Interactive Shader Development Using Python Scripts

Florian Mannuß, André Hinkenjann
Computer Graphics Lab
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Germany

Abstract

With software based global illumination algorithms, shaders can become very complex. We present the interactive shader development subsystem of our ray tracing framework. Using this technique, shaders can be developed or modified at run time without the need to recompile or restart the system. Python has been chosen as the scripting language.

1 Introduction

Shaders describe the appearance of a surface. The creation of an efficient and quality shader requires some effort and is often done iterative. Sometimes it is even important to fine tune shaders in the display environment in which the shader is utilized. Developing a shader in a compiled language, like C++, leads to recompiling and restaring the system after changing the shader. This is a time consuming process. Being able to change the shader online leads to significant time savings.

Scripting languages interpreted at run time can be used to do interactive shader development. All shader changes are visible the next frame and the workflow is not interrupted for recompiling and restarting the system. Continuous working with the model is achieved.

Our ray tracing framework, written in C++, is designed to support shader-scripts and it provides an interactive interface to allow changes at run time. We decided to use Python [4] as the scripting language mainly because of its object orientation and clear syntax.

Other systems that support scripting are Renderman [3] for off-line global illumination calculations and CG [2] for local illumination on the graphics hardware.

2 Design

In order to be able to write a Python shader all relevant C++ data types, like vectors, have to be accessible by Python code in order to interact with the calling system. On the other hand the Python script has to be callable from the global illumination framework. As a result, Python has to be *extended* with the C++ data types and Python has to be *embedded* into C++ as sketched in figure 1.

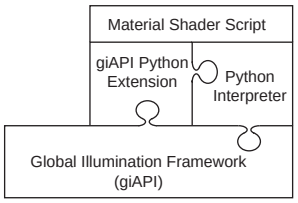

Figure 1. Extending and embedding Python into the ray tracing framework

Furthermore, it has to be transparent to the renderer which shader type is in use, since we now have two possibilities to implement a shader. Firstly, in C++ by deriving from the shader base class and secondly by a Python script.

For *Embedding* Python into the framework (c.f. right side of figure 1) the Python C-API [5] has been used. Only a few commands are needed to embed the interpreter.

Extending Python is done by writing wrappers. They call the C++ methods and attributes, validate and convert parameters and convert the return value to Python data types. In order to assist the extending process SWIG [1] is used to create the Python specific wrapper functions that are used to access the C++ functionality from Python. Since SWIG supports different scripting languages, support for those languages can be added, if needed.

Keeping C++ and Python shaders transparent to the renderer is done by introducing a *proxy class* derived from the shader base class to store the Python shader. This proxy class is responsible to load the Python script and to call all methods. Consequently, every Python shader has to have a predefined appearance. Every shader is represented through one class and must have a predefined set of methods to call the shader as seen in figure 2. Setting all parameters is done

through a method in the proxy class. The different parameters are identified by their names, like “diffuseColour” and all parameters that can be set must be derived from a data class hierarchy.

```
class sampleMaterial:
  diffuseColour = giAPI.colour(0.8, 0.8, 0.8)

  def calcBRDF( self, inVec, normal, outVec, retColour ):
    retColour = self.diffuseColour

  def calcBRDFPart( self, inVec, normal, outVec, sMatComp, retColour ):
    retColour.set( 1.0, 1.0, 1.0 )

  def sampleBRDF( self, inVec, normal, sMatComp, outVec, prob ):
    outVec.set( 1.0, 1.0, 1.0 )
    prob = 1.0
    sMatComponent = "diffuse"
```

Figure 2. Interface functions that have to be implemented in the shader script class.

Run time interaction is provided by a *scriptInterpreter* object that is listensing at a predefined tcp-port for incoming commands. Commands are always Python commands and they are passed to the interpreter using the Python C-API. In order to access the shader proxy class and to set attributes of a Python shader, Python has to be extended with the shader proxy class, the data class hierarchy and a repository class that stores all material objects. This repository acts as an interface to the C++ material objects and has to be implemented as a *singleton*. This ensures that only one repository is created and this repository can be accessed within Python.

3 Results

Using scripts for interactive shader development is convenient, since recompiling and restarting the system drops out. However, the disadvantage of using a scripting language is the speed loss that makes script shaders well suited for rapid prototyping, but less suited for time critical interactive applications. The reason for this speed loss is the time consuming process for calling methods of extended class objects in Python and the fact that the shader is called for nearly every ray shot. The bottleneck is the parameter validation and conversion that has to be carried out for every method call. Dropping the validation step to gain higher performance is not possible. Table 1 shows the rendering times needed to calculate figure 3 using a IBM G4 CPU running with 1GHz. The overall rendering speed using the C++ shader is about twice as high as with the Python shader. Since only the shader has been changed and everything else kept unchanged the speed loss is the result of using the Python shader. This can be clearly seen when splitting up the overall rendering time in the time needed for shader calculation and all other calculations.

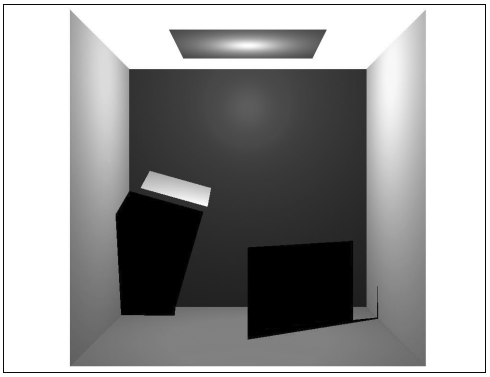

Figure 3. Test scene (800x600 pixels) to compare the rendering speed of C++ and Python shaders

-	C++	Python
Overall Time (s)	6.4	13.9
Shader Time (s)	0.26	7.40
left over Time (s)	6.14	6.5

Table 1. The Python shader is about thirty times slower than the C++ version. However, the total calculation time is only doubled.

4 Conclusion and Future Work

Despite the speed loss, productivity is increased due to the ability of continuous and uninterrupted working with the model when using interactive shader development.

One solution to overcome the speed problem is to enable automatic translation of the Python shader to a C++ shader after the development phase. The generated C++ shader has to be compiled to a shared object and loaded into the running system, replacing the Python shader. This is a topic for future work.

References

[1] D. M. Beazley. *SWIG Users Manual*, June 1997.

[2] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[3] S. Upstill. *The RenderMan Companion : A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Professional, 1990.

[4] G. van Rossum. Python homepage. www.python.org, 2005.

[5] G. van Rossum. *Python/C API Reference Manual*, March 2005.