PyFX – An active effect framework

Calle Lejdfors and Lennart Ohlsson
Dept. of Computer Science
Lund University, Sweden
{calle.lejdfors,lennart.ohlsson}@cs.lth.se

Abstract

The programmability of modern graphics processing units (GPUs) provide great flexibility for creating a wide range of advanced effects for interactive graphics. Developing such effects requires writing not only shader code to be executed by the GPU but also supporting code in the application where the effect is to be used. This support code creates dependencies between effects and the applications that use them, making it harder to evolve applications and to reuse effects. Existing effect frameworks, such as DirectX Effects and CgFX, can only provide partial encapsulation because they consider effects as passive data structures. In this paper we present an effect framework written in an ordinary scripting language where effects are active entities. This makes it possible to completely encapsulate both shaders and support code thereby minimizing the dependencies to the application.

1 Introduction

The availability of programmable graphics processors has made procedural effects a key ingredient in real-time graphics productions. Where content creation previously was mainly the combination of a wide range of different kinds of artwork such as geometric models, textures, and motion data, it now also has to include algorithmic development. Writing the shader code to be executed on the graphics processors is something which traditionally is not part of an artist's skill set. Instead this new development model requires a closer relationship between artists and shader programmers. Previously programmers of interactive graphics applications were primarily concentrated with loading and displaying content created by the artists in an efficient and correct manner, a task which is handled fairly independent of the actual content. But with programmable graphics processors the roles of artists and programmers become more intertwined. When the artist conceives of a visual effect it is the programmers job to supply shader programs and the necessary modifications to the application for achieving that effect. But once written, the shader program typically requires actual textures and parameter values and it is the artists job to supply that.

For efficient collaboration it is important, to both artists and programmers, that the graphical effect is a well-defined entity. It should include all relevant resources and functionality, both shader code and application support, required for correct operation. This need for encapsulation is the motivation behind technologies such as the DirectX Effects by Microsoft and CgFX by NVIDIA. In these frameworks the notion of an *effect* is used as the key unit of abstraction. But although these technologies provide a number of features which improve the handling of effects they still require a substantial amount of application support. All but the most trivial effects have dependencies in the application that use them.

The effect framework presented in this paper aims to provide *complete encapsulation* of effects in the sense that specific support code avoided and parameter passing is made with the most unobtrusive mechanism possible. We have implemented a prototype which uses Python both for the implementation of the framework and to express the effects themselves. This paper is focused on the implementation of the framework and its application interface, whereas the benefits of writing effects in Python is described in more detail elsewhere [Lejdfors and Ohlsson 2004].

1.1 Related work

The focus on complete effects is different from most other approaches. Most real-time shading language research has been focused on mapping high-level shading languages to real-time shading hardware. Research was initiated by Cook [Cook 1984] with the introduction of shade trees, which spawned a number of shading languages such as RenderMan [Hanrahan and Lawson 1990] or Perlins image synthesizer [Perlin 1985]. These languages were originally used for off-line shading but with advances in hardware Peercy et al showed that it was possible to execute RenderMan shaders on an extended OpenGL 1.2 platform by viewing the graphics hardware as a SIMD pixel processor [Peercy et al. 2000]. Olano et al presented an alternative approach with the pfman language [Olano and Lastra 1998] for the Pixelflow rendering system [Molnar et al. 1992], a flexible platform based on image composition which, unfortunately, bears little resemblance to the GPUs of today.

The computational model of separating per-vertex and per-pixel computations was introduced by Proudfoot et al [Proudfoot et al. 2001] which allowed the to efficiently map shader programs to hardware. This separation is used explicitly the in real-time shading languages used in the industry today: Cg by NVIDIA [Mark et al. 2003], HLSL by Microsoft [Gray 2003] and OpenGL shading language [3D 2002] introduced with OpenGL 2.0. This is also the case with the Sh language [McCool et al. 2002][McCool et al. 2004]; a shading language embedded in C++ which provides a number of powerful high-level features for shader construction. Another embedded shading language is Vertigo [Elliott 2004] which uses the purely functional language Haskell as a host language to provide a clean model for writing shaders for generative geometry.

All these efforts have focused on various aspects of shader programming but the writing of effects containing multiple shaders have not received the same amount of attention. The Quake3 shader model [Jaquays and Hook 1999] provides a rudimentary interface for controlling the application of multiple textures. DirectX Effects [Dir] extend the HLSL shading language and introduce a richer, more powerful interface for controlling the rendering pipeline. NVIDIA provide a superset of this functionality with their CgFX framework [CgF], based on Cg. This is further elaborated on in Section 1.3.

1.2 Shader programming

To demonstrate the issues involved in the implementation of shader based effects and how an active framework like PyFX can alleviate these problems we will use a running example throughout this

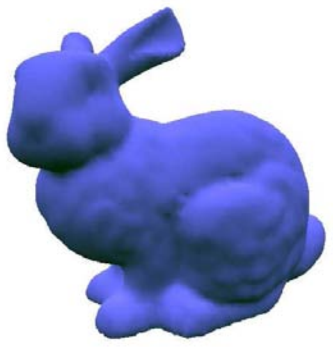

Figure 1: Hemispheric lighting on bunny

paper. The description of this example will be fairly detailed because the causes of application dependencies and need for support can often be found in those details which would usually be omitted in a more concise description. The example we use is the lighting model known as hemispheric lighting, where the idea is to give a contribution of indirect light as a mixture of light from the sky and light from the ground. A given point is colored depending on orientation of its surface normal, the more it is oriented towards the sky the more light from the above light source it receives, and vice versa. The effect of using this model can be seen on the bunny in Figure 1. A shader program which implements this model can be written in Cg as

```
void main(float4 position : POSITION,
          float4 normal   : NORMAL,

      out float4 clipPosition : POSITION,
      out float3 color        : COLOR,

  uniform float4x4 ModelViewProj,
  uniform float4x4 ModelViewIT,
  uniform float4x4 WorldView,
  uniform float3   MaterialColor,
  uniform float3   SkyColor,
  uniform float3   GroundColor)
{
  clipPosition = mul(ModelViewProj, position);
  float4x4 ModelWorldIT = mul(WorldView,ModelViewIT);
  float3 worldNormal = mul(ModelWorldIT,normal).xyz;

  worldNormal = normalize(worldNormal);
  color = lerp(GroundColor, SkyColor,
               (worldNormal.y + 1)/2)*MaterialColor;
}
```

Listing 1: Hemispheric lighting in Cg

This shader program is a vertex shader. It computes the vertex normal in world-space `worldNormal` by using the inverse transpose of the model-world transform `ModelWorldIT`. The amount of incident light of the vertex is then computed by linear interpolation lerp of the sky and ground color where the weighting factor is determined the y-component world-space normal. The incident light is weighted by the material properties of the object. Finally, as required by all vertex programs the clip-space coordinates `clipPosition` are computed using the projection matrix `ModelViewProj`.

The parameters to the program which are marked as `uniform` are those which are constant for the duration of the shader program, whereas the other parameters vary over the vertices of the mesh. The extra field (`POSITION`, `NORMAL`, and `COLOR` in this example), known as the *semantic* of the parameter specify how they are mapped to application data. For example, an `in` parameter with semantic `NORMAL` is specified using OpenGL's `glNormal*` calls from the application.

Shaders require application programmers to write support code for every shader to be used. In order to access shader program parameters an application level identifier is needed. Accessing the parameter identifiers of our example shader from C++ would be as follows.

```
cg_mvp = cgGetNamedParameter(cg_prog,"ModelViewProj");
cg_mvit = cgGetNamedParameter(cg_prog,"ModelViewIT");
cg_wv = cgGetNamedParameter(cg_prog,"WorldView");
cg_materialColor = cgGetNamedParameter(cg_prog,
                                       "MaterialColor");
cg_groundColor = cgGetNamedParameter(cg_prog,
                                     "GroundColor");
cg_skyColor = cgGetNamedParameter(cg_prog,"SkyColor");
```

Listing 2: Finding parameter identifiers

Each time the shader is used it must be bound after which each parameter has to be set to its current value using the corresponding Cg parameter identifier. The target for which the shader program has been compiled, called the profile of the program, must also be enabled.

```
cgGLBindProgram(cg_prog);

cgGLSetStateMatrixParameter(cg_mvp,
    CG_GL_MODELVIEW_MATRIX,CG_GL_MATRIX_IDENTITY);
cgGLSetStateMatrixParameter(cg_mvit,
    CG_GL_MODELVIEW_MATRIX,
    CG_GL_MATRIX_INVERSE_TRANSPOSE);
cgGLSetMatrixParameterfr(cg_wv,
    camera->inverseTransform());
cgGLSetParameter3fv(cg_materialColor, MaterialColor);
cgGLSetParameter3fv(cg_groundColor, GroundColor);
cgGLSetParameter3fv(cg_skyColor, SkyColor);

cgGLEnableProfile(cg_profile);
```

Listing 3: Binding shader program and setting parameters

Changing the parameters of the effect at run-time amounts to changing the local variables used here, for example `MaterialColor`, `SkyColor`, and `GroundColor`.

This code can be compiled and delivered together with the shader code as a complete package which can be used by the artist. However, there are limitations with this approach. All but the most trivial shaders require support code for setting parameters and renderer pipeline states. This support code is specific to each application due to differences in how textures are loaded and accessed, renderer pipeline state are set, *etc*. This gives unwanted dependencies between shaders and applications. Encapsulating these dependencies is a difficult problem since different applications have very different notions of what is important, for instance an artist's tool must be able to provide GUI components for manipulating the shader whereas an engine is primarily concerned with efficiency.

This encapsulation is made even more difficult when using shaders written by an external party. Externally written shaders use different interfaces but must still be accessible in the same manner as in-house developed ones in order to provide a unified working model for both artists and developers. The amount of work needed to adapt such shaders can often be too large.

1.3 Effects

The problems associated with using shaders as shown above are caused by a lack of encapsulation. Information associated with the shader and necessary for the shader to work is mixed with application code and not packaged together with the shader itself. This has called for a new level of abstraction and a new kind of entity to do the encapsulation. These entities are known as *effects*.

Today there are two major effect frameworks in use, the DirectX Effects by Microsoft [Dir] and CgFX by NVIDIA [CgF]. Both provide a text-based format where shader code, parameters and pass specifications are written in one file. This file is loaded by the application and compiled for the current run-time platform. The two formats are very similar and can in many instances be used interchangeably. Using CgFX the hemispheric lighting example can be implemented as:

```
float3 MaterialColor = { 1.0, 1.0, 1.0 };
float3 SkyColor = { 0.5, 0.5, 1.0 };
float3 GroundColor = { 0.0, 0.1, 0.0 };

float4x4 ModelViewProj : MODELVIEWPROJ;
float4x4 ModelViewIT   : MODELVIEWIT;
float4x4 WorldView     : WORLDVIEW;

shader code as in listing 1

technique Hemispheric {
  pass p0 {
    VertexShader = compile vs_1_1 main(ModelViewProj,
                                       ModelViewIT,
                                       WorldView,
                                       SkyColor,
                                       GroundColor);
  }
}
```

Listing 4: Hemispheric lighting in CgFX

This effect declares three parameters which are intended to be set at design time, `MaterialColor`, `SkyColor`, and `GroundColor`, and three parameters that are intended to be set at run-time by the application: `ModelViewProj`, `ModelViewIT`, and `WorldView`. The design-time parameters, also known as *tweakables*, may have associated annotations which can be used by design tools to automatically provide a suitable user interface for setting the parameter. For example a color picker control may be used to set the value of a color parameter. Run-time parameters on the other may have *semantic* identifiers associated with them, and similar to shader semantics, their purpose is to specify the mapping to application data without relying on parameter name. Instead an application can define a number of semantic identifiers which may be used in the effect.

Following the declaration of the effect parameters is the shader code. It is identical to Listing 1 and is therefore omitted here. Finally the effect declares a so called *technique* which describes number of rendering passes needed and the render states to be used in each pass. In this case there is a single rendering pass and in that pass the vertex shader `main` is to be compiled for the shader profile `vs_1_1` and the uniform shader parameters should have the values of the corresponding effect parameters.

Once loaded an effect can be used in the application like this

```
unsigned int numPasses;
effect->Begin(&numPasses, 0);
for (unsigned int p = 0; p < numPasses; p++)
{
  effect->Pass(p);
  renderMesh(mesh);
}
effect->End();
```

Listing 5: Effect usage with CgFX

The textures used by an effect are generally declared as tweakables where an annotation is used to specify the filename.

```
texture colorTexture : DiffuseMap <
   string File = "default_color.dds";
>;
```

Using a texture in a shader program is done indirectly through something called a *sampler* which specifies how the texture is accessed. Declaring a simple 2-dimensional sampler using linear minification and magnification filters for the above texture we write

```
sampler2D colorSampler = sampler_state {
  Texture = <colorTexture>;
  MinFilter = Linear;
  MagFilter = Linear;
};
```

This sampler is then passed to a shader program just as any other parameter.

Effects provide a number of mechanisms for separating applications from shaders. First, the effect format give a clear, high-level, and concise specification of shader programs, textures, and render states. This includes a unified method for handling multipass effects as well as having multiple implementations (fixed-function fall backs *etc.*) of the same visual effect. This specification is independent of the target architecture on which the effect is to run.

Second, tweakables provide the artist with a method for setting parameters at design time. This reduces support code since the engine only needs to concern itself with providing run-time parameters such as projection matrices *etc.*

Third, user-defined semantics provides a method for the engine to provide such run-time parameters. The application defines a number of semantic identifiers which it support and this creates an rudimentary interface to effects which, together with default values for parameters, relieves the effect developers of writing per-effect support code (cf. listings 2 and 3).

However, as in the case with using shaders directly, there still exist a problem of encapsulation. The application defines an interface for the effects by using user-defined semantics. This interface is fixed, and this limits the number of shaders that may be expressed and used within a single application.

2 PyFX

The limitations in encapsulation of existing effect frameworks is due to the fact that effects are passive entities, text files, which are operated on by the application, which is the active party. If this relationship could be reversed so that effects are active instead, a better interface can be built where they can be responsible for retrieving the data they need from applications rather than the other way around. To achieve this reversed flow of control the effects must be embedded in a context which can do actual execution on their behalf.

2.1 PyFX overview

We have used Python, an existing scripting language to develop an active effect framework called PyFX. The current implementation supports applications using OpenGL and shaders written in Cg and its feature set closely resembles that of CgFX. In PyFX however, Python is used both to implement the framework and to write the effects themselves.

In an object-oriented language, it is natural to represent different effects as subclasses to a common effect base class. The subclasses implement specific functionality whereas functionality common to all effects are inherited from the base class. The object-oriented model also provides a natural mapping to the collaborative workflow between programmers and artists. Effect programmers write new effects by making new effect subclasses, whereas the artist provides textures, sets parameters, etc. to make effect instances from existing classes.

Below is the hemispheric lighting example written in PyFX. It shows the Python class `Hemispheric` as a subclass of the general `Effect` class.

```
class Hemispheric(Effect):
   vs = Cg("""
      shader code as in listing 1
   """)

   SkyColor    = (0.5, 0.5, 1.0)
   GroundColor = (0.0, 0.3, 0.0)

   def __init__(self,
                MaterialColor = (1.0, 1.0, 1.0)):
       Effect.__init__(self)

       self.MaterialColor = MaterialColor

       self.technique = [Pass(VertexShader = vs())]
```

Listing 6: Hemispheric lighting in PyFX

The declaration has two main parts: the class variables and the constructor (the `__init__` member). The first class variable `vs` contains the shader program as a string wrapped by an instance of a Python class called `Cg`. and the other two class variables `SkyColor` and `GroundColor` are simply effect parameters. The class constructor, which creates new instances of the class, takes one additional effect parameter `MaterialColor` as an argument. The ability to differentiate between class variables and instance variables allows the effect writer to indicate that some parameters are intended to be the same for all instances of the class whereas other parameters may be different. The constructor body calls the superclass constructor and sets the instance variable `MaterialColor` of the object. Finally, the instance variable `technique` is set to specify that this is a single pass effect and that the pass should use the shader `vs` as its vertex shader.

Having instantiated this effect, for example like this

```
effect = Hemispheric(MaterialColor = (0.0,0.0,1.0))
```

it can be applied to a mesh by

```
while effect.hasMorePasses(mesh):
    renderMesh(mesh)
```

The `Effect` member function `hasMorePasses` does setup for each pass of the effect and also specifies how many times the mesh needs to be rendered.

Having applied effects to meshes the next issue is the passing of information from the application to the effect and its shader. In PyFX this data can be passed through a number of different channels.

The most obvious way is through constructor parameters when the Hemispheric effect is instantiated. The example above shows how `MaterialColor` is set to the color blue.

In the hemispherical lighting example the constructor parameters correspond exactly to an instance variable of the effect. Another method of passing data to the shader is to assign new values to this variable. For example

```
effect.MaterialColor = (0.5, 0.5, 1.0)
```

changes material color so that it is now light blue. Similarly class variables can also be assigned new values

```
Hemispheric.GroundColor = (0, 0, 0)
```

The framework then make sure that these changes are made available to the shader code.

Another type of parameters are the transformation matrices used by the effect; `ModelViewProj`, `ModelViewIT` and `ViewWorld`. The matrices `ModelViewProj` and `ModelViewIT` can be retrieved from the OpenGL rendering pipeline and in PyFX this is handled automatically.

The third parameter `ViewWorld` needs special treatment. It is the inverse camera transform, used by the effect to compute the `ModelWorld` transform, neither of which can be automatically retrieved from the pipeline. It must therefore be provided by the application. Since this parameter is the same for different instances it makes sense to make it a class variable. However, it is even more general than that since you could easily think of other effects that might need it. In this case we can therefore set it as a class variable on the `Effect` base class, for example:

```
Effect.ViewWorld = camera.inverseTransform()
```

Yet another method for passing data from the application is when the effect needs additional data at each vertex, *i.e.* non-standard varying parameters. In our hemispherical lighting example this is not the case, but a more advanced version of hemispheric lighting can used to illustrate this case [Hem]. This version use additional per-vertex mesh data, called the *occlusion factor*, which determine the amount of hemispheric light which reach the point in question. If the shader program has the following prototype

```
void main(..., float OcclusionFactor : COLOR )
```

Then, if the mesh has an array member `OcclusionFactor`, PyFX will automatically bind this to the varying parameter with the same name.

2.2 PyFX details

2.2.1 Techniques

The structure of PyFX effects is inspired by that of DirectX Effects and CgFX frameworks. As in these each effect contain one or more techniques. Each technique contain a number of passes which are to be run consecutively. Each render pass has associated render states specifying the necessary pipeline states required to run the pass. Specifying that back-face culling should be disabled while alpha-blending is enabled is written in CgFX as

```
pass p0 {
  CullMode = NONE,
  AlphaBlendEnable = True
}
```

In PyFX the same render pass specification would look like

```
Render(CullMode = None,
       AlphaBlendEnable = True)
```

A single technique effect for CgFX is shown in listing 4. The corresponding effect in PyFX is given in listing 6. Providing two techniques `Hemispheric` and `Ambient` in CgFX is done by providing multiple technique blocks

```
technique Hemispheric {
  pass p0 {
    VertexShader = compile vs_1_1 main();
  }
}

technique Ambient {
  pass p0 {
    Color = <ambientColor>;
  }
}
```

The same would be written in PyFX as

```
technique = {}
technique['Hemispheric'] = [
    Render(VertexShader = vs())]
technique['Ambient'] = [
    Render(Ambient = AmbientColor)]
```

2.2.2 Textures

Texturing in PyFX is, as in CgFX, divided into textures and samplers. Declaring the same texture and sampler as above (Section 1.3) in PyFX would be written as

```
colorTexture = Texture(filename="default_color.dds")
colorSampler = Sampler(colorTexture,
                       MinFilter = Linear,
                       MagFilter = Linear)
```

This sampler can then be used either by a shader program, using parameter resolution, or in the fixed-function pipeline by

```
Render(Texture0 = colorSampler)
```

Multi-texturing is naturally supported and when using multiple textures in a shader program this is automatically handled by the shader parameter resolution code. For fixed-function effects the different texturing-units are accessible via

```
Render(Texture0 = colorSampler,
       Texture1 = lightMapSampler)
```

where `colorSampler` and `lightMapSampler` are two samplers with appropriate settings.

2.2.3 Shaders

Shaders are provided via strings wrapped with classes providing information on the type of shader code contained in the string. Sometimes it is useful to specify the target for which a given shader should be compiled. This can be achieved via

```
Render(VertexShader = vs(target=arbvp1))
```

Also, passing explicit parameters to shader programs can be done by adding keyword arguments to the shader invocation. Suppose we have an outlining effect which draws a gradually more transparent outline around an object. This effect should run multiple passes with the same shader program (called `outline`) but with a parameter `offset` determining the size and opacity of the outline

```
[Render(VertexShader = outline(offset=1.0)),
 Render(VertexShader = outline(offset=0.75)),
 Render(VertexShader = outline(offset=0.5)),
 Render(VertexShader = outline(offset=0.25))]
```

Listing 7: Setting compile-time parameters

The same thing can be expressed in CgFX but the result is more verbose since every shader parameter must be passed explicitly.

If a shader program source code `shader` contains multiple programs, say a vertex shader `shadeVertex` and a pixel shader `shadePixel`, these programs entries can be accessed by the corresponding methods on the shader object

```
Render(VertexShader = shader.shadeVertex(),
       PixelShader = shader.shadePixel())
```

2.2.4 Parameter resolution in PyFX

Application level variables having the same name as the shader program parameters are used as arguments to the shader program. These arguments are defined in one of the following places:

- Either it is a compile-time parameter to the shader program (see listing 7), or
- an attribute of the effect object, or
- an attribute on the mesh currently being rendered, or, lastly,
- a member of a predefined set of state parameters giving access to current pipeline states.

Attributes of the effect instance include both instance parameters, such as the material color parameters above (Section 2.1), and class parameters, `SkyColor` and `GroundColor` above. As usual the class scope includes the scope of its superclass making the `WorldView` transform accessible to the shader programs. In the above examples the `OcclusionFactor` is a mesh attribute and them `ModelViewProj` and `ModelViewIT` matrices are both pipeline state parameters.

If there are multiple variables with the same name the order of precedence is that compile-time parameters take precedence over instance attributes, which take precedence over class variables. Effect class variables take precedence over mesh attributes and state parameters are used last.

This gives a natural correspondence between parameters to the shader program and application data. Setting effect class-specific values amounts to setting effect class-variables whereas effect instance-specific values are set by setting the appropriate attribute on the effect instance in question. Effects take a more active role since they are allowed to extract information from the mesh currently being rendered thus minimizing the amount of application level dependencies.

The mapping is recursive so the following Cg shader program

```
struct Light {
  float3 position;
  float4 color;
};

void main(..., uniform Light light) { ... }
```

will use `position` and `color` member of the application level variable `light`.

2.2.5 Name maps

The lookup scheme above gives great flexibility in both writing and using effects. However when dealing, for instance, with third-party effects a name-based lookup is not always sufficient since naming conventions may differ. Suppose we wish to use an effect which uses the name `DiffuseMap` where our application use `DiffuseTex`. An obviously unattractive solution would be to add `DiffuseMap` to our code and make sure to update it each time we change `DiffuseTex`.

PyFX solves this problem by having user defined *name maps*. The `Effect` class allows us to pass a dictionary of how parameter names at the shader level should be mapped to parameter names at the application level. Defining a dictionary containing our mappings and passing it to the effect nicely handles this.

```
myNameMap = {'DiffuseMap' : 'DiffuseTex'}

effect = SomeTexture(nameMap = myNameMap)
```

A request for the `DiffuseMap` will now be automatically translated to a requests for `DiffuseTex`.

2.2.6 Language embedding

The fact that Python is used to write effects and not only for implementing the framework is convenient but not strictly necessary. It would have been possible to write an interpreter and for example use the CgFX format. However, the complete *embedded* of effects in Python has the advantage that all the ordinary language features such as lists, tuples, loops, functions, dictionaries, list comprehension, etc. are available to the effect writer[Lejdfors and Ohlsson 2004]. As a very simple example we could have used a list comprehension to write the pass specification of the outline effect (listing 7) as

```
[Render(VertexShader=outline(factor=f))
                        for f in [1.0, 0.75, 0.5, 0.25]]
```

2.2.7 Module-style effects

When using concrete subclasses of `Effect` the application needs to know about every such class at compile-time, something known as the *library problem*. This is clearly not desirable in a graphics application and it was one of the problems effects where created to alleviate. In traditional object-oriented design it is solved by introducing an abstract factory for handling instantiation of concrete subclasses [Gamma et al. 1994]. However, using the flexibility of Python we can provide a method which simultaneous solves this problem while giving a cleaner and more direct syntax for declaring effects. Effects can be implemented simply as Python modules which can be loaded by

```
effect = Effect('Hemispheric')
```

This loads the `Hemispheric` effect module which can be used just as any other effect. Note however, that since the actual subclass is not known setting class variables such as `GroundColor` (cf. Section 2.1) is not possible.

2.2.8 Image processing

PyFX also provides a mechanism for specifying render targets other than the frame buffer to which rasterization should occur. Furthermore it is possible to have passes which do not render geometry but instead do shader based image processing. These two features, which are not available in CgFX or DirectX, allow effects such as blurring, edge detection, image compositing *etc.* to be expressed in an application independent manner.

3 Implementation

PyFX is implemented on top of PyOpenGL [PyO] and a SWIG [SWI] generated interface to the Cg runtime library. The implementation consists of about 800 lines of Python code. The bulk of it is concerned with basic functionality needed in any effect framework such as loading and binding textures, compiling shader programs, and initializing OpenGL extensions. The remaining part implements the distinguishing features of PyFX, *i.e.* mapping declarative state specification to function invocations and performing parameter resolution. This part is remarkably small, only about 10% or 80 lines of code. This compactness is possible because of Python's dynamic object model and introspection facilities.

3.1 Renderer management

The entry point of the PyFX framework is provided by the top-level `Effect` class. It is essentially a container for other objects, *i.e.* techniques, passes, textures, samplers, and shaders. These classes interact with the underlying graphics API through a global `RenderState` singleton class which implements manipulation of the renderer pipeline state. The majority of its methods correspond one-to-one to the available state variables. For instance the `CullMode` state is implemented as

```
class RenderState:
  ...
  def CullMode(self, val):
      if val:
          glEnable(GL_CULL_FACE)
          glFrontFace(val)
      else:
          glDisable(GL_CULL_FACE)
```

When a `Render` is activated it instructs the `RenderState` object `state` to change the state of the rendering pipeline. This is done by mapping every state specified in the pass object to a method invocation. For example, a pass specified by

```
Render(Color = (1.0, 0.0, 0.0),
       CullMode = None)
```

will result in the following method calls on:

```
state.Color((1.0, 0.0, 0.0))
state.CullMode(None)
```

Doing this mapping is the responsibility of the `Render` class and by using the dynamic introspective features in Python, it can have a very small implementation:

```
class Render:
  def __init__(self, **kwords):
      self.kwords = kwords

  def use(self, state):
      for s,v in self.kwords.items():
          marshalFX(state,s,v)
```

The `marshalFX` maps the state name `s` to the proper method name and calls this method with argument `v`. It is similar to the marshaling used by RPC (remote procedure calls), whereby serialized data (dictionary tuples) are converted to method invocations. Implementing `marshalFX` is a two-liner:

```
def marshalFX(obj, name, *args):
    method = getattr(obj,name)
    return method(*args)
```

3.2 Texture and sampler state

The class `Texture` provides an encapsulation similar to `RenderState` but for the available texture states such as filtering, texture coordinate wrapping, etc. The texture state information is maintained by the corresponding `Sampler` and it is responsible for marshaling this information to method invocations on the `Texture` object.

When a `Sampler` is used by either the fixed-function pipeline or by a shader the framework allocates a free texture unit and asks the sampler to bind itself to that unit.

3.3 Shaders

Just as samplers are responsible for performing binding textures and setting texture states, every `Shader` object is responsible for performing its own loading, binding, compilation, and parameter resolution. This implementation is actually contained in subclasses for different shader programming languages. Currently the only subclass implemented is `Cg`.

When the pass specifies a vertex or fragment shader the `state` object instructs the shader to bind itself. A shader binding itself includes setting the value of every parameter needed by the shader. The mapping scheme of PyFX between parameters and application variables is implemented by a `Resolver` object whose responsibility it is to search the effect and mesh name spaces as well as providing name mapping (Section 2.2.5). The resolver searches a list of objects for a given attribute, optionally transform the attribute name via the name mapping dictionary:

```
class Resolver:
   def __init__(self,nameMap,*objs):
       self.nameMap = nameMap
       self.objs = objs

   def __getattr__(self,attr):
       if self.nameMap.has_key(attr):
          attr = self.nameMap[attr]

       for obj in self.objs:
          if hasattr(obj, attr):
              return getattr(obj, attr)
```

The `Cg` class use the resolver to locate shader parameters and set these by invoking the corresponding CgGL functions. For simple variables the `cgGLSetParameter`-family of functions are used. Aggregate parameters, such as arrays and structs, are handled by iterating over the members and setting each element recursively.

4 Conclusions and future work

The most prominent features provided by the PyFX framework is the decoupling of effects from the application. This "activation" of an effect, enabling it to obtain needed data from *e.g.* the current mesh without the need to introduce application level support code, greatly reduces dependencies between effects and the application. Using this activation together with the introspection features of Python gives a natural mirroring between data at the application level and data at the level of shader programs. This also eliminates the need for user-defined semantics since there is no longer any need to provide *ad hoc* hooks for applications to provide specialized data and operations. Instead the object-oriented extensible nature of the host programming languages can be used to provide this functionality natively at the effect level.

There are some limitations however, in the current implementation of PyFX. Support for manipulating fixed-function effect parameters is limited. Consider a simple effect such as

```
class SimpleColor:
    color = (1,0,0)
    technique = [Render(Color=color)]
```

Manipulating the `color` attribute of this effect will not have the desired effect, the color used for drawing will remain red. The reason why the parameter resolution algorithm (Section 2.2.4) can not be applied in this case is that it requires access to the parameter names. These names are only available to shader based effects where they are supplied by the Cg run-time library.

The overall purpose of PyFX is to be a flexible tool for investigating what kind of features and functions are needed to make effect programming as easy and productive as possible. Future work includes investigating how effects can be combined efficiently at run-time allowing, for instance, stencil-buffer shadow algorithms to coexist with other visual effects.

Acknowledgement

The authors wish to thank Benny Åkesson and Andreas Hansson for help with implementing the framework as well as providing a number of interesting effects testing various aspects of PyFX. We also extend our thank to Chuck Mason for providing the initial Python interface to Cg.

References

3D LABS. 2002. *OpenGL 2.0 Shading Language White Paper*, 1.2 ed., February.

CgFX 1.2 Overview. `http://developer.nvidia.com/`.

COOK, R. L. 1984. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, 223–231.

DirectX SDK Documentation. `http://msdn.microsoft.com/`.

ELLIOTT, C. 2004. Programming graphics processors functionally. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, ACM Press, 45–56.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns*. Addison-Wesley.

GRAY, K. 2003. *DirectX 9 programmable graphics pipeline*. Microsoft Press.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ACM Press, 289–298.

Hemispheric lighting. Example in DirectX 9 SDK documentation. MSDN Library, `http://msdn.microsoft.com`.

JAQUAYS, P., AND HOOK, B. 1999. *Quake III Arena Shader Manual*, 12 ed. Id Software Inc., December.

LEJDFORS, C., AND OHLSSON, L. 2004. Tools for real-time effect programming. Submitted to publication.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph. 22*, 3, 896–907.

MCCOOL, M., QIN, Z., AND POPA, T. 2002. Shader metaprogramming. In *Graphics Hardware*, T. Ertl, W. Heidrich, and M. Doggett, Eds., 1–12.

MCCOOL, M., TOIT, S. D., POPA, T. S., CHAN, B., AND MOULE, K. 2004. Shader algebra. In *To appear at SIGGRAPH 2004*, 9 pages.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM Press, 231–240.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 159–168.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 425–432.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, 287–296.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 159–170.

PyOpenGL project. `http://pyopengl.sf.net/`.

SWIG project. `http://www.swig.org/`.