Python for Scene and Model Description for Computer Graphics

Mark Tigges[*] Brian Wyvill
mtigges@cpsc.ucalgary.ca blob@cpsc.ucalgary.ca
Department of Computing Science, University of Calgary

November 24, 1999

Abstract

Python is a general purpose language for use in many applications. In this work we have used Python as the input scene and model description language for a suite of rendering and modeling tools developed at the University of Calgary. Little languages have been used extensively in this arena. Most notably Lua [7, 5], and the Renderman Languages [16].

Python has been used at the University of Calgary for the construction of very complex models. We describe a set of software tools for interactive rendering of models and the description of scenes built up from these models. We also describe a parallel rendering system all based on Python. Parallel renderings are computed on a 30 node distributed memory computer, a discussion of using MPI [6] in concert with Python is given.

1 Introduction

One of the major problems of building complex digital scenes for the rendering of synthetic images, is that model descriptions often come from a wide variety of different sources and output needs to be customized to different rendering software depending on user requirements. Thus it is necessary to handle a wide variety of input and output formats. These formats result from complex GUI applications for both modeling and animation, input formats for rendering packages, and special purpose languages for physically correct scene description [18]. There are many aspects to consider. One of these aspects should be the programmibility of the description format. This essentially implies a format which generates models. Flexibility can mainly be achieved through parameterization of all aspects of the model. Some scene description systems support this very well and others do not.

Research in the GraphicsJungle laboratory [8] is partly focused on shape description with implicit surfaces. An implicit surface is a point set which satisfies some non-trivial implicit function. The typical implicit surface uses a function similar to function 1.

$$F(x,y,z) = \sum_{i=1}^{n} c_i f(s_i(x,y,z)) \quad (1)$$

$$f(r) = \begin{cases} 1 & r < 0 \\ \left(1 - \frac{4}{9}r^2\right)\left(1 - r^2\right)^2 & 0 \le r \le 1 \\ 0 & r > 1 \end{cases} \quad (2)$$

Where s_i returns the distance to a skeleton (point, line, circle *etc*.), f is a (usually sigmoidal) field function which when summed with others produces soft blending between the influences of sibling skeletons, and c_i is a skeleton dependent scalar coefficient. This definition implies a set of geometric elements (point, line, circle *etc*.), which form the user defined skeleton. For any point in space, the function F, returns a scalar value. This value is a measure relative to the weighting supplied by f of the influence of the skeleton for the given point. To define a surface using this definition a particular value can be chosen in the range of F, called the iso-value. When F is applied to the volume of space surrounding the skeleton a set of iso-valued points is found which lie in the surface of interest (see [4] for details of implicit surface models).

1.1 Problem Statement

The problem being addressed by this paper is how to specify the primitive user defined skeletons. Of course this problem is not unique to implicit surfaces. The same problem exists when modeling with any other paradigm (CSG[1], subdivision[2] *etc*). The varied solutions for the specification of models break down as follows. First, a GUI program can be written. Such a program

[*] Currently at Mainframe Entertainment Inc. mtigges@reboot.com

[1] *CSG*: Constructive solid geometry, the construction of complex volumes through boolean operations of more simpler volumes.

[2] Inferrance of a complex surface by subdivision and refinement of a complex of polygons

allows the interactive construction of a model and a scene graph. Second, a declarative text file can be used to specify in serial the structure of the model. Third, a full programming language can be used. This in fact exists by default in our system, since the data structures and algorithms are implemented in C^{++}. However the raw implementation in a compiled language is likely not as flexible as it could be, nor is the compile/link/run cycle advantageous for efficient development of a geometric model.

The GUI solution suffers from flexibility. A user is constrained to working within the GUI which is constrictive for an expert user and is often more applicable in a production environment. In contrast we work within a research environment where experimentation at a variety of levels within the system requires a programattic interface such as provided by Python. The vast majority of contemporary computer users balk at the suggestion that any meaningful and efficient work can be done through any non graphical user interface.

The purpose of a GUI is to handhold. A brand new computer user will be able to compose and print a document using a GUI word processor. If the same user begins with LaTeX nothing will be done without serious consultation with documentation. For modeling and scene description the same holds true however it becomes extremely difficult to produce a GUI which is immediately comfortable. For graphics one need only look at the GUI*s* that exist. It is very difficult to construct a comprehensive GUI whereas it is easy to produce parameteric or procedural models using application specific tools. With the release to the public of Blender [11], a high end modeling, animation and rendering package, eveybody has the technical capability to create first rate images and animations. The learning curve of the GUI for this software (and its competetitors) is extremely steep. The nature of the problem of specifiying three dimensional models and changes to them over time is such an inherently complex problem that creating a simple yet comprehense GUI has proven formidable. It is not at all clear though that a programmatic paradigm is inherently better, what is clear though is that for users who are doing graduate level research on implicit surfaces it is far better.

The declarative text file format fares a bit better than the GUI. It is the most common format, a prime example is the input format for Rayshade [9] or POV-Ray [14]. The obvious problem with this method is that it is difficult for novice users. Evidence to support this is the existence of a large number of tools (of varying usefulness) which provide a GUI frontend to the specification file format. In our case this isn't that large a consideration since as noted above it is mostly graduate students and researchers who use the system. Parameteric models can be defined in such a declarative text file by using a preprocessor like M4 [12] or the C preprocessor.

1.2 Language Choice

Besides Python, the choices for this project included Lua [7], Scheme, Java and Tcl. Lua was an obvious choice for the simple reason that it is small, fast and originally developed by computer graphics researchers. Two choices not enumerated were VB and Perl, these were trivially rejected due to either not being available on Unix, or seemingly too steep a learning curve for people whose only interest in the chosen language is to use the modeling suite. Scheme would have been a good choice due to the large amount of documentation available for it and its ease of expression construction. Java also supports a large amount of documentation and is relatively easy to learn, the difficulty in embedding it in an application ultimately caused its dismissal. Tcl was rejected due to the expressiveness of the language not being suitable for the hierarchical structures we wanted to be able to define.

This left the final choice being between Lua and Python. Ultimately the choice was made in Python's favour because it seemed that everyone could program with Python [17]. The lack of Lua documentation made its learning curve too steep for use only as a modeling tool.

1.3 Outline

In order to use any programming language for the task of model and scene description, several pieces of software are required. For our case of scene description of implicit surface models the most obvious piece was the Python extension modules which expose the C^{++} API for the implicit surface and scene graph data structures to accessible types. Since the nature of the development of a geometric model is evolutionary, it is extremely likely that the model will need to be refined many times. A prototyping tool for interactive viewing of a model was also developed and necessarily embeds the language of choice in order to execute the model code. The last piece of software required was a modified interpreter which could orchestrate inter-process communication across a network of workstations. This requirement was so that we could take advantage of the high performance computing cluster system available at our University. Both Python and Lua satisfied all of these requirements. The availability of "Programming Python" [13] influenced our choice of Python considerably.

The remainder of this paper is organized in the following manner. The extension modules are discussed in section 2. The interactive program `pybv` is discussed in section 3. A description of the use of Python with MPI is given in section 4. Finally, conclusions and future work are discussed in section 5.

2 Extension Modules

This section discusses the extension modules that provide the framework for the construction of Python objects which encapsulate implicit surfaces and the scene graphs used to display them. Both the implicit surface software and the scene graph software are C^{++} hierarchies. Therefore the modules which encapsulate them must provide the ability to create instances and the ability to manipulate the public interface of the instances.

There are two possibilities for wrapping these hierarchies for use in Python. The first is to use a tool like SWIG [1] to completely wrap the data structures. The second is to generate the wrapper by hand providing a completely different interface suited only to model definition. The trade off between these two is essentially one of flexibility and ease of use. The choice made for our project is the latter.

The reason for choosing a minimally exposing wrapper of the underlying API was simple. Computer graphics is a compute intensive application and contrary to the example set by the Renderman [16] shading language the systems for modeling and rendering should not be implemented in an interpreted language. If we were to use a tool like SWIG and mirror the complete modeling and rendering API's then there would be no reason for a user to not use Python for everything. This would include creating new types of implicit functions to be used inside composite implicit surface models. This is not a recommended practice since it is very expensive to create polygonal approximations and compute line intersections with these models, both of which are required for rendering images of the models. If the image synthesis includes interpreting some code the time to render an image increases unacceptably. The extension modules only expose the construction of the hierarchies to facilitate the eventual rendering of images. Anything more would be beside the point and allow Python code to execute during rendering rather than just scene construction causing interminably long rendering times.

2.1 BlobTree

Our implicit surface system implements the *BlobTree* [19]. This is a hierarchically based data structure. An example is shown in figure 1. An instance of a

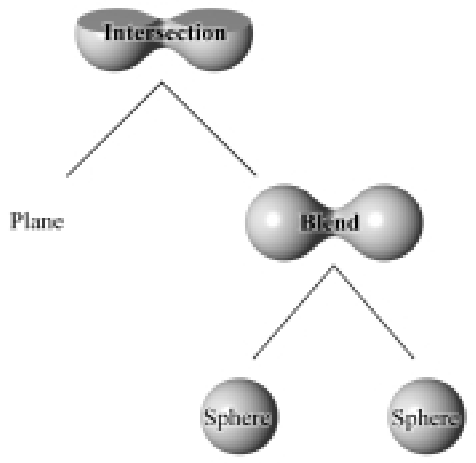

Figure 1: An example *BlobTree*.

BlobTree is an n-ary tree. The leaves of the tree form the user defined skeleton of geometric primitives. The interior nodes are either transformation operations or they are blending operations. A transformation node may be as simple as a translation, or rotation, or it may be as complex as a non-linear warping of the space. A blending operation in effect specifies what exactly F, (equation 2), looks like. In section 1, summation was used. However, some other function can also be used to implement varying degrees of blending, or in fact, in the extreme, the CSG operators: difference, intersection and union. Boolean operations which combine sub-*BlobTree* using the inferred operation producing a more complicated surface.

Exposing a minimum of the underlying API in the extension module which wraps this data structure is not very difficult. A simple procedural stack based approach is taken to define objects. For those familiar with the programmatic interface of OpenGL this is clear, for those who are not the following example illustrates the approach.

Example 1 *The Python function defining the model shown in figure 1.*

```
def half_peanut(dx,dy):
    o = pyjbt.BlobTree()
    o.intersection()
    o.blend()
    o.push()
    o.translate(-dx,0,0)
    o.point()
    o.pop()
    o.push()
    o.translate(dx,0,0)
    o.point()
```

```
        o.pop()
        o.end()
        o.translate(0,dy,0)
        o.plane()
        o.end()
        return o
```

The above example shows how the definition of an implicit surface using our system is accomplished. The extension module is named `pyjbt` and exposes the *BlobTree* data type. Interior nodes in the hierarchy are constructed through maintained levels in a stack of composite nodes (eg. blend, intersection *etc.*), using the push and pop functions. The leaf nodes are always the skeletons, in this case point and plane. This example is a parameterized model. Namely it is parameterized on how far apart the point skeletons are from the origin and at what position along the y-axis the intersection should take place.

The reason for choosing this stack based approach is mostly due to its traditional use in this role in computer graphics. However that statement does not do justice to why it is the traditional choice. The extension module provides a procedural interface to what in the C^{++} implementation is an object hierarchy. For the construction of a tree of instances of types in this hierarchy where each level of the tree defines attributes inherited by subtrees the stack based approach is the most natural. See [3] for a complete discussion on this topic.

This module has been successfully used to build very complex models. In figure 2 a rendering is given of a model which is built from well over 500 nodes. The Python code for this model is nearly 800 lines. Every aspect of the model has been parameterized. A considerable portion of the model is procedural. Rather than directly specfying the points the author implemented algorithms in Python to compute the desired effect. For instance the helical structure of the shell is computed using mathematical models described by Meinhardt *et al.* [10], the placement of the spikes are also due to procedural codes.

2.2 Scene Graph

A scene graph is a data structure which encodes the geometry and reflectance information used to render a digital image. The rendering software used at the University of Calgary is a distributed ray tracing system developed in our lab. It serves as a platform for experimentation for new ray intersection algorithms for implicit surfaces.

In order to render scenes easily one needs a very simple interface to specify the scene graph. Input formats such as seen in the POV-ray [14] and Rayshade [9] systems are not very flexible. The specification of the scene graph is given in a very serial manner. There is no break up of the graph into logical units. Python provides three levels of abstraction for components of the scene graph: modules, functions and classes. A very simple Python extension module for generating a scene graph can be used in any of these three manners. At the module level a user specifies the scene in a completely serial fashion. At the functional level the scene graph can be composed of a set of function calls, and similarly for the class level, conviently segmenting the graph into a coherent group of components, easily written and easily understood.

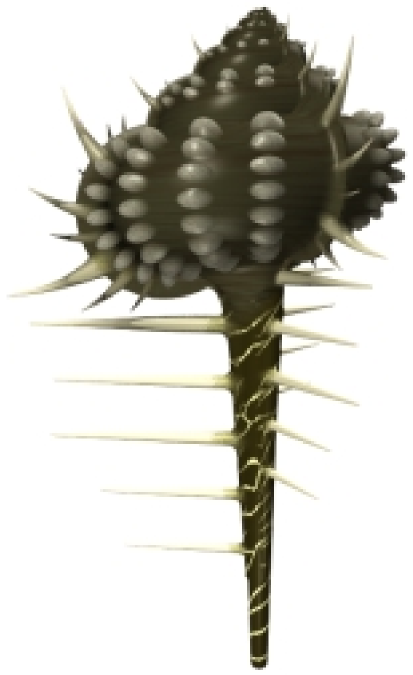
Figure 2: A procedurally defined model of the *Cabrets Murex* seashell.

The following example shows why this is so beneficial. The example shows the code needed to produce a short animation. The animation features an implicit surface constructed from two point skeletons. The two skeletons move further apart from each other with each frame. The ease of use of Python and the readability of it makes it an extremely valuable tool for this application.

Example 2 *A simple scene graph parameterized on time through the use of function-level modularization. (The sequence of images can be found at http://www.cpsc.ucalgary.ca/˜ mtigges/mpipython.html)*

```
    import pyjbt, pyrtl
```

```
def peanut(x):
   o = pyjbt.BlobTree()
   o.blend()
   o.diffuse((1,1,0))
   o.translate(-x,0,0)
   o.point()
   o.diffuse((0,0,1))
   o.translate(2*x,0,0)
   o.point()
   o.end()
   return o

def scene(frame,total_frames):
   s = pyrtl.World()
   s.lookat((0,0,4))
   s.soft(peanut(float(frame)/
      float(total_frames)*1.2))
   s.render()
   s.save('frame.%d.ppm' % frame)

import sys

total_frames = sys.argv[1]
frame = 0
while frame <= total_frames:
   pyrtl.Print('Rendering frame:%d' %
   frame)
   pyrtl.NewAlloc()
   scene(frame)
   pyrtl.KillAlloc()
   frame = frame+1
```

3 Interactive Renderer

The preceding section outlined our methods for the description of models and scene graphs. These scene graphs can only be viewed by rendering with the ray tracing library. However, using ray tracing to render an image of a model is not very efficient. In order to build a model it is more convenient to render it interactively. This can be done by finding a polgonal approximation which can be rendered on a workstation at interactive speeds. This need is filled by `pybv` [15], an interface between the rasterization systems windowed output and a Python interpreter.

The use of Python described in the previous section extends the language to provide data types and services for the description and rendering of models and scenes. In this section the software described embeds the interpreter. The input for `pybv` is a Python module. This module must adhere to a certain form. It must have a function called `blobtrees`. This function returns a list of tuples, each tuple describes one model defined in the module. This description includes a text string label, a function (which must return a *BlobTree*, the arguments to pass to the function and a set of arguments used to polygonize and view the model).

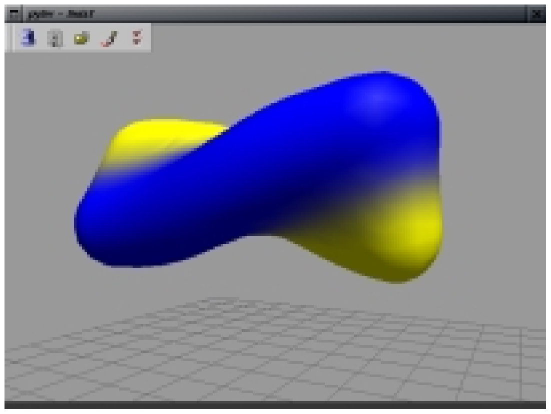

Figure 3: The `pybv` main window showing the twisted blend of two line primitives.

When `pybv` is invoked it uses `PyImport_ImportModule` to load the module definitions into the dictionary. This dictionary is then accessed through GUI elements of the application to invoke the functions described by the return value of the `blobtrees` function (see example 3). Each time one of these functions is invoked the resulting *BlobTree* is polygonized and then rasterized to the display. The cycle to create a model is an edit/reload cycle. This is in contrast to an edit/compile/run cycle that would exist if Python were not used as a glue to the specification of models and users simply written against the C^{++} interface.

Example 3 *An example* `blobtrees` *function. This function is called by the* `pybv` *program through its embedded python interpreter to query what implicit surface objects can be constructed by the module. The returned value is a list of functions which can be called to construct implicit surface models.*

```
def blobtrees():
   return [
      ('half-peanut',
      half_peanut,
      (0.6,0.1))
   ]
```

This system was used to create the model in figure 2. The complexity of this model, in that case, prohibited the viewing of the entire model at once. There were two reasons for this restriction. The Python code executed too slowly for effective interaction, but more seriously the polygonization of the model took too long for an accurate representation. The use of `pybv` and the extension modules provided an easy and effective work around. The system afforded the definition of multiple functions one for each part of the shell. The `blobtrees` function gave reference to each major part that the user wished to be able to view indepently. This allowed hierarchical abstraction of the entire model to its constituent parts, something which is quite difficult to do with a GUI program.

The net result of the combination of extending and embedding Python is a very convienient rapid prototyping environment for the development of our models. To measure its value one could do a user test in comparison with a GUI designed for the same task. Such a user test however has not been completed. Personally the author finds the cycle of text editing the source of the model with subsequent interpretation of that text by `pybv`'s embedded interpreter a very satisfying and efficient process.

4 Python with MPI

In this section the embedding of Python into a simplified interpreter is discussed which allows Python to be used as the input description language in a distributed computing message passing environment. The University of Calgary is equipped with a 30 node cluster of computers. Since there are few computing applications which scale to distributed parallel computing as efficiently as ray tracing our graphics group was extremely excited at the availability of this resource.

To take advantage of this hardware system we needed an interface between the resident message passing system on the cluster and our python extension modules. The message passing system resident on the cluster is MPI [6] a very flexible system in very wide use. Our extension modules had to be modified to distribute portions of the scene to render. However in order to use the extension modules which now used MPI the Python interpreter needed to be aware of MPI for initialization and cleanup of the MPI state machine.

When a render job is submitted to the cluster each machine builds its own data structure for the scene graph. This infers that each machine runs the Python interpreter locally. To facilitate this the Python interpreter has been embedded in a program called `mpipython`. This program is assigned the following tasks:

- Initialize the Python interpreter.
- Initialize the MPI library.
- Communicate the Python scene graph input and the `PYTHONPATH` variable.
- Execute the script.

The last item is due to the fact that the slave children do not receive the same run-time environment as the master. The difference between the above enumeration and previous work [2] is that there is no parallel execution of the Python code. Each node in the job executes the script in serial. The reason for this is that the time to execute any one line of the script is trivial compared to the one line which causes the scene to be rendered.

The scene graph extension module implements the parallel rendering code. The scene graph exposes a function called `prender`. This function, instead of invoking the rendering code from the ray-tracing library directly, orchestrates the rendering through each of the child processes. The master process keeps track of which scan lines have been rendered and sends to each child in turn a line to render. When a child has finished a line it sends the information back to the master requesting a new one. In addition to responding to the completed lines from the children the master is also rendering its own lines.

The modeler writing the scene description language in Python should not be aware that the scene will be interpreted and rendered in parallel. To this end the extension modules do not expose the MPI interface to the modeling environment. The extension modules render a scene; if a parallel environment exists then the image synthesis is distributed across the cluster. The extended Python interpreter ensures that the MPI environment is initialized and is able to facilitate the required communication. The choice of MPI is then reduced to the typical reasons that one would choose MPI over an alternative, say PVM.

5 Conclusions and Future Work

A set of Python tools developed in the graphics lab at the University of Calgary has been presented. The tools use Python in multiple ways.

Python is extended with the atomic data types around which our research interests centre. This allows us to use Python for the description of instances of this data giving us the benefit of the natural expressiveness and readability of Python. This provides us with a very efficient and useable system.

Python is embedded in a GUI program which allows the code describing our models to be executed and a derived tesselation displayed. This provides an edit/interpret/view cycle which makes model design relatively painless. The alternative is a GUI program which through a point and click interface allows the construction of a model. Python provides a clear syntax, which for certain types of models, particularly models with procedural elements (for example geometric spirals) makes modeling easier than a full GUI directed interface.

Python is again embedded in order to run the interpreter coherently across nodes of a distributed memory computer. This embedding allows the use of MPI for the communication of data across the child nodes involved in the rendering of an image.It has been found in our experience that Python and its associated C API is very flexible in supporting the variety of design issues in modeling.

The use of Python as the glue between our modeling and rendering systems has provided us the benefit of procedural animation. The Python code allows the expression of a scene parameterically over time allowing for very flexible descriptions of animations without any extension of the underlying languages in their native C^{++}language.

Future work could centre on providing a complete wrapper of the API using SWIG [1]. It has been noted before that this is possibly undesireable due to the fact that Python should not be used directly in rendering code due to speed of execution issues. However, it should be noted that if the user is careful enough, the full flexibility of exposing the complete C^{++} hierarchy can be used without risk.

Acknowledgements

The authors are indebted to the reviewers for careful reading and excellent suggestions for improvement. The authors would like to thank the adminstrators of the MACI Cluster for High Performance Computing. Also deserving of thanks is Callum Galbraith for the sweat and tears put into the construction of the Cabrets Murex model. This work was partially supported by grants from the Natural Sciences and Engineering Research Council of Canada.

References

[1] David M. Beazley. Simplified wrapper and interface generator. http://www.swig.org.

[2] David M. Beazley and Peter S. Lomdahl. Feeding a large scale physics application to python. In *Sixth International Python Conference*, 1997. http://www.python.org/workshops/1997-10/proceedings/beazley.html.

[3] Jim Blinn. Nested transformations and blobby man. *IEEE Computer Graphics And Applications*, 7, 1987.

[4] Jules Bloomenthal, editor. *Introduction to Implicit Surfaces*. Morgan Kaufmann, July 1997.

[5] Waldemar Celes, Roberto Ierusalimschy, and Luiz Henrique de Figueiredo. Lua: the programming language. http://www.tecgraf.puc-rio.br/lua/.

[6] The MPI Consortium. Message passing interface. http://www-unix.mcs.anl.gov/mpi/.

[7] Luiz Henrique de Figueredo, Roberto Ierusalimschy, and Waldemar Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal*, pages 26–33, December 1996.

[8] University of Calgary GraphicsJungle Lab. Jungle software projects. http://www.cpsc.ucalgary.ca/Redirect/GJ/software/jspdoc/index.html.

[9] Craig Kolb, David P. Dobkin, and David C. Hoffman. Rayshade rendering software. http://www-graphics.stanford.edu/˜cek/rayshade/.

[10] Hans Meinhardt, Przemyslaw Prusinkiewicz, and Deborah Fowler. *The Algorithmic Beauty of Seashells*. Springer Verlag, 1998.

[11] NaN. Blender. http://www.blender.nl/.

[12] GNU project. M4: A powerful macro processor. http://www.gnu.org.

[13] Programming Python. *Mark Lutz*. O'Reilly, 1997.

[14] The POV-Ray Team. Persistence of vision ray tracer. http://www.povray.org/.

[15] Mark Tigges. pybv. http://www.cpsc.ucalgary.ca/˜mtigges/pybv.

[16] Steve Upstill. *The Renderman Companion : A Programmer's Guide to Realistic Computer Graphics*. Addison Wesley, 1989.

[17] Guido van Rossum. Computer programming for everybody. Technical Report 90120-1a, Corporation for National Research Initiatives, 1999.

[18] Greg Ward, Rob Shakespeare, Ian Ashdowne, and Holly Rushmeier. Materials and geometry format. http://radsite.lbl.gov/mgf/.

[19] Brian Wyvill, Eric Galin, and Andrew Guy. Extending the csg tree - warping, blending and boolean operations in an implicit surface modeling system. In *Proceedings of the Third Eurographics Workshop on Implicit Surfaces*, June 1998.