A concept for database oriented 3D graphics engine infrastructure

V.A. Frolov[1,2], V.S. Sangarov[3], V.A. Galaktionov[1]
[1]Keldysh Institute of Applied Mathematics (Russian Academy of Sciences), Moscow, Russia;
[2]Moscow State University, Moscow, Russia;
[3]Gubkin russian state university of oil and gas, Moscow, Russia

Integration layer between digital content creation software (DCCS) and rendering software in a form of specialized database is proposed in this paper. In our approach, we focus on providing fast 3D-scene updates, ability to work with large digital assets (not fitting into memory), importing and exporting arbitrary parameters, serialization, convenient debugging tools and distributed rendering. Such database can be used as means to integrate different rendering engines with DCCS and also to transfer data between different DCCS.

Keywords: *software architecture, rendering, process interoperation.*

1. Introduction

Fast growth of computing power in the past 20 years gave rise to new research and industrial fields. Data volumes have grown from megabytes to gigabytes and performance has raised from megaflops to teraflops. But it's not only about the numbers, more important is how it influences interaction with the user. As technology matures, it generally becomes more accessible and friendly for the users. This also holds true for digital content creation. Since rendering is inevitably tied with such applications, rendering engines' developers need to answer the challenges created by the need to improve user experience. In our work we aim to answer these challenges with a database oriented approach to rendering engine infrastructure design.

1.1. Modern digital content creation process requirements

1. *Interactivity* of digital content creation process (WYSIWYG paradigm). Working in the interactive and non-interactive modes can be thought of as editing a text document in MS Word and in LaTeX. The latter has certain advantages, however, creating a document in MS Word is faster.
2. *No restrictions on memory*. The whole 3D scene may not fit into RAM of a single computer, but content creation process and rendering (at the very least in preview mode) should not have delays [1].
3. *Parameters variability* and extensibility of both DCCS and rendering systems. Different projects may require different computer graphics algorithms to be used (like for hair and fur). It's quite common for post-production studios to create their own rendering plugins both for DCCS and renderers [2].
4. *Serialization*, import and export. Artists reuse variety of digital assets from their previous works or content repositories and therefore it's necessary to import and export everything. Moreover, in the visual effects and animation industry pipeline content needs to be passed between different software back and forth [3, 4].
5. *Debugging* and testing. While working on complex projects it is almost inevitable for some errors and bugs to appear. Some of these errors appear only for a certain order of actions and thus can be hard to reproduce. To isolate and fix these errors, it's important to track changes in the scene.
6. *Distributed rendering* [4] and explicit transfer of changes. Changes made in the DCCS should be visible as the same for every other computer participating in rendering. It's unacceptable to send the whole scene over the network, we need to *track and transfer changes only*.

2. Previous work

2.1. Wavefront OBJ, Filmbox FBX and others

The most basic way to transfer digital content between applications is to use binary or text files with strictly defined format. For example, simple but limited OBJ file format [5] or more complex and flexible FBX [6]. In the case of strictly defined file formats there's always a trade-off between flexibility and complexity. The problem is that it's impossible to predict what features DCCS or rendering engine developers will need in the future - what parameters would materials have, will there be new forms of geometry and light sources, etc.

2.2. OpenCollada and Alembic

OpenCollada is a step forward compared to rigid file formats. The main difference is that OpenCollada defines only the standard for storing objects (called «COLLADA») in XML [7] and allows developers to *add new parameters*. It's possible since OpenCollada is an open source project and uses XML descriptions both for DCCS and rendering engine. More recent format with similar ideology is **Alembic**, presented on SIGGRAPH in 2011 [8]. The main focus of Alembic is to efficiently store complex animation - key framed or a product of any kind of simulation (fluid, cloth, etc.). However, support for materials in Alembic is still not available [9], so one needs to use additional file formats like MaterialX [10] to transfer such data.

2.3. Limitations of «just files»

In the best case, only 2 or 3 of the requirements for modern digital content creation can be met with common file formats (most likely parameters variability, serialization and debugging/testing). To move forward there needs to be a way for working with large scenes, tracking and logging changes, and, most importantly,

prompt transfer of changes between different software (without rewriting the whole file).

2.4. DRAM and DLL plugins

The opposite of using files for the integration purpose is to transfer data structures through memory or shared memory via serialization as in [11]. Dynamic Loading Library (DLL) plugins closely integrated with DCCS are worth mentioning specifically. These plugins usually directly call virtual functions of DCCS and thus don't import anything. For example, rendering plugin can call a function called $Shade$ for material evaluation. While possessing all of the advantages of being closely integrated, such approach limits the performance, scalability and does not guarantee correctness. You can never be sure that function $Shade$ is implemented sufficiently effective and that it does *what render developers want* it to do. Moreover, this type of interaction between rendering system and DCCS strongly depends on DCCS in question [12] and that in turn complicates debugging process and integration into any other DCCS.

Both of these approaches, that we just described, are simple and usually provide sufficient performance. However, they can meet only two requirements - interactivity and parameters' variability.

2.5. SDB

The comprehensive explanation of some of the requirements we listed in the beginning of our paper can be found in [4]. Relying on their experience, authors of [4] conclude that rendering engine should exist in the form of an API to the Scene Database (SDB) and describe functionality of such database. The implementation was not specified thus leaving an open space for future research.

2.6. USD

Universal Scene Description (USD) developed by Pixar was revealed to the public in 2016 [3]. This technology was primarily designed to address issues arising when different artists work on digital content for a big project like an animated feature film. USD assumes that digital content is created by *different people in different applications*.

Basically, USD is represented by a set of files in a JSON-like format which are organized together by the means of references and compositions. For example, a composition of a location scene and a scene with animated character. This approach allows group of artists to work efficiently and simultaneously on a film shot components which are called layers and can be combined into a final result or reused in other shots. The resulting hierarchy of digital content created in various software allows tracking the history of scene creation and making several versions of the whole scene or it's components. Also, by using references and delayed loading USD makes it possible to work with large scenes [3].

While USD satisfies a lot of requirements mentioned in the beginning of this article, it's difficult to use this technology directly as an integration layer with rendering system. The main problem is the absence of mechanism for fast updates. USD does not use global identifiers for objects and *with every change* one needs to recursively check all of the file hierarchy. Moreover, the fact that any file can be overwritten complicates the implementation of distributed rendering, since some of the scene files on different rendering nodes can have different versions.

2.7. Multiverse

Product called Multeverse being developed by J-Cube [13] was initially designed to tackle the problem of loading large scenes (alembic files in particular) and working with them in Autodesk Maya. To achieve this Multiverse takes over the job of loading and otherwise accessing geometry from the editor and implements delayed loading. The scene data is streamed directly into the rendering system or into OpenGL-based interactive viewport of the editor (i.e. Maya).

The downside of this approach is that Multiverse becomes too closely integrated with DCCS and *reimplements* many of it's functions. If system like Multiverse is *already integrated* with a certain DCCS then it could be used as integration layer between the DCCS in question and a rendering system. Otherwise, amount of work required to integrate Multiverse with a DCCS can be tenfold more than any other integration approach. Currently, Multiverse is available only for Maya and Katana, and does not support 3Ds Max, Blender or any of the CAD/CAM systems like Rhino or CATIA.

2.8. Bunsen

Project Bunsen being developed by The Foundry [14] is a cloud-based software for assembling the final scene from different digital assets which supports import from various DCC and CAD/CAM software.

Bunsen provides users with the ability to process imported data in the most suitable for the task at hand way through a *data processing node graph*. The nodes in this graph can perform variety of operations such as converting splines into polygons, polygon mesh optimization, materials assignment, applying level-of-detail (LOD) techniques. If any of the assets used in a particular scene is changed, user needs to re-export it. Bunsen will recognize that one of the asset files has changed and will reload it and execute all nodes dependent on that asset again. The scene data is then prepared and streamed into selected imaging software – interactive or «offline» renderer.

Since Bunsen is in the stage of active development, many details about it are not yet known to the public [14]. Nevertheless, the announcement of such system shows that there's a demand for novel integration solutions oriented on user-friendliness and emerging technologies as we pointed out in the begging of this paper.

3. Suggested approach

Our concept can be thought of as a hybrid of object oriented database and a version control system, (like Git or

Subversion) with no-overwrite strategy of making changes. We believe that between the Editor and the rendering engine should exist special API. This API needs to fulfil the requirements listed in the beginning of the article in a simple and transparent manner (Fig. 1).

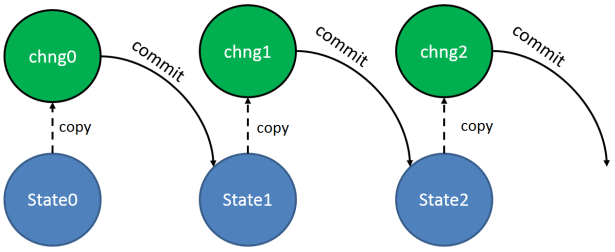

Fig. 1. Our intermediate layer (API) treats 3D content creation like working with source code – make changes and explicitly commit them.

A single state of the scene is a single XML file called «state_001.xml» in some directory «myscene». This file references several text or binary files in subdirectory «data» (usually geometry and textures). The internal XML structure is subdivided into «library» and «scenes». The library contains references to all external files in subdirectory «data» and describes materials, lights and camera. A single scene is just a list of geometry instances. Instance is a reference to geometry object with custom transformation matrix and material remap list (if one should have instances with different materials).

Neither XML files, nor geometry or textures (thinking of them as external files in subdirectory «data») should be actually saved to the hard drive. When the Editor passes them to the render, they are transferred through Operating System (OS) shared memory (Fig. 2). We will discuss this issue more specifically when we talk about «Virtual Buffer».

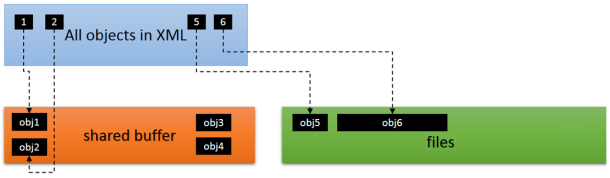

Fig. 2. New or changed objects are always placed in the OS shared memory cache.

3.1. Making changes

A distinctive feature of the proposed technology is explicit tracking and recording the changes. Our API has 3 methods for each object (geometry, material, light and others) – *Create*, *Open* and *Close* (like files in OS). *Create* method makes an empty object. The pair of *Open* / *Close* allows to change the object. When user code (from the Editor side) calls *Open* for some existing object, the copy of the object's XML parameters will be automatically created. Next, user can work with this copy, changing its XML parameters in RAM via pugixml [15]. When the work is finished *Close* method should be called. From this point on, the new state of the object is considered as «ready to commit», but it is still stored as a separate copy (Fig. 3, right).

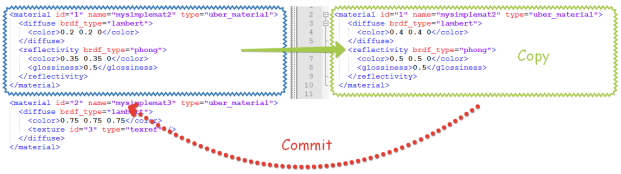

Fig. 3. User of our API can only change an XML copy of an object (right). After «commit» this copy will replace original XML description (left) and form new state file.

We can say that if the current state of the object is stored in the file «state_001.xml», then the new state of the object is stored in a separate file «change_001.xml» (Fig. 1). All these files do not need to be saved to the hard drive – they can be stored in RAM in some dynamic structures (used by the pugixml library in our case). The user may change any number of any objects (including changing one object several times). The file «change_001.xml» will contain only last changes for these objects. Thus it is important to note that file «change_001.xml» will not contain any information about objects that were not actually changed.

Finally, the Editor calls *Commit* to pass the new scene state to the renderer engine. The *Commit* operation creates file «state_002.xml» in which old objects from «state_001.xml» are replaced by their copies from «change_001.xml». It should be clarified that during the execution of the *Commit* operation, only new, modified objects and their XML nodes will be passed to the renderer to update their states inside render engine. Thus, the new render state will be consistent with file «state_002.xml», however, the state file itself is not analyzed. It should be mentioned that it does not matter in which order user changes objects. After *Commit* operation has been called, the API will pass all changes to the render in a fixed, well-known order. Thus the rendering engine developers may rely on fixed and *well-known* sequence of calls from our API.

3.2. Virtual Buffer

To handle big data (geometry and textures) we use a concept of virtual append-buffer with infinite size. The buffer can append linear data blocks to its end. But only last N Megabytes are put into RAM. The rest of the buffer is flushed to a hard drive as a set of chunks – binary files in *data* subdirectory (Fig. 2). It should be outlined that any new or changed object is *always* appended to the end of the buffer. This is due to the no-overwrite strategy. If you need to change a texture placed in «chunk_036.bin», you'll have to create a copy of this texture and place it into «chunk_037.bin». The XML description will change the reference from 36 chunk to 37, but both chunks will exist in the buffer «until the end of time». This way we can

be sure that most of the objects that user works with are placed in RAM. For common execution scenario any object that was flushed to a hard drive (in the green rectangle in Fig. 2) may only be in 2 states inside renderer. First state: this object has already been passed to the renderer and has valid state inside of it. Second: the object will not be passed to the renderer at all since it is not needed anymore (details further).

3.3. Why no-overwrite

We use no-overwrite strategy due to support for network rendering. Allowing to overwrite any file would create a possibility that on one machine this file will have an old state, and on the other a new one. No-overwrite strategy guarantees this can never happen. If the texture file is not on the local machine, it means that the file has not been transferred to this machine yet (and then the renderer waits for the transfer of this file), or it would not be transferred, because the system has a new state for the same texture. In the latter case, the renderer must go to the next state of the scene and wait for the new file of the same texture to be transferred and ignore the old one.

4. Results and discussion

The API is integrated with a freeware rendering system Hydra Renderer and two DCC applications – 3ds Max and Fabric Engine. The suggested concept of DCC application and renderer integration meets all of the requirements that we listed in the beginning of the paper. Among the drawbacks of the proposed approach, one can note an obvious overrun of disk memory caused by the need to store complete copies of different versions of the same object. Nevertheless, effective network transfer of such copies is possible (for example, by using paged memory for virtual buffer and COW [16] for each page) but wasn't considered by us.

It should be noted that in the proposed approach the rollback to some previous system state in most cases is equivalent to loading this state «from scratch», – a complete analysis of some file «state_N.xml» and loading most of the *needed* (i.e. we don't have to scan the whole virtual buffer and can load only necessary chunks) data for geometry and textures from the hard drive.

5. Acknowledgments

This work is sponsored by RFBR 16-31-60048 «mol_a_dk» and 16-01-00552.

6. References

[1] Ingo Wald Andreas Dietrich and Phlipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. // Eurographics Symposium on Rendering (2004).
[2] Marsel Khadiyev. Ornatrix mentalRay shaders. // third party plugin for mental ray for hairs.
[3] Pixar USD. URL: https://graphics.pixar.com/usd/docs/index.html.
[4] Khodulev A.B., Kopylov E.A., Zdanov D.D. Requirements to the Scene Data Base // Proc. 8th International Conference on Computer Graphics and Visualization, Moscow, 1998, p. 189-195.
[5] Wavefront obj file format. URL: https://en.wikipedia.org/wiki/Wavefront_.obj_file.
[6] Autodesk FBX format. URL: https://www.autodesk.com/products/fbx/overview.
[7] OpenCollada. URL: http://www.opencollada.org.
[8] Siggraph 2011, Alembic talk.
[9] Alembic. URL: http://www.alembic.io/.
[10] MaterialX An XML standard for export and import a shader graph. URL: http://www.materialx.org/.
[11] Дерябин Н.Б., Денисов Е.Ю. Объектно-ориентированная инфраструктура систем компьютерной графики // Graphi'Con 2007, Россия, МГУ июнь 23-27, 2007, с. 289-292.
[12] Барладян Б.Х., Волобой А.Г., Шапиро Л.З. Построение реалистичных изображений в системах автоматизированного проектирования // Графикон 2013, 16-20 сентября 2013 года, с.148-151.
[13] Multiverse. URL: http://multi-verse.io/.
[14] Adam Glick, George Matos. Scalable Enterprise Visualization, GPU Technology Conference 2017.
[15] Arseny Kapoulkine. A light-weight C++ XML processing library. URL: http://pugixml.org.
[16] Bovet, Daniel Pierre; Cesati, Marco. Understanding the Linux Kernel. // O'Reilly Media, Inc. p. 295. 2002.