Eurographics Italian Chapter Conference (2007)
Raffaele De Amicis and Giuseppe Conti (Editors)

Challenges for modern Scene-Graph Systems

Wolfgang Müller-Wittig† [1]

Gerrit Voß‡ [1]

[1]Centre for Advanced Media Technology
Nanyang Technological University
Singapore

Abstract

Current scenegraph systems, especially systems used to build general purpose virtual reality systems, are trailing game engines and similar specialized systems in terms of the adaption of new rendering methods like the different real-time shadow algorithms. This paper analysis the fundamental OpenGL state abstraction layers present in current scenegraph systems with respect to their influence on the adaption of new and complex rendering algorithms.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computing Methodologies]: Computer Graphics Methodology and Techniques I.3.7 [Computing Methodologies]: Computer Graphics Three-Dimensional Graphics and Realism

1. Introduction

Driven by the recent advances in computer graphics hardware, previous too complex and time consuming algorithms, like enhanced shadow maps [MT04] or parallax bump mapping [Wel04], became available for the use in real time graphics environments. But upon closer inspection it becomes clear that many of these techniques are generally deployed in game engines and similar, highly specialized environments. In comparison their occurrence as part of general purpose virtual reality systems is more sketchy. These algorithms are available but usually they are integrated as add ons, build to suit a narrow application environment and only of use with datasets specifically designed to cope with the given limitations, like number of textures and which of the available texture units to use. The core difference, which leads to the observed discrepancy in usage, is the difference in knowledge about the scenes to be rendered. Game engines are build against a predefined set of models and effects, general purpose virtual reality systems on the other hand are build to serve a wide variety of application environments, and hence need to support a wide variety of models and effects. As a consequence game engines can optimize their underlying graphics structure implementation according to the requirements given by the models and effects used. Scene graphs, which provide the foundations for common virtual reality systems, in contrast have to provide more general abstractions and have to balance the requirements given by the wide variety of application requirements.

An integral part of any scene graph system is to provide an abstraction for the underlying graphics hardware, especially to hide the complexity needed to optimally feed the deep pipelines of modern graphics hardware. This is achieved by utilizing a low-level API, in general OpenGL [SAFL99] [SALB04] is used for general purpose cross platform systems. On top of this API several layers are build. Shortcomings in these layers as well as overall missing layers are responsible for the observed lack of general support described above. Of particular interest are the layers abstracting the appearance attributes because here the major substantial progress has be made, whereas as the OpenGL primitive interface stayed nearly constant.

† e-mail: mueller@camtech.ntu.edu.sg
‡ e-mail: vossg@camtech.ntu.edu.sg

© The Eurographics Association 2007.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

In general the part of a scene graph system which is responsible of preparing the scene for processing by the graphics card is called rendering backend. It has the responsibility to convert the scenes as given by the user into the final image. The rendering backend does so by traversing the scene graph, collecting the visible geometries and passing them on to the underlying hardware. During this process it is the responsibility of the backend to find an order of the geometries which allows the best utilization of the given hardware.

In the following we will describe the abstraction layers found in current OpenGL based scene graph systems.

2. Challenges for current Systems

Graphics hardware as abstracted by OpenGL and thus seen by scene graph systems can be viewed as a state machine by which the primitives are processed and transformed to form the pixels of the final image. The state variables which describe how the primitives are drawn include light and material parameters, textures and blending, and customized shaders.

2.1. OpenGL Abstraction

This layer has two main tasks. The first, which is visible to the programmer, is to provide access to the state variables. Modern scene graph systems like OpenSG [RVB02] or OpenSceneGraph [OB05] group similar variables together to form logical clusters, for example all texture, material or light information are grouped together. These are called StateAttribute (OpenSceneGraph) or StateChunk (OpenSG), where a set of them is called State and fully describes how the associated primitives will be drawn. The benefit of grouping them in a logical way is that the programmer has a better understanding which variables belong together and must be manipulated in a synchronized way.

The second task relates to the way modern graphics hardware is build. The OpenGL graphics pipeline is not only put into hardware but in order to achieve real-time performance for complex models it has to be pipelined, so that different parts of the rendering pipeline work independently of each other, like independent vertex and fragment stages. Furthermore each of the stages them self contains multiple units working in parallel, like the 24 fragment units of a NVidia Geforce 7800 GTX. To be able to build these parallel units and the pipelines in hardware they have to be rather simple. As a number of different pieces of geometry may be in operation at any given time, changing the state, and thus the way the units operate, has to be synchronized with the data flow through the pipeline. As simple hardware cannot do that the pipeline has to be, at least partially, empty before a state change can be applied. After wards it has to fill up again before the nominal performance can be reached. If these state changes happen too often it might not be possible to fill up the pipeline completely, and in the worst case waste all the benefits of the pipelined architecture.

But not all problems with state changes can be attributed to the simplicity of the hardware. Some are caused by limited resources that have to be managed, like texture memory and texture caches. Texture are usually stored in special, high performance memory, the texture memory, on the graphics board itself. Furthermore the active areas of the textures are stored inside a special cache, the texture cache, as close as possible to the processing unit. Like any other multi-level cache system changing its contents to often will decrease the overall performance. Thus the second task of the scene graph system is to take care of managing the state. Specifically to minimize the number of state changes in order to avoid pipeline stalls and cache pollution.

Scene graphs therefore include state sorting facilities as part of their draw operation. Current systems like OpenSceneGraph, OpenSG or OpenGL Performer [RH94] identify a State by a unique identifier and use a bucket sorting strategy to group primitives by their associated state. This way primitives sharing the same state are drawn directly after each other without the need of a state change. The concept of using a unique state identifier for the sorting procedure was based on the observation that each StateChunk occurred exactly once inside the State and thus the primitives where likely to share the complete State instead of single StateChunk and thus sorting by chunks would be an nearly never used overhead. Unfortunately this assumption broke down with the appearance of multi textures and shader programs from OpenGL version 1.3 on. Now it was common to share one or more textures between States or use the same shader program and only vary the shader parameters between different primitives as the multi texture example in figure 1 illustrates.

The OpenGL state of each of the tunnel segment building blocks (ceiling, floor, left and right wall) contains a color texture and a light texture (see figure 2). Blended together they allow the simulation of the complex lighting environment. In order to create a complete tunnel n segments are joint together. Due to the varying light textures each of the n floor elements, for example, is seen by the existing state sorters as having a unique, independent state even if all floor elements share the same color texture and other state settings. This inability to utilize information about shared StateChunks during the sorting process may result in the following worst case rendering order. First each element of segment i (floor, ceiling and walls) is drawn followed by each element segment $i+1$. This segment by segment processing continues until segment n. Thus activating each of the color textures n times. In comparison a StateChunk aware sorter will be able to first draw all floor elements followed by all ceiling, all left and

© The Eurographics Association 2007.

all right wall elements, activating each color texture exactly one time.

Figure 1: *Multi texture environment.*

Figure 2: *Color and light textures.*

The state based approach is the major limitation of this first layer.

2.2. Multi Pass Abstraction

This abstraction is used to draw a single set of primitives repeatedly with different States in the specific order given by the order of the States.

This method is called multi pass rendering where pass n is defined as drawing the primitives with $n-th$ State. Because of the specific order of passes required the primitives can not be sorted into the existing set of buckets according to their state used for the respective pass. The simple solution systems like OpenSG use is to introduce a so called MultiState which instead of StateChunks contains the States for each of the n passes. For sorting purposes the MultiState is treated like any other State with its corresponding bucket into which the primitives can be sorted. The drawback of this approach is that if the States of individual passes are shared between different MultiStates the fact that they are contained in different buckets requires them to be activated for each MultiState instead of once for each pass.

2.3. Multi Stage Abstraction

Not only are sets of primitives drawn repeatedly using multiple States but whole subtrees are traversed multiple times in a specific order. Furthermore temporary results like color or depth texture have to be stored in order to reuse them in later stages.

Again the simple bucket sorting approach of the rendering backend can neither handle the ordering constraints of the traversals nor provide storage for the temporary results. A simple example for multi stage algorithms are shadow maps [SKW*92], figure 5. This algorithm creates real-time shadows by first rendering the scene as seen from the light position (Stage 1, figure 3). The resulting depth buffer is stored in a texture and reused later. The next stage (Stage 2, figure 4) renders the scene as seen by the camera but only the ambient color terms are active in order to generate the shadow colors. The last stage (Stage 3, figure 5) draws the scene from camera perspective with diffuse and specular lighting enabled. The result from Stage 1 is used to decide whether a pixel is lit or inside the shadow.

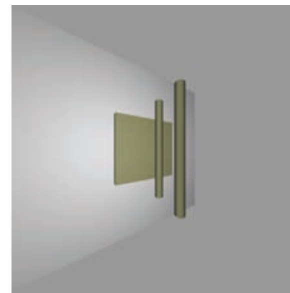

Figure 3: *Shadow mapping stage 1 (color buffer).*

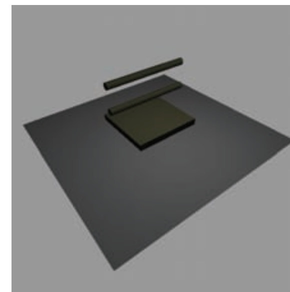

Figure 4: *Shadow mapping stage 2.*

As the rendering backend was not capable of handling multiple stages, current systems offloaded the logic needed to different areas. OpenSG for example uses per stage viewports in order to provide the correct ordering and the possibility to store temporary results. And by doing so it splits each stage into an independent rendering traversal which has no knowledge about any other. This split makes it difficult to pass information between different stages. Often it is also found that one stage is completely executed, including all OpenGL calls, before the next stage is processed. One set of problems with this approach arise from the fact that the processing of the stages is taken out of the one integrated

© The Eurographics Association 2007.

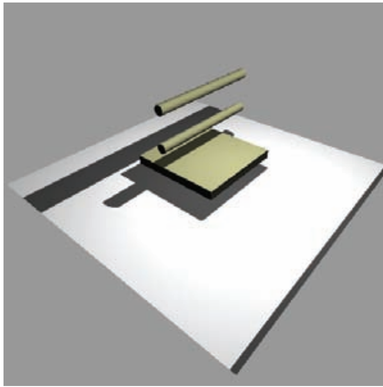

Figure 5: *Shadow mapping stage 3.*

rendering traversal. Thus information only available during the rendering traversal, like accumulated transformations or bounding volume visibility in dynamic scenes, is lost and must be recreated. This introduces unnecessary overhead by using additional traversals in order to recreate the lost information. Furthermore it is difficult to decide if a particular stage has to be executed or not. If, for example, the object which uses the temporary result from a stage is not drawn because it is culled away the stage creating the result should be skipped too. As the stages are independent and each is executed before the next stage is started there is no easy way to prevent an earlier stage from running once it is detected that the result is not needed in a later stage.

2.4. Shading Languages

With the introduction of shading languages OpenGL lost a key feature current scene graph systems heavily relied on, the ability to distribute the actual state changes. Within current systems the state changing OpenGL calls were distributed on two levels. At the lowest level the OpenGL calls are grouped and abstracted into individual objects, for example the OpenSG state chunks described in 2.1. The abstraction objects (StateChunks), as they can be defined and created anywhere in the scene graph, are collected during the rendering traversal and each geometry is associated with a collection of them. If, during the geometry processing, consecutive collections contain changes the minimum amount of OpenGL calls are generated as the changing StateChunk pairs are evaluated individually.

Shading languages introduced another layer into this process. As described above prior to the arrival of shading languages each individual StateChunk would emit the required OpenGL calls directly. With shading languages a unique shading program has to be generated from the collection of StateChunks, in addition to the still needed state changing OpenGL calls. As the regeneration of the shading program is to expensive to repeat it every frame there is the need to provide an intermediate caching structure. This structure should be able to exploit frame to frame coherences and at same time detect valid changes that trigger a required rebuild of the shading program.

2.5. Out of Order Rendering

The ever increasing complexity of geometric models requires optimizations already at the initial traversal stage of a scene graph based rendering system. An important consequence of the introduction of optimizing traversals, as for example implemented for occlusion culling, is the loss of a fixed traversal order. As a consequence any proposed approach to the previous mentioned challenges should integrate well with a dynamic change in the traversal order from frame to frame. In particular intermediate structures should be light-weight enough to be stored and restored with the current traversal state.

3. Conclusion

In this work we have analyzed the abstraction layers as they are found in current, OpenGL based, scene graph systems. Our analysis focused on challenges which are likely to impact the adaption or development of newer real-time computer graphics algorithms, like the class represented by enhanced shadow map methods. In particular we were interested in the impact in terms of performance penalty and flexibility of use.

All the shortcomings and inflexibilities described actually do not prevent modern multi pass or multi stage algorithms to be used with the current rendering backends of modern scene graph systems. But they come at a price, both in terms of unnecessary overhead in runtime and implementation complexity. Often this price is too high, so that algorithms are either not adapted or adapted in a way that is tailored to a specific scenario in order to prevent the performance penalties. Thus making it hard to use the tailored implementation in general scenarios.

Currently approaches for first set of challenges have been proposed [VR06], but especially in the shading language area further research is needed.

© The Eurographics Association 2007.

References

[MT04] MARTIN T., TAN T.: Anti-aliasing and continuity with trapezoidal shadow maps. In *Proceedings of the 2004 Eurographics Symposium on Rendering* (2004), Eurographics, pp. 153–160.

[OB05] OSFIELD R., BURNS D.: Openscenegraph, reference guide. Online, http://www.openscenegraph.org, 2005.

[RH94] ROHLF J., HELMAN J.: Iris performer. a high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of SIGGRAPH 1994* (1994), Glassner A., (Ed.), Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, pp. 381–395.

[RVB02] REINERS D., VOSS G., BEHR J.: Opensg - basic conepts. OpenSG 2002 Workshop, January 2002.

[SAFL99] SEGAL M., AKELEY K., FRAZIER C., LEECH J.: *The OpenGL graphics system: A specification (version 1.2.1)*. ftp://ftp.sgi.com/opengl/opengl1.2/opengl1.2.1.pdf, 1999.

[SALB04] SEGAL M., AKELEY K., LEECH J., BROWN P.: *The OpenGL graphics system: A specification (version 2.0)*. http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf, 2004.

[SKW*92] SEGAL M., KOROBKIN C., WIDENFELT R. V., FORAN J., HAEBERLI P.: Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH 1992* (1992), Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, pp. 249–252.

[VR06] VOSS G., REINERS D.: Towards a Flexbile Back-End for Scenegraph-Based Rendering Sytems. In *Proceedings of Graphite 2006* (2006).

[Wel04] WELSH T.: Parallax mapping. In *Shader X3. Advanced Rendering with DirectX and OpenGL*, Engel W., (Ed.). Charles River Media, 2004, pp. 89–95.

© The Eurographics Association 2007.