Object-Oriented Shader Design

Roland Kuck

Virtual Environments Department, Fraunhofer IAIS, Germany

Abstract

We present an extremely lightweight object-oriented framework for writing shaders. It provides a way to invoke methods of objects from the shading language and to use references of objects as normal variables. Classes are declared and instantiated in the application language using proxy classes. We then apply object-oriented design to several typical shading problems showing their strength compared to the standard methods.

Categories and Subject Descriptors (according to ACM CCS): D.1.5 [Programming Techniques]: Object-oriented Programming I.3.6 [Computer Graphics]: Methodology and Techniques

1. Introduction

Graphics hardware has become extremely programmable. The previous fixed-function design has been exchanged with stages that can execute an arbitrary program called a *shader*. Shaders are not only used to perform the shading of the geometric primitives, but can be used to transform vertices, perform skinning of meshes or generate meshes as well.

Today shaders are written in C-like languages. These shaders are uploaded to specific pipeline stages. Therefore the structure and the granularity of a shader is defined by the hardware design and not the given problem. It is also difficult to write maintainable and reusable code using these languages alone.

Both problems can be solved by using object-oriented programming. It structures the problem naturally and allows for the development of reusable components. Shading e.g. can be presented as a *surface object* that interacts with possibly multiple *light source objects*. Adding a light source to a scene should then be possible by creating a light object and connecting it appropriately.

We therefore introduce an object-oriented framework for writing shaders. It is extremely lightweight and has no runtime costs. It also does not hide the actual shading language and hence can be adapted easily to new hardware features. It provides a way to invoke methods of objects from the shading language and to use references of objects as normal variables. Classes are declared and instantiated in the application language using proxy classes.

We then apply object-oriented design to several typical shading problems showing their strength compared to the standard methods. This list of problems is by no means meant to be complete, but it was chosen to highlight the advantages of using classes and objects.

2. Related Work

One of the first higher-level shading languages was the *Stanford shading language* [PMTH01]. Shaders are divided into different groups depending on their purpose, e.g. surface shaders. It closely matches the ideas of the Renderman shading language [HL90] with changes required for graphics hardware. This abstraction makes programming easier but also imposes restrictions. Our object-oriented framework allows a similar abstraction to be used. It does not modify the shading language but uses a library concept.

The Cg [MGAK03] and the OpenGL Shading language [Ros06] closely resemble the way the hardware works. Only the lowest hardware levels are abstracted using a C-like language instead of an assembly language making larger scale, modularized development difficult. Cg offers some support for interfaces but has no dynamic polymorphism used in the state pattern (see section 4.4) and uses aggregation for all attributes instead of associations leading to duplication of data when multiple classes reference the same data. It also only provides a complex runtime API to manipulate these data structures making them difficult to handle.

A different approach is used by Sh [MQP02]: A C++ API is used to write the shading code directly in C++. The instructions are translated to the shading language. Due to this abstraction changes in the graphics hardware require changes in the library. It also does not support dynamic polymorphism. In [MTP*04] a shader algebra is built using Sh, where components are connected. While the paper stresses the fact of reusable components and easy reconfiguration, this type of data flow design does not allow more complex control structures as described in section 4.1.

In [MSPK06] an abstract shade tree is presented. Using building blocks that are connected shaders can be written. A visual representation is available. No interface to the host application is discussed. The object-oriented approach presented here provides a richer infrastructure of which the shading process is only one part of.

The system in [LSK*06] has similarities to our approach. It focuses on building complex data structures for the GPU. No direct usage of objects to express behavior is discussed.

CUDA [NVI07] is a low-level framework to perform computations on the GPU. It extends *C* and offers a library with little support for graphics tasks. The access to the texture hardware is limited, while the rasterizer and the framebuffer are not accessible. Some tasks can be easier expressed with this framework though and an extension to the object system described here could be used to combine it with the graphics pipeline.

3. Framework

We present a framework that consists of two dependent parts: an object system for the *OpenGL Shading Language* (GLSL) and proxy objects in C++ that are used to directly manipulate the objects. We first describe the usage and then give details about the implementation.

3.1. Usage

The fundamental type we added to GLSL is the reference type and is used to access objects. Using a reference *texture* to an object we can call a method like a normal function passing the reference as the first argument:

```
color = TextureArray_texture2D(texture, index, uv);
```

Classes are declared in C++ as shown in figure 1. The C++ classes have two purposes: they are used to define the data structure of the class in GLSL and they function as a proxy [GHJV93] to this data structure. Objects are also very closely related. For every C++ object there exists exactly one GLSL object. This makes lifetime management easy as one only has to manage the C++ object.

The C++ declaration code makes heavy use of advanced template programming like the *Curiously Recurring Template Pattern* [Cop96] and some static data structures (hidden by the macros *DERIVED_DECL* and *CLASS_INIT*) to compensate for the lack of introspection support in C++.

```
class TextureArray;

class HardwareTextureArray :
  public Shader<HardwareTextureArray, TextureArray>
 {
 public:
  HardwareTextureArray(list<Image> tiles);

  /*virtual*/ vec3 texture2D(int_ index, vec2 uv)
   { return invoke<vec3>("HardwareTA_texture2D"); };

 private:
  // Declare attributes as usual in C++
  sampler2DArray<uniform> texture;

  DERIVED_DECL(HardwareTextureArray, TextureArray)
 };
CLASS_INIT(HardwareTextureArray, "TextureArray.glsl",
           (texture2D), (texture))

class EmulatedTextureArray :
  public Shader<EmulatedTextureArray, TextureArray>
 {
 public:
  EmulatedTextureArray(list<Image> tiles);

  /*virtual*/ vec3 texture2D(int_ index, vec2 uv)
   { return invoke<vec3>("EmulatedTA_texture2D"); };

 private:
  sampler2D<uniform> texture;
  int_<uniform> num_rows;
  vec2<uniform> scale;

  DERIVED_DECL(EmulatedTextureArray, TextureArray)
 };
CLASS_INIT(EmulatedTextureArray, "TextureArray.glsl",
           (texture2D), (texture)(num_rows)(scale))
```

Figure 1: *Class declaration of two derived classes of* TextureArray. *The declaration of the base class is omitted.*

The exported attributes of the C++ class can be hidden just like any other member variable of a C++ class. This can be used to offer a more convenient interface to the host application and to ensure encapsulation from the implementation. The exported methods are not meant to be called directly from C++ but are only used to declare the required function signature. The implementation of the C++ class returns the name of the GLSL function to be called when the method is invoked. We need to specify this function as in figure 2.

```
vec3 HardwareTA_texture2D(HardwareTextureArray_SELF,
                          int index, vec2 uv)
{
  vec3 coord = vec3(uv, float(index));
  return texture2DArray(texture, coord);
}

vec3 EmulatedTA_texture2D(EmulatedTextureArray_SELF,
                          int index, vec2 uv)
{
  float u = floor(index / num_rows);
  float v = mod(index, num_rows);
  vec2 coord = scale * vec2(u, v);
  return texture2D(texture, uv*scale + offset);
}
```

Figure 2: *Implementation of classes in GLSL. The simplified emulation code does not correctly sample the border of tiles.*

```
#define HardwareTextureArray_SELF \
  OBJREF self, sampler2DArray texture

uniform sampler2DArray obj_0x1_texture;
uniform sampler2DArray obj_0x2_texture;

vec3 TextureArray_texture(OBJREF self,
                          int arg1, vec2 arg2)
{
  if (self == 1)
    return HardwareTA_texture2D(self, obj_0x1_texture,
                                arg1, arg2);
  else if (self == 2)
    return HardwareTA_texture2D(self, obj_0x2_texture,
                                arg1, arg2);
  // Default return value
  return vec3(0., 0., 0.);
}
```

Figure 3: *Generated GLSL dispatcher code for two instances of* HardwareTextureArray

3.2. Implementation

The exact definition of the reference type can be useful to encode this information into the vertex data or a texture. We therefore see this definition as part of the interface and not as an implementation artifact.

References are simple integer numbers. All referenced objects are enumerated and these numbers are used in the GLSL code. A dispatch function is automatically generated (see figure 3). Normally the object references are constants and therefore the dispatch code is optimized out. Using objects thus does not influence the performance of the program. Object references do not need to be constant and we describe such a situation in section 4.4. If a method is called with an object reference to an object that does not exist, the default behavior is to provide a standard return value. This is defined behavior and can thus be relied on (see section 4.1).

We also support a list data type. To achieve this aliases for the object numbers are created and all the objects in one list get assigned additional consecutive numbers. A list can then be represented by the start and the end value and these references act like iterators [GHJV93].

One method is given as the entry point of a shader. All required objects that are directly or indirectly referenced are automatically collected. A single object can be used in different pipeline stages at the same time (see section 4.3).

4. Application

We apply the framework described above to selected problems typically encountered in shader programming. They should provide a good idea of how the object-oriented design works in the shader programming context.

4.1. Illumination and Shading

The standard use of a shader is to calculate the color of an illuminated surface element. This usually involves a surface material and possibly several light sources. We need to evaluate the BRDF of the material given by some shader code for each light source. It therefore seems logical to model the material and the light source as classes and iterate over a list of light source objects in the material class and retrieve the received light amount from each. But what happens if some materials react differently to certain kinds of light sources? A example for this is given in [AG99]: UV light.

We can use the visitor pattern [GHJV93] or rather the double dispatch technique here. The implementation for the material calls an *illuminance()* method of the light source and passes a reference to itself. The light source then calculates the light direction and the intensity and calls *illuminate()* on the material object passing this information along. It can also call *illuminate_uv()* to let the material receive UV light. Materials that are not sensitive to UV light simply do not implement this method.

4.2. Texture Array Access

A new feature of modern graphics hardware is *texture arrays* [Bly06]. These are multiple 2D textures that are bound to the same texture unit and that can be selected at runtime using an index. On older hardware this can be emulated by tiling the different textures in one larger 2D texture and transforming the texture coordinates in the shader.

The layout of the tiles has to be stored in addition to the sampler parameter. We can hide the details by using classes. Two implementations are given: One for the hardware that

exposes direct support for texture arrays and the other one that provides the emulation (see figures 1, 2 and 3).

The C++ interface can be designed to present the host application a uniform interface hiding the texture binding process. A factory [GHJV93] can create the instances depending on the capabilities of the used graphics hardware.

4.3. Shared Data

There are two situations in which data needs to be shared: The data from different pipeline stages in the graphics hardware is propagated to the next stage and therefore shared. It is also possible that different stages require the same information and thus require shared data.

Different shading languages provide direct support for the first problem. In Cg the data from the previous stage is provided as arguments to the entry function of the next. Thus it is required that the entry point knows about all shared data. In GLSL these values are global parameters but global variables expose the data directly to the whole shader code.

Using objects we can provide a much cleaner solution: Use one object in multiple pipeline stages at the same time. Shared data is then identical in all stages. Data written in one stage and read from another can also be used by simply declaring attributes with the appropriate qualifiers, e.g. *varying* for the connection of vertex and fragment shaders. We can also provide different interfaces for different stages to encapsulate the implementation.

4.4. State

Assume we are implementing a particle system. We want to color different groups of particles in different ways: Some are colored with a fixed color, while others are illuminated using a light source. If each particle contains a group identifier we can use *if-then* constructs to check the type and handle each case appropriately.

Maintaining the dispatch function is an unnecessary bottleneck. The state pattern [GHJV93] provides a simple solution. It uses the polymorphic dispatch function and one simply associates a state object with each particle. As the reference to an object is an integer we can store it in the vertex data and invoke the methods in the shader.

5. Conclusion and Future Work

We discussed object-oriented design in the context of shading and showed how it improves the design. We also introduced a lightweight framework to apply these methods.

As described above the framework relies on the compiler to optimize out the dispatch functions. Our tests show that the compiler does perform this optimization reliably. A preprocessor can reduce this dependency but it needs to be updated with new revisions of the shading language.

The object system creates a tight coupling between the C++ and the GLSL parts of the program. This can make it difficult to reuse the shading objects in other languages. Normal methods to access objects from other languages can be used, e.g. automatically generated wrappers.

We are interested to continue this research and evaluate more complex and larger shading algorithms in the context of object-oriented design.

References

[AG99] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., 1999, pp. 222–224.

[Bly06] BLYTHE D.: The Direct3D 10 system. In *Proc. SIGGRAPH '06* (2006), ACM Press, pp. 724–734.

[Cop96] COPLIEN J. O.: A curiosly recurring template pattern. In *C++ Gems* (1996), Lippman S. B., (Ed.), Cambridge University Press, pp. 135–144.

[GHJV93] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns: Abstraction and Reuse of Object - Oriented Design*. Addison Wesley Longman Publishing Co., Inc., 1993.

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. In *Proc. SIGGRAPH '90* (1990), ACM Press, pp. 289–298.

[LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph. 25*, 1 (2006), 60–99.

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A system for programming graphics hardware in a c-like language. In *Proc. Siggraph '03* (2003), pp. 896–907.

[MQP02] MCCOOL M. D., QIN Z., POPA T. S.: Shader metaprogramming. In *Proc. SIGGRAPH/Eurographics Graphics Hardware Workshop '02* (2002), Eurographics Association, pp. 57–68.

[MSPK06] MCGUIRE M., STATHIS G., PFISTER H., KRISHNAMURTHI S.: Abstract shade trees. In *Proc. Symposium on Interactive 3D graphics and games '06* (2006), ACM Press, pp. 79–86.

[MTP*04] MCCOOL M., TOIT S. D., POPA T., CHAN B., MOULE K.: Shader algebra. In *Proc. SIGGRAPH '04* (2004), ACM Press, pp. 787–795.

[NVI07] NVIDIA: *CUDA Programming Guide*, 2007.

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A real-time procedural shading system for programmable graphics hardware. In *Proc. SIGGRAPH '01* (2001), ACM Press, pp. 159–170.

[Ros06] ROST R. J.: *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., 2006.