*, *max* or *mul*. The *voxel-declaration-list* is a comma-separated list of *voxel-declaration*s as defined for the volume predicate slang. The *function-identifier* is the name of a mapping function, and the *arg-list* is the list of arguments for the mapping function.

```
float3 coordinate(voxel v, integer id, integer m
    ){
  float3 pos=m;
  if(v.value==id){
    pos.x=v.x; pos.y=v.y; pos.z=v.z;
  }
  return pos;
}

p1 = min[voxel v in vol] coordinate(v, 1, 4096);
p2 = max[voxel v in vol] coordinate(v, 1, 0);
```

Listing 9: Example of defining a mapping function and executing two reduction operations.

With the *Map-Reduce* slang the user can quickly implement reduction operations that are executed on the GPU. Listing 9 shows an example of a mapping function that returns the position of a voxel if the voxel's value equals *id*. Using the *min* and *max* operation, this mapping function is used to calculate the bounding box of the region with *id*=1 in this example.

REFERENCES