Data-Driven Method for High Level Rendering Pipeline Construction

Victor Krasnoproshin and Dzmitry Mazouka

Belarusian State University, Minsk, Belarus
krasnoproshin@bsu.by, mazovka@bk.ru

Abstract. The paper describes a software methodology for the graphics pipeline extension. It is argued that common modern visualization techniques do not satisfy current visualization software development requirements adequately enough. The proposed approach is based on specialized formal language called visualization algebra. By invoking data-driven design principles inherited from the existing programmable pipeline technology, the technique has a potential to reduce visualization software development costs and build a way for further computer graphics pipeline automation.

Keywords: graphics pipeline, visualization, visualization algebra.

1 Introduction

Computer graphics (CG) remains one of the most rich and constantly evolving fields of study in computer science. CG consists of two large parts, each studying its own problem: image recognition and image generation. Both of these problems are very broad and complex in nature. In this paper, we concentrate on image generation, i.e. visualization.

Literally every area in human life which involves computer technology requires some sort of visual representation of information. Accurate and adequate visualization becomes vitally necessary with the growing complexity of problems being solved with computers. CG provides tools and theories that target the growing requirements for visualization, but as requirements become more complex and demanding so does the need for improvement in this field.

In this paper we analyze the most widely used visualization methodology and provide a technical solution for its improvement.

1.1 Related Works

Graphics pipeline design has been studied closely by game development companies and graphics hardware manufacturers. In the closest related work – "Advanced Scenegraph Rendering Pipeline" (nVidia) [1], authors recognize deficiencies of the common visualization system architectural approach. They offer unified structured data lists which can be processed uniformly in a data-oriented way. This allows to decrease redundant computations determined by the previous architecture. Developers of Frostbite rendering engine [2] distinguish three major rendering stages in their architecture: culling (gathering of visible entities), building (construction of high level states and commands) and rendering (flushing states and commands to graphics pipeline). These stages represent independent processing events and can run on different threads simultaneously.

In our work we do not concentrate on performance problems of visualization systems. Taking into account the growing recognition of high level data-oriented design principles in graphics development, we are trying to provide a base line for a general solution from the system design's point of view. Technical implementation (Objects Shaders) naturally emerges from the existing shader languages of the graphics pipeline [3].

1.2 Previous Works

In our previous works [4, 5] we studied a theoretical model of a generalized visualization system. We introduced mathematical constructs for objects data space and provided a formalized algebra operating in that space. This work is a practical implementation of those formal concepts.

2 Basic Definitions

Visualization, broadly defined, is a process of data representation in visual form. In computer graphics visualization has a special definition – **rendering**. The rendering process is implemented in computers with a set of dedicated hardware components and specialized software.

There are several rendering algorithms that lay a base for computer visualization. The most widespread are ray tracing and **rasterization** [6]. These algorithms have their own application areas: ray tracing is used for photorealistic rendering, sacrificing computation performance to physically correct output; rasterization, in its turn, is designed for high-performance dynamic applications where photorealism is not as essential.

Almost all contemporary rendering hardware implements the rasterization algorithm. A technical implementation of the algorithm is called a **graphics pipeline**. A graphics pipeline is structured as a sequence of data processing stages, most of which are programmable. These stages are responsible for data transformations specified by the rasterization algorithm's logic. The pipeline is accessible for software developers with special hardware abstraction layer libraries, available on most operating systems. The most popular libraries are DirectX (for Windows) and OpenGL (for any OS). Both of them provide a software abstraction of the underlying hardware implementation of graphics pipeline with all necessary access methods.

The graphics pipeline is a very efficient and flexible technology, but it may be difficult to use in complex applications. This is because the pipeline's instructions operate on a low level with objects like geometrical points and triangles. The best analogy here would be to compare this with the efforts of programming in an assembly language.

The problem of the pipeline's complexity gave rise to a new class of visualization software: **graphics engines**. A graphics engine is a high level interface that wraps around the pipeline's functionality, and introduces a set of tools which are much more convenient to use in real world applications. The engine is normally built around an extendible but nevertheless static computer model, which generalizes a whole spectrum of potential visualization tasks. The most popular visualization model for graphics engines nowadays is the **scene graph**.

Scene graph is not a well defined standard model and many software developers implement it differently for different tasks. However, the implementations often have common traits, which can be summarized in the following definition. Scene graph – is a data structure for visualization algorithms, based on a hierarchical tree where every node is an object and every subnode – a part of that object. That is, a scene graph may have a node representing a chair object and direct subnodes representing its legs. Furthermore, objects may be assigned with so-called materials – fixed properties lists describing how particular objects should be rendered.

A primitive scene graph's rendering algorithm consists of visiting every node of the tree and rendering each of the objects individually using material properties.

3 Analysis

The following figure (Fig.1) summarizes the structure of the visualization process and differences between two approaches mentioned above.

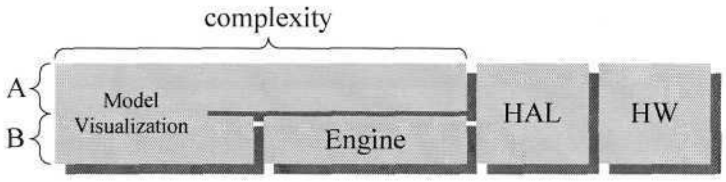

Fig. 1. Visualization process

HW and HAL stand correspondingly for graphics hardware and hardware abstraction layer represented by DirectX or OpenGL. Engine stage is the part of the process taken over by a graphics engine.

The rest of the process is marked as model visualization. This stage is directly connected to the very visualization problem in question. Computer graphics, like any other computing activity, works with **computer models**. These models may represent any kinds of real or imaginary entities, phenomena or processes. Model visualization is essentially a set of algorithms translating the model into another form of representation which is suitable for automatic rendering.

Earlier, we described two common visualization approaches: pure graphics pipeline rendering and graphics engines. They are marked on the scheme as A and B respectively. It is emphasized how graphics engines cut off a serious portion of implementation complexity. However, the real situation is not that simple.

We have mentioned that graphics engines is a whole class of non-standard software. And, moreover, this software is built on the base of certain fixed assumptions, and implements fixed input models that aren't necessarily compatible with the current target model. This situation is depicted on the next figure (Fig.2).

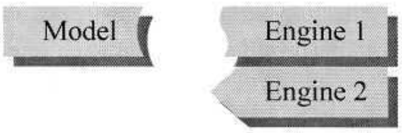

Fig. 2. Model and engines

The model on the picture is not compatible with either of two engines. This happens, for example, in the case when the engines specialize in solid 3D geometric object rendering, and the Model represents a flat user interface.

In this situation software engineers can take one of the three ways:

1. change the model so it fits an engine;
2. provide an engine-model adapter layer;
3. fallback to the pure pipeline rendering.

The first option probably needs the least effort, but the outcome of visualization may seriously differ from the initial expectations as the visualized model gets distorted.

The second option tries to preserve the model's consistency with additional translation stage (Fig.3):

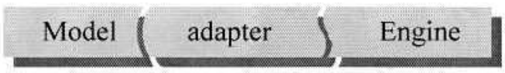

Fig. 3. Model-engine adapter

This may work in some situations depending on how different the model and engine are. If the difference is too big, the adapter itself becomes cumbersome and unmaintainable. In this case the third option becomes preferable: the visualization problem gets solved from the scratch.

The conclusion of this is that we cannot rely on graphics engines from a general perspective. Tasks and models change all the time and engines become obsolete. The variety and quantities grow, which make it troublesome to find the proper match. And so, we have a question: whether anything can be done here in order to improve the visualization process construction.

In our previous works [4, 5] we suggested that it was possible to develop a general methodology for high-level visualization abstractions. That is, to create a model-independent language for visualization process construction.

Together with corresponding support layer libraries, the new visualization process construction would change in the following way (Fig.4):

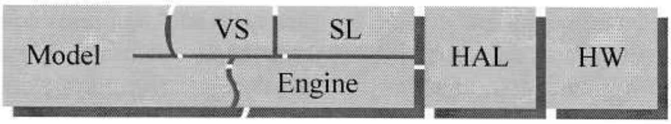

Fig. 4. A new way of the process construction

In this scheme, the model is rendered using an engine with either the first or the second method. SL stands for the standard layer, and VS – for visualization system.

Standard layer is a set of support rendering libraries based on visualization algebra methodology [5]. The library provides tools for process construction in model-agnostic data-driven way. Less effort (in comparison to pure pipeline development) is necessary to make a model adaptation visualization system. And, though the implementation complexity remains slightly bigger than that of the graphics engines, this technique has the advantage of preserving the pipeline's flexibility together with higher level of language abstraction.

4 Object Shaders

Before we start describing the details of the proposed technology, a few words needs to be said regarding how the graphics pipeline is programmed in general.

The graphics pipeline consists of several stages including (DirectX11 model [7]): Input Assembler, Vertex Shader, Hull Shader, Tesselator, Domain Shader, Geometry Shader, Rasterizer, Pixel Shader and Output Merger.

The stages implement different parts of the rasterization algorithm and provide some additional functionality. Shader stages are programmable; all the others are configurable. Configurable stages implement fixed algorithms with some adjustable parameters. Programmable stages, in their turn, can be loaded with custom programs (which are called shaders) implementing specific custom algorithms. This is what essentially gives the pipeline its flexibility.

Shader programs of different types operate with different kinds of objects. Vertex shaders operate with geometry vertices and control geometrical transformation of 3D model and its projection onto 2D screen. Pixel shaders work with fragments – pieces of the resulting image that are later translated into pixels.

The most common way in writing the shader programs is to use one of the high level shading languages: HLSL for DirectX, or GLSL for OpenGL. These languages have common notation and similar instruction sets determined by underlying hardware. By their nature, the languages are vector processing SIMD-based. And corresponding shader programs implement algorithms that describe how every vertex or fragment needs to be treated independently, enabling massive parallel data processing capability.

In our work, we pursue technological integration into the graphics pipeline, rather than its replacement with another set of tools. That is why the technical realization of the proposed methodology is based on emulation of an additional programmable pipeline stage which we call Object Shader.

Object Shader stage is a broad name for three types of programmable components: pipeline, sampling, and rendering. These components are based on corresponding notions from visualization algebra [5]: visual expression, sample, and render operations.

Visual expression in visualization algebra (VA) is a formalized algorithm operating in a generalized object space. Objects in VA are represented with tuples of attributes projected onto a model-specific semantic grid. The methodology does not make any assumptions on the nature of objects and their content, treating all data equally.

Visual expressions in VA are constructed using four basic operations:

1. sample – object subset selection,
2. transform – derivation of new objects on the base of existing ones,
3. render – objects translation into an image,
4. blend – operations with the resulting images.

The final expression for target model visualization must have one input (all the model's data) and one output (the resulting image).

On the technical side, the expression is represented with a program on a language similar to HLSL. The additions are:

1. data type ptr used for declaration of resources
2. object types Scene, Objects and Frame
3. various rendering-specific functions

Object layout declaration in object shaders is done in a common way with structures:

```
struct Object
{
  field-type field-name : field-semantic;
  ...
};
```

Sampler functions are simple routines returning boolean values:

```
bool Sample(Object object)
{
  return sampling-condition;
}
```

Render procedures generate operation sequences for objects of the supported type and return Frame as a result:

```
Frame Render(Object object)
{
  instruction;
  ...
  instruction;
  return draw-instruction;
}
```

Pipeline procedures combine sampling and rendering operations into the final visualization algorithm. The input of a pipeline procedure is a Scene object and a Frame object is output:

```
Frame Pipeline(Scene scene)
{
  Objects list = Sample(scene);
  ...
  Frame frame = Render(list);
  return frame;
}
```

Visualization system routes data streams according to the pipeline procedure logic, splitting it with samplers and processing with renderers. The unit routines are designed to be atomic in the same way how it is done for the other shader types: processing one object at a time, allowing massive parallelization.

This technique is fairly similar to effects in DirectX [8]; but, at the same time, the differences are apparent: effects cannot be used for building the complete visualization pipeline.

In the last part of the paper we will go through the real usage example.

5 Usage Example

The sample model consists of one 3D object (a building) and one 2D object (overlay image). The resulting visualization should visualize the building at the center of screen and make it possible to change its orientation. The overlay image should be drawn at the top right corner.

From the model description we know that there are two types of objects and therefore we need two sampling and two rendering procedures.
Sampling procedures:

```
bool Sample1(int type : iType)
{
  return type == 1;
}

bool Sample2(int type : iType)
{
  return type == 2;
}
```

The procedures get the type field from object stream and compare it against predefined constant values. So the first procedure will sample 3D objects (type 1) and the second 2D objects (type 2).

Rendering procedure for 3D objects starts with object layout declaration:

```
struct Object
{
  float4x4 transform : f4x4Transform;
  ptr VB : pVB;
  ptr VD : pVD;
  ptr tx0 : pTX0;
  int primitive_count : iPrimitiveCount;
  int vertex_size : iVertexSize;
};
```

The supported objects must have transform matrix, vertex buffer (VB), vertex declaration (VD), texture (tx0) and common geometry information: primitives count and vertex size.

Then, we declare external variables, which are required to be provided by the user:

```
extern ptr VS = VertexShader("/Test#VS_World");
extern ptr PS = PixelShader("/Test#PS_Tex");
extern float4x4 f4x4ViewProjection;
```

These variables are: common vertex shader (VS), common pixel shader (PS) and view-projection transformation matrix.

The rendering procedure itself:

```
Frame OS_Basic(Object obj)
{
  SetStreamSource(0, obj.VB, 0, obj.vertex_size);
  SetVertexDeclaration(obj.VD);
  SetVertexShader(VS);
  SetPixelShader(PS);
  SetVertexShaderConstantF(0, &obj.transform, 4);
  SetVertexShaderConstantF(4, &f4x4ViewProjection, 4);
  SetTexture(0, obj.tx0);
  return DrawPrimitive(4, 0, obj.primitive_count);
}
```

The procedure makes a number of state change calls and invokes a drawing routine.

The second rendering procedure is implemented in a similar way:

```
struct Object
{
  float4 rect : f4Rectangle;
  ptr tx0 : pTX0;
};

extern ptr VB = VertexBuffer("/Test#VB_Quad");
extern ptr VD = VertexDeclaration("/Test#VD_P2");
extern ptr VS = VertexShader("/Test#VS_Rect");
extern ptr PS = PixelShader("/Test#PS_Tex");
Frame OS_UI(Object obj)
{
  SetStreamSource(0, VB, 0, 8);
  SetVertexDeclaration(VD);
  SetVertexShader(VS);
  SetPixelShader(PS);
  SetVertexShaderConstantF(0, &obj.rect, 1);
  SetTexture(0, obj.tx0);
  return DrawPrimitive(4, 0, 2);
}
```

The resulting pipeline procedure is very simple: it needs to use the declared samplers and renderers, and combine their output:

```
Frame PP_Main(Scene scene)
{
  return OS_Basic(Sample1(scene))+OS_UI(Sample2(scene));
}
```

Model data in Json format:

```
"Object1" :
{
  "iType" : "int(1)",
  "f4x4Transform" : "float4x4(0.1,0,0,0,  0,0.1,0,0,
    0,0,0.1,0,  0,-1,0,1)",
  "pVB" : "ptr(VertexBuffer(/Model#VB_House))",
  "pVD" : "ptr(VertexDeclaration(/Model#VD_P3N3T2T2))",
  "pTX0" : "ptr(Texture2D(/Model#TX_House))",
  "iPrimitiveCount" : "int(674)",
  "iVertexSize" : "int(40)"
},
"Rect1" :
{
  "iType" : "int(2)",
  "pTX0" : "ptr(Texture2D(/Test#TX_Tex))",
  "f4Rectangle" : "float4(0.75, 0.75, 0.25, 0.25)"
}
```

And the following picture represents the result of visualization (Fig.5):

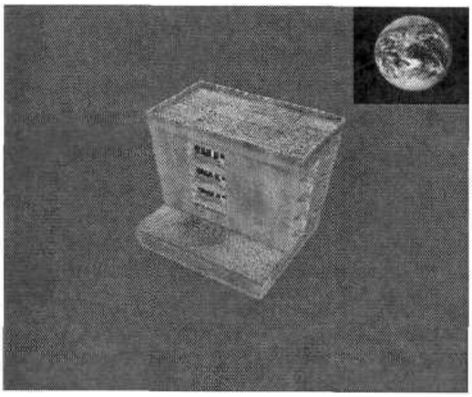

Fig. 5. Result of visualization

6 Conclusion

Computer visualization still holds the status of a heavily evolving scientific and engineering area. Dozens of new techniques and hardware emerge every year. And, with further advancements, this environment may require certain intensive changes in order to stay comprehensible and maintainable.

This paper provides justification and a short overview of the technological implementation of the new visualization methodology based on so-called visualization algebra. This methodology has a potential to improve the most popular existing methods of visualization in terms of flexibility and accessibility.

References

1. Tavenrath, M., Kubisch, C.: Advanced Scenegraph Rendering Pipeline. In: GPU Technology Conference, San Jose (2013)
2. Andersson, J., Tartarchuk, N.: Frostbite Rendering Architecture and Real-time Procedural Shading Texturing Techniques. In: Game Developers Conference, San Francisco (2007)
3. MSDN, Programming Guide for HLSL, `http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635%28v=vs.85%29.aspx`
4. Krasnoproshin, V., Mazouka, D.: Graphics pipeline automation based on visualization algebra. In: 11th International Conference on Pattern Recognition and Information Processing, Minsk (2011)
5. Krasnoproshin, V., Mazouka, D.: Novel Approach to Dynamic Models Visualization. Journal of Computational Optimization in Economics and Finance 4(2-3), 113–124 (2013)
6. Gomes, J., Velho, L., Sousa, M.C.: Computer Graphics: Theory and Practice. A K Peters/CRC Press (2012)
7. MSDN, Graphics Pipeline, `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476882%28v=vs.85%29.aspx`
8. MSDN, Effects (Direct3D 11), `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476136%28v=vs.85%29.aspx`