EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

Data Driven Graphical Applications: A Fluid approach

A. Jones and C. Mantle and D. Cornford

Knowledge Engineering Group, Computer Science, Aston University, Birmingham, UK

Abstract

The inclusion of high-level scripting functionality in state-of-the-art rendering APIs indicates a movement toward data-driven methodologies for structuring next generation rendering pipelines. A similar theme can be seen in the use of composition languages to deploy component software using selection and configuration of collaborating component implementations. In this paper we introduce the Fluid framework, which places particular emphasis on the use of high-level data manipulations in order to develop component based software that is flexible, extensible, and expressive. We introduce a data-driven, object oriented programming methodology to component based software development, and demonstrate how a rendering system with a similar focus on abstract manipulations can be incorporated, in order to develop a visualization application for geospatial data. In particular we describe a novel SAS script integration layer that provides access to vertex and fragment programs, producing a very controllable, responsive rendering system. The proposed system is very similar to developments speculatively planned for DirectX 10, but uses open standards and has cross platform applicability.

Categories and Subject Descriptors (according to ACM CCS): D.2.11 [Software Engineering]: Software Architectures I.3.3 [Computer Graphics]: Methodology and Techniques D.2.13 [Software Engineering]: Reusable Software I.3.8 [Computer Graphics]: Applications

1. Introduction

Data processing is at the heart of almost all modern software. Various paradigms are used to develop software for the specific requirements. In the traditional view of computer programs, they are seen as the flow of data through a fixed set of processes to produce the desired output. In recent years more attention has been paid to ideas of data driven programming, especially in the computer games industry. Data driven programming means that data passed into the program affects not only the behaviour, but also the structure, of the executable program. Thus the distinction between data, scripting and code is more and more blurred. In this paper we describe a framework, Fluid, that supports data driven programming concepts and will incorporate a data driven rendering system that can support near-photorealistic visualizations at interactive frame-rates.

Fluid is a component framework with a particular emphasis on driving runtime behaviour via high-level abstractions and data manipulations. In the Fluid framework, data drives the logical and behavioural composition of a component-based application. Fluid utilises concepts available to the object oriented programming paradigm to support expressive composition and manipulation of small scale software implementations. Fluid thus introduces object-oriented data-driven programming (OODDP) to the field of component based software development [JC06].

Fluid's movement toward higher-level abstractions reflects a similar trend in popular modern rendering APIs. While previous graphics hardware architectures have provided limited facilities for the description of object appearances and special effects, modern hardware and software APIs allow a wide range of effects to be described using a variety of high level languages. Furthermore, a small number of state-of-the-art APIs (such as CgFX and DirectX 10) offer an additional level of abstraction, whereby multiple effects may be chained together via an abstract scripting system written within the effect scripts themselves.

The paper is organised as follows: Section 2 describes the design of a high-level scripting interface to a high-performance rendering pipeline, placing the design within the context of both previous advances and modern standards in rendering technology. We describe the development of a scripted rendering system that will form part of a component based application, whose initial domain is the visual-

ization of geospatial scenes. Section 3 provides an overview of the Fluid project, relating its current incarnation to an earlier prototypical implementation. We describe our plans to include a data-driven rendering system as part of the final proof-of-concept system. Focusing on the use of data-driven methodologies, we describe how application composition and runtime behaviour may be driven by high-level data manipulations. We conclude in Section 4 with an overview of the relationship between the Fluid framework and the rendering technologies described.

2. A data-driven rendering pipeline

The complexity and dynamic nature of modern real-time rendering software has matched the extraordinary power and flexibility of modern graphics hardware. Simple textured surfaces have been replaced by near-photorealistic representations using physical approximations, and developers now have the ability to apply film-quality pre- and post-process effects to rendered scenes. A generic real-time render system must give developers and artists low-level control of the rendering process, whilst still maintaining the speed required for real-time applications.

We propose a generic, data-driven render system to sit between client application and graphics library (GL), which uses text-based, human-readable descriptions of effects and a scripting component to abstract the vast majority of the logic required to generalize the creation and use of effects. The PirateHat render system library (PirateHat|RS), currently under development, operates on one or more effect definition files, each containing chains of embedded pipeline control scripts.

2.1. Architecture of graphics hardware, past to present

The typical render pipeline, as employed by the majority of modern graphics hardware, can be seen in Figure 1.

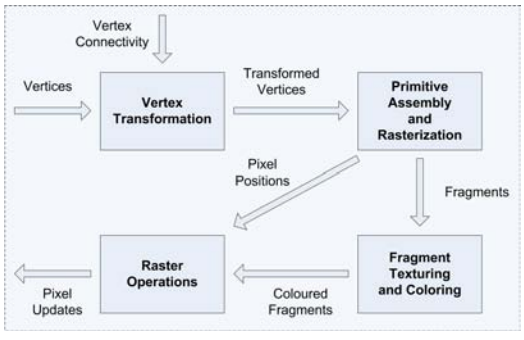

Figure 1: *A typical hardware render pipeline.*

The render pipeline, however, does not simply concern the graphics hardware. Its layered nature is illustrated in Figure 2. A rendering pipeline extends out of the graphics hardware, through its driver and a graphics library, into the application. The capabilities of graphics hardware, however, have always determined the capabilities of higher-level abstractions.

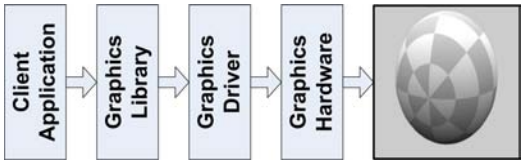

Figure 2: *The four layers which make up a rendering pipeline, from the most abstract level to the finished, rendered image.*

2.1.1. Fixed function rendering pipelines

Early commercially-available GPUs had fixed pipelines, which conformed to the specifications of one or more specific graphics libraries. The rendering available through fixed-function GPUs was very limited, allowing only the material properties, texturing and lighting prescribed by the specification of the GL in use. Once vertex and texture data was submitted to the GPU through a GL, client applications had no direct control over the rest of the rendering process; the data was pushed through the pipeline, and the resulting rasterization was displayed.

2.1.2. Configurable rendering pipelines

The limitations of fixed-functionality pipelines were partly addressed by the next generation of GPUs, through the introduction of configurable pipelines. Configurable pipelines allowed developers to indirectly manipulate the rasterization of fragments by configuring the inputs and outputs of combiner registers inside the GPU. This provided more flexibility in shading, allowing more realistic lighting based on more complex physical models, and enabling developers to use effects such as bump mapping in real-time rendering for the first time. However, the way in which configurable functionality was exposed was at times complicated and unintuitive, and based in part on vendor-specific extensions. This required client applications to provide multiple render paths for different hardware, each utilizing the functionality available to a specific set of GPUs.

2.1.3. Programmable rendering pipelines

The next leap forward in graphics hardware saw the introduction of programmable GPUs, where vertex and fragment processing is performed by dedicated programmable subprocessors with their own instruction sets. Through Shader Model version 1.x in Direct3D 8.0, and vendor-specific vertex and fragment program extensions in OpenGL, developers were given the freedom to completely rewrite parts of the

rendering pipeline which reside on the server (GPU) side. This was achieved by submitting to the GL two programs, both written in dedicated assembly languages: one which replaced the vertex processing part of the pipeline, and another which replaced the fragment rasterization part.

Further advancements in GPU technology have permitted longer, more advanced programs to be written; this has been reflected in the two dominant graphics APIs. Shader Model version 2.0 and the High-Level Shading Language (HLSL) arrived with Direct3D 9, version 3.0 was added in a later revision, and version 4.0 will be central to Direct3D 10. Vendor-specific extensions to programmable functionality in OpenGL were combined and standardized by the ARB into ARB_vertex_program and ARB_fragment_program; version 2.0 of OpenGL added support for the OpenGL Shading Language (GLSL).

2.2. Current technologies

The advances in GPU hardware have continued apace, with designers and manufacturers taking advantage of new technology, ever-increasing GPU clock speeds and the parallel nature of rendering to keep the latest generation of graphics cards ahead of even the speed increases predicted by Moore's Law [Moo65].

Probably the most significant recent development has been the replacement of low-level assembler with dedicated shading languages, which are closer to the programming languages used to create code for CPUs, and therefore much more intuitive. Direct3D 9 introduced HLSL, and version 2.0 of OpenGL included new ARB-approved extensions for writing GPU programs using GLSL. A third language, Cg (C for Graphics), was developed by NVIDIA in parallel with HLSL, and can be used with both OpenGL and Direct3D. Sh, originally part of the SMASH project [MQP02], offers an alternative approach to manipulating GPU behaviour using high level scripts embedded in the application code.

HLSL, GLSL and Cg use a C-like syntax, familiar to most programmers, with implicit support for composite data types like floating-point vectors and matrices. Sh provides similar capabilities via its C++ types and operators.

2.3. A scripted approach to rendering pipelines

While the power and programmability of modern graphics hardware and GLs permits developers to create complex surface, pre-process and post-process effects, the logic required to implement these effects in a generic way extends far beyond the GL and GPU, into the client application itself.

Microsoft have created the FX file format, designed to package an effect, which uses the concept of techniques and passes. An effect can have one or more techniques, each of which targets specific hardware. Techniques have a number of passes which, when combined, produce the effect. Each pass holds the required state changes and GPU programs. Other required objects, such as textures and arbitrary values, are declared as effect-wide variables. Support for FX files is built into Direct3D, and is also provided to applications using either OpenGL or Direct3D through NVIDIA's CgFX framework [FK03].

FX files provide a basic way to modify rendering pipeline behaviour. However, FX files describe only static state changes, and on their own do not contain enough information to describe a full range of effects. In order to remedy this, Microsoft recently released a standard for adding a scripting component to FX files, in the form of Standard Annotation and Semantics (SAS). As SAS script is embedded in the techniques and passes of effects, complex chains of script are formed, with each script driving part of the effect it controls. Operating at a higher level of abstraction than the shading languages presented in section 2.2, SAS facilitates the runtime combination of individual scripts in order to define more complex and dynamic rendering effects and processes.

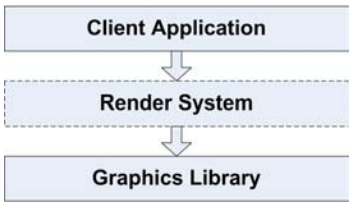

Figure 3: *The position of the render system layer with regard to the rendering pipeline.*

PirateHat|RS is a C++ library which provides a complete interface between a client application and the GL by wrapping the CgFX platform with an additional layer of abstraction (Figure 3). A complete abstraction, in this case, is a requirement; the RS must fully replace calls from the client application to the GL. It is infeasible for both the client application and the RS to share the responsibilities of interacting with the GL, due to the rendering methods employed by the RS, which will be elaborated on shortly. A complete abstraction also has other benefits. It permits close management and instancing of resources, and optimizations to the rendering process. Client applications primarily interact through a PirateHat|RS IRenderSystem interface object, which is implemented internally as a RenderSystem object.

PirateHat|RS operates on modified FX files, which are given the file extension *.cgfx, and referred to as shaders in PirateHat|RS terminology. The modifications required to convert an FX file to a CgFX file are trivial, mainly thanks to Cg being almost identical to HLSL, along with the underlying functional similarities between Direct3D and OpenGL. Some modifications are simple identifier replacements: substituting "PixelShader" for "FragmentProgram", for example. Other modifications are slightly more involved: matri-

ces must be converted from the row-major order expected by Direct3D to the column-major order OpenGL requires. It would be a simple exercise to write a program which automated the conversion of FX files into CgFX files; so far, however, we have relied on manual modification.

The extension of the render pipeline provided by PirateHat|RS relies on SAS scripting. As outlined above, SAS script is embedded into FX files through "Script" annotations, which provide additional semantic information to applications. Annotations reside in techniques and passes, and also in a special effect variable which carries the STANDARDSGLOBAL semantic. SAS script comprises a set of commands, each taking the form of an assignment, each instructing the RS to dynamically change the state of the GL, the state of the RS itself, or the state of the script execution. Each script is composed of one or more of these commands, which are executed serially by the RS. Several of the commands form the basis for loops, and conditional branching is also permitted through the use of a modified, n-ary ?: operator. Figure 4 contains some sample SAS script, taken from the standalone technique of a post-process negative-image shader.

```
string Script =
  "RenderColorTarget0=SceneTexture;"
  "RenderDepthStencilTarget=DepthBuffer;"
  "Clear=Color;"
  "Clear=Depth;"
  "ScriptExternal=color;"
  "Pass=p0;";
```

Figure 4: *An example SAS script taken from a negative-image shader. Note that the individual strings will be concatenated to form a single script delimited by semicolons. Interested readers may refer to the DirectX Standard Annotations and Semantics Reference [Mic] for more information.*

When client code requests through an IRenderSystem interface that a shader be loaded, the CgFX file specified is opened through the CgFX framework, and the SAS script is extracted and parsed. PirateHat|RS contains a class for each command, based on a common ICommand superclass interface, which perform the task of the command they represent by calling methods on the RenderSystem object. Command classes are created sequentially through a factory pattern as required by the ordering of the commands in the script, and each script is therefore composed of a vector of commands. Shaders are always one of two types. FX file terminology refers to these types as "scene" and "object"; PirateHat|RS terminology prefers the RenderMan-inspired labels "imager" and "surface". The former operate on the framebuffer before and after rendering of objects has taken place; the latter operate on objects themselves. Prior to rendering, the client application specifies which imager shaders it would like applied to the scene before and after rendering, and these are placed in an internal list. Each geometric object in the scene has a surface shader attached when it is created through the IRenderSystem interface.

During rendering, each imager shader in the list is visited, and the effect-wide script from it's STANDARDSGLOBAL variable is executed. This script will contain, among other things, a Technique command, specifying a technique to execute, and each of these will contain one or more Pass commands, specifying a pass to execute. The CgFX framework is used manage the setting of pass states.

In order to be executed, a script has a "frame" created for it, which contains a copy of the script, a command counter and other data related to the script's execution. This is exactly the method through which function calls are handled by traditional programming languages. Frames are pushed on to a call stack, and execution of the script at the top of the stack continues until the script ends, and it's frame is popped off the call stack, or until it specifies another script should execute through the Technique, Pass or ScriptExternal commands. In this way, each script can be thought of as a function, with all of the scripts together comprising a program which drives the PirateHat|RS rendering pipeline.

Once all imager shaders in the list have been executed, the geometric objects which populate the scene are rendered with their surface shaders (all of which also contain SAS scripts), and the call stack unwinds as all shaders finish executing. An interesting side-effect of this traditional stack-based execution (along with correct use of the ScriptExternal command in the scripts themselves) is that a high-level execution order of the scripts in the list is guaranteed: pre-process shaders execute first, followed by the rendering of scene geometry, and finally post-process shaders are executed. This is extremely useful to client applications, as properly-written imager shaders can be added to and removed from the list while only taking into account the respective positions of other shaders in the same category (i.e. pre- and post- processing).

2.4. Summary

A rendering pipeline is much more than just the familiar graphics pipeline. The client application itself is a large and important part, and must be written to control and take advantage of the hardware available. A scripted approach to rendering pipelines provides high-level abstraction, but retains the great flexibility required in rendering. Wrapping this scripted functionality inside a render system layer further enhances the abstraction, permitting an entire library of code to be reused, and providing a stable platform for client applications to be built upon and interact with. Graphics hardware and software will continue to evolve; render pipeline scripting seems set to become one of the best methods for keeping up with these

changes in a manageable and timely fashion. A number of game developers are already using render pipeline scripting to build powerful drag and drop tools (for example, see http://www.unrealtechnology.com/html/technology/ue30.shtml).

3. Data-driven applications

The rendering system described in Section 2 was initially intended to form part of a flexible, extensible framework for the visualization of geospatial scenes [JC06]; this framework has since been named the Fluid prototype. The data-driven manipulations and abstract descriptions of runtime behaviour provided by the PirateHat|RS library form a focal part of the Fluid prototype. The Fluid prototype is a plug-in based architecture, which makes use of a collaborating collection of dynamically linked modules in order to provide runtime behaviour. The configuration and composition of these modules is driven by a high-level XML-based composition language, which incorporates a novel object-oriented type system in order to provide additional flexibility during application deployment. Furthermore, we emphasize the use of the data-driven methodology described above in both plug-in implementations, and when (dynamically) configuring the application's runtime behaviour as a whole. We focus on facilitating the use of high-level data manipulations to determine the application's behaviour on multiple scales: from altering the functionality of individual plug-ins, through the formation of more complex behaviour via plug-in associations and collaborations, to the configuration of application sub-systems.

The Fluid prototype's design has evolved to incorporate themes from component based software development; the definition of its component model has been generalized to support a wide range of component types, and its composition language has been improved to provide additional flexibility, extensibility, and robustness to component software developers. Section 3.1 provides an overview of the Fluid project's current status, while Section 3.2 outlines our future plans to re-implement the rendering system described in Section 2 within the context of the improved Fluid system.

3.1. Fluid: A component framework

Fluid consists of a component framework, component model, and composition language, with a focus on providing an environment for the development of data-driven component software. Figure 5 gives an illustrative overview, while the following paragraphs provide a brief description of Fluid's constituent parts.

Fluid's component framework is a generalization of the Fluid prototype's high level architecture. As described elsewhere [JC06], the Fluid prototype presented two customization points for plug-in based extension: developers could replace entire subsystems in order to modify functionality

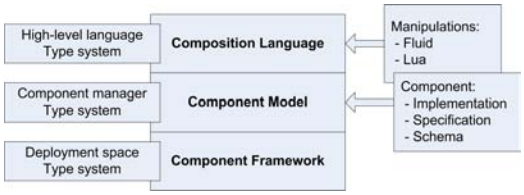

Figure 5: *An overview of Fluid's architecture, displaying its three tiers, their respective roles, and main client interactions. Note that the Component Framework provides functionality to the higher-level Fluid tiers, and does not provide a substantial client interface.*

at a coarse level of granularity, and new subtypes of scene object facets could be introduced in order to provide custom functionality at a finer scale. By contrast, the Fluid component framework provides a uniform deployment space for components of any scale, as long as their design adheres to Fluid's component model. Fluid's uniform deployment space was originally intended to mimic the concept of namespaces in popular object-oriented languages such as C++ [Str97], although its current inception is more closely modelled after Lua's tables [Ier06], and provides a hierarchy of named value associations. Fluid thus provides a hierarchical, dynamically typed, globally accessible storage location in which components may be deployed, configured, interconnected and manipulated, with the flexibility and expressiveness of Lua, a popular scripting language.

Fluid's component model builds upon a common design from component based development. In order to communicate with the Fluid framework, component implementations must first make use of a given component interface. The component interface is used to create and manage component instances; as part of its focus on data-driven application functionality the component model also configures a component's runtime behaviour via its interface. The component interface also provides two facilities for component communication and collaboration: a procedure binding system, and an event subscription system. In the case of procedure connections, a component implementation is able to expose its procedures to the Fluid framework's deployment space, and form local proxies to procedures belonging to other components. The event subscription system allows components to expose event types to the deployment space, and form subscription callbacks to other components' event types. The Fluid component model thus builds upon the facilities of the component framework in order to allow for component instantiation, configuration, and collaboration, within the context of a flexible deployment space.

Fluid's composition language forms a major contribution of the Fluid project. Building upon the Fluid prototype's configuration files, as well as concepts from relevant research in composition languages [Bir01, TC03], the

Fluid framework leverages XML as a language for component software composition. An XML based composition language provides a number of benefits, including platform independence and a hierarchical structure that maps well to the Fluid framework's deployment space. Furthermore, XML serves as a suitable basis for the declaration of Fluid components and their configuration and wiring, and provides a means for composition document validation through the use of XML schema. Fluid's composition language includes mappings for a number of XML types, such as booleans, floating-point types, integral types, strings, optional values, sequences and choices between two or more types; these supported XML types, and additional types based on them, are used to facilitate type-safe communication between the Fluid framework, component compositions and their configurations, and component implementations at runtime.

Fluid's composition language is used to form coherent component software applications from collaborating component deployments. A Fluid component deployment consists of three types of file: one or more implementation libraries (currently Windows dynamically-linked libraries) implementing the component's runtime behaviour; a component specification, which provides a name for the component, and describes the component's required and provided procedure and event exposures; and finally an XML schema specifying how the component's behaviour may be configured, if at all. A Fluid developer introduces component deployments to the Fluid framework via composition files, whereupon the components become part of the framework's integrated type system. Further composition files may instantiate defined components and assign them to Fluid's deployment space; a component's configuration (if any) will be validated by its deployment's configuration schema, and any procedure or event connections described by the deployment's specification may be used to form links with compatible exposures made by other component instances. In a similar manner to existing component frameworks, Fluid components may thus be added to the framework's runtime environment, instantiated, configured and manipulated to form collaborating behaviours.

Finally, the Fluid composition language includes an additional level of abstraction to component software developers. Forming a novel link between computer games technology and the composition language field, we introduce an object-oriented, data-driven type system that operates at a macro level to allow for expressive manipulation of component compositions. While Fluid components may be instantiated and assigned to the deployment space alone, they may also form part of much larger component hierarchies, where the particular selection, hierarchical arrangement, and configuration of collaborating components forms a *composition type* at a larger scale. For example, a component software developer may associate a combination of sibling Appearance and Geometry components as a *renderable* that may be passed to a compatible component providing rendering capabilities. Fluid's composition language supports the use of such composition types. Developers may define a given hierarchical arrangement of component instantiations, configurations, and wiring, and associate them with a *composition type name*, which may be used thereafter to instantiate a copy of the associated arrangement. Furthermore, the composition type is introduced as a prototypical instance in a *types* section of the Fluid framework's deployment space. Once defined, a composition type may be used to create additional instances of its component arrangement; alternatively, it may form part of an object-oriented inheritance relationship, whereby child composition types may inherit or override portions of its parent type's definition.

Fluid consists of three tiers: the Fluid framework, which provides a flexible deployment space alongside an XML-based type system; the Fluid component model, which allows a wide range of component implementations to interact with Fluid's other constituent parts; and a composition language, which supports a number of high-level abstractions and manipulations to be described. Together, these three tiers facilitate the development of expressive component software compositions, with a focus on driving runtime application behaviour using high-level abstractions and data-driven methods.

3.2. Fully data-driven rendering

The Fluid prototype incorporated two subsystem plug-ins that were closely tied to the geospatial visualization application domain: a scene system, which was responsible for the management and manipulation of objects representing the scene's contents; and a render system, which was responsible for providing a visualization pipeline to the other subsystems in the application. Using an early revision of Fluid's composition language, application developers could describe a given geospatial scene as a combination of small-scale plug-ins and their configurations. The Fluid prototype also incorporated the object-oriented data-driven manipulations that Fluid's composition language applies to its composition types. The scene system was required to maintain a scene graph of objects in the scene, and to update their runtime state by triggering the behaviour provided by the objects' constituent plug-ins. Objects with visual representations included geometry and appearance plug-ins, and would be passed to the render system once per frame for rendering.

We expect application developers to visualize geospatial scenes via the Fluid framework using similar techniques to those described above. However, the Fluid framework's application domain is not restricted to geospatial visualization, and it provides no native facilities for scene management, and no default rendering pipeline. In order to validate the Fluid framework, we aim to replace the Fluid prototype's fixed architecture with a collection of corresponding components and component configurations. The development of a PirateHat|RS component is expected to form a substantial

part of the resulting visualization application. Section 3.2.1 gives a high-level outline of the methods we intend to use in order to integrate PirateHat|RS as a Fluid component, and Section 3.2.2 describes how other component deployments may interact with it.

3.2.1. A PirateHat|RS Fluid component

Our first task will be to develop a wrapper class that implements the Fluid component model interface, while acting as a thin proxy to the PirateHat|RS library. The resulting Fluid component, depicted in Figure 6, will expose the PirateHat|RS library's interface to the Fluid component framework, and will be responsible for forwarding procedure calls from collaborating components to their PirateHat|RS equivalents. The Render System component's specification will describe the procedures exposed by the Render System component, as well as the Fluid types required to communicate with it. For example, the specification may include definitions for a Vertex type consisting of a sequence of floats, a Geometry type consisting of a sequence of Vertex types, an Appearance type that is configured by an Effect type instance (see below), and so on. While the use of XML-derived types will allow the Fluid framework to support many interface types such as these, there will be a number of types with their own behaviour that must be defined as related component types; we describe the Effect component type as one such type below. The Render System component will also require a corresponding configuration schema that describes its configuration parameters.

An excellent example of the component types we plan to develop is the Effect component type, which will represent a data-driven rendering effect that can be attached to geometry to support a wide variety of object appearances, and also applied to the scene as a whole to provide a number of pre- and post- rendering manipulations. The Effect component will take either a complete high level SAS script or a path to an SAS script file, which will be validated using an appropriate XML schema. Upon instantiation, the Effect component will make use of the PirateHat|RS library to parse the given effect script, and to form an internal representation that can be communicated to the Render System component. After parsing the SAS script, the Effect component will also decorate its own instances with child elements corresponding to the script's various parameters.

3.2.2. Using the Render System component

The following paragraphs present an intended usage scenario for a PirateHat|RS component instance as part of a Fluid visualization application. Figure 7 provides a complimentary illustration to the description given below.

In order to make use of the PirateHat|RS library as part of a component software visualization application, the component developer will first create an instance of a Render System component, with an appropriate configuration, and deploy it to the Fluid framework's deployment space. This Render System component instance will be analogous to the Fluid prototype's singleton rendering subsystem, and will provide the visualization application's rendering pipeline functionality.

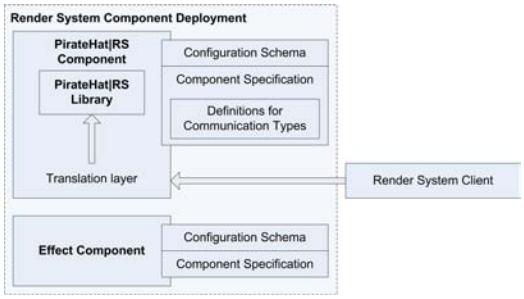

Figure 6: *An illustration of how the PirateHat|RS library is to be wrapped as a Fluid component deployment, consisting of multiple component implementations and interface type descriptions.*

In a similar vein to the Fluid prototype's scene descriptions, the Fluid framework provides a high level of abstraction and expressiveness for defining geospatial scenes as a collection of configured component instances. However, the Fluid framework will not rely upon a dedicated system for managing such scenes; instead, composition types representing objects in the scene will be assigned to the Framework's hierarchical deployment space, while a collection of visitor components will be used to manage and manipulate them.

Application developers will describe visible geospatial objects as composition types including an associated pair of Geometry and Appearance components. One or more visitor components will be responsible for collecting visible objects and communicating their Geometry, Appearance, and other constituents to the Render System component, which will then render the object's visible representation as part of next frame.

Application developers will also be able to invoke Render System components directly in order to attach and detach pre- and post- rendering effects. Alternatively, effects may be added and removed by procedural calls made by other components, they may form part of the Render System component's configuration, or they may be manipulated via Lua scripts either forming part of current composition, or sent to the Fluid Framework by the user.

3.3. Summary

The Fluid prototype is currently being redeveloped as a component framework. We aim to validate the transition, from a flexible albeit ultimately fixed architecture, to a generalized component environment by demonstrating Fluid's capabil-

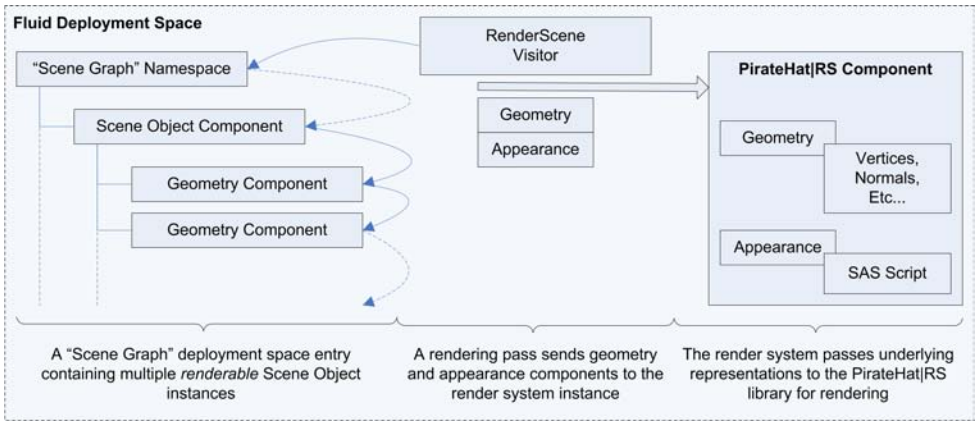

Figure 7: *A depiction of the PirateHat|RS component's intended use as part of a Fluid composition.*

ities as a visualization component application for geospatial data. In order to achieve this, we aim to wrap a number of the Fluid prototype's subsystem implementations as Fluid components. In particular, the Fluid prototype's render system implementation will be replaced by the PirateHat|RS library, which continues the data-driven focus of the Fluid project by providing a high-level scripting interface to a high-performance rendering pipeline. As part of the Fluid project, we aim to develop a component software visualization application that makes full use of Fluid's expressiveness, flexibility and extensibility, while facilitating highly detailed visualizations of geospatial scenes at interactive frame-rates.

4. Conclusion

Modern rendering systems are moving toward higher-level descriptions for object appearances and special effects that were previously only available to off-line renderers. State of the art rendering APIs are already introducing the next layer of abstraction, in the form of a scripting system that forms part of effect scripts themselves. The incorporation of such scripts, and the use of the bindings and exposures they provide, indicate a movement toward higher level, data-driven manipulations of next-generation rendering pipelines.

The Fluid framework places a similar data-driven focus on the composition and runtime behaviour of component-based software. As described above, we support the composition and collaboration of small-scale components, while also providing a flexible and expressive OODDP composition language that can be used to make logical manipulations at high-levels of abstraction.

In this paper, we have described a Fluid approach to the growing trend in data-driven methodologies. With the inclusion of a data-driven rendering system, we aim to introduce a novel and unique level of flexibility and expressiveness to visualization applications for geospatial data.

References

[Bir01] BIRNGRUBER D.: CoML: Yet another, but simple component composition language. In *Proceedings of First Workshop on Composition Languages* (Vienna, Austria, Sept. 2001).

[FK03] FERNANDO R., KILGARD M. J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[Ier06] IERUSALIMSCHY R.: *Programming in Lua, Second Edition*. Lua.org, 2006.

[JC06] JONES A., CORNFORD D.: A flexible, extensible object oriented real-time near photoreaslistic visualisation system: The system framework design. In *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling* (2006), Riedl A., Kainz W., Elmes G., (Eds.), Springer-Verlag, pp. 563–579.

[Mic] MICROSOFT: DirectX Standard Annotations and Semantics Reference. http://msdn2.microsoft.com/en-us/library/bb173004.aspx.

[Moo65] MOORE G. E.: Cramming more components onto integrated circuits. *Electronics 38*, 8 (1965), 114–117.

[MQP02] MCCOOL M. D., QIN Z., POPA T. S.: Shader metaprogramming. In *Proceedings of the 17th Eurographics/SIGGRAPH workshop on graphics hardware* (New York, Sept. 1–2 2002), Spencer S. N., (Ed.), ACM Press, pp. 57–68.

[Str97] STROUSTRUP B.: *The C++ Programming Language*, 3 ed. Addison-Wesley, 1997.

[TC03] TANSALARAK N., CLAYPOOL K. T.: XCompose: An XML-based component composition framework. In *Proceedings of Third International Workshop on Composition Languages* (2003).