Interactive shader development for ray tracing using Python scripts

Florian Mannuß
Bonn-Rhein-Sieg University of Applied Sciences
Computer Graphics Laboratory
florian.mannuss; andre.hinkenjann@fh-brs.de

Andre Hinkenjann
Bonn-Rhein-Sieg University of Applied Sciences
Computer Graphics Laboratory
florian.mannuss; andre.hinkenjann@fh-brs.de

Abstract

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 scripting language.

1 Introduction

Shaders describe how a surface of an object has to be coloured. In ray tracing systems every object can use a different shader and new shaders are continuously developed. Developing a shader in a compiled language, like C++, leads to recompiling and restarting the system after changing the shader. This is a time consuming process, especially in design evaluation stages where shader fine tuning is needed, while working with the model in the desired interactive viewing environment, like a powerwall. Being able to do changes online is of advantage. 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 need not to be interrupted for recompiling and restarting the system. Continuous and uninterrupted working with the model is now possible.

Our ray tracing framework, written in C++, is designed to support scripts as shaders and it provides an interactive interface to allow changes at run time. Due to its popularity we decided to use Python [3] as scripting language.

2 Design

In order to be able to write a Python shader all relevant C++ data types, like vectors, have to be accessible through Python in order to interact with the calling system. On the other side the Python script has to be callable within C++. 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. A shader script makes use of the giAPI-Extension in order to get the input data, do the calculations and pass back the return values.

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

Embedding Python into the framework, this is the right side of figure 1, the Python C-API [4] has been used. Only a few commands are needed to embed the interpreter.

Extending Python is done by writing wrappers to call the C++ methods and attributes, does parameter validation and conversion and converting 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.

Keeping C++ and Python shader transparent to the renderer is done by introducing a proxy class [2] derived from the shader base class to store the Python shader. This proxy class is responsible to load the Python script and 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 one 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 through a scriptInterpreter object that is listening 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 interface to the C++ material objects and has to be implemented as singleton [2]. 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 shader good for rapid prototyping, but not for usage. 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 shot ray. The bottleneck is the parameter validation and conversion that has to be carried out for every method call. Table 1 shows the rendering times needed to calculate figure 3 using a IBM G4 CPU running with 1GHz. The C++ shader needs only half of the time as in case of the Python shader. Since only the shader has been changed and everything else kept unchanged the speed loss in the Python shader case is only a 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.

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

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Time (s)</td>
<td>6.4</td>
<td>13.9</td>
</tr>
<tr>
<td>Shader Time (s)</td>
<td>0.26</td>
<td>7.40</td>
</tr>
<tr>
<td>left over Time (s)</td>
<td>6.14</td>
<td>6.5</td>
</tr>
</tbody>
</table>

4 Conclusion

Despite the speed loss, subjective comfort due to the ability of continuous and uninterrupted working with the model is gained 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. This generated C++ shader has to be compiled to a shared object and loaded to the running system while replacing the Python shader.

References