The Problem of Automation in Dynamic Models Visualization

V. Krasnoproshin [1], **D. Mazouka** [2]
1) Belarussian State University, 220030 Minsk, Nezavisimosti av. 4, krasnoproshin@bsu.by
2) Belarussian State University, 220030 Minsk, Nezavisimosti av. 4, mazovka@bk.ru

Abstract: *This paper describes a methodology for the graphics pipeline extension. The introduced approach is based on specialized formal language called visualization algebra. We argue that this technique can lower visualization software development costs and build a way for further computer graphics automation.*

Keywords: Graphics pipeline, visualization system, visualization algebra.

1. INTRODUCTION

Visual representation is the easiest way for people to deal with complex information. This is why most of the practical problems, which are solved today with computer technology, require visualization for the resulting data. And this was the cause for computer graphics to become a popular area of scientific and engineering interest. During its rapid development, computer graphics has formulated a plenty of sophisticated concepts, and one of the most important is the graphics pipeline. The notion of graphics pipeline stands for a set of methods, devices and software implementing the process of visualization.

The development of the graphics pipeline has been going so far in direction of complete automation. The principal part of its implementation consists of hardware, and software part plays mostly auxiliary interface role. This software is represented now by a number of hardware independent programing libraries [1]. In the same time, however, there are a lot of other problems not covered by the pipeline which stay between automated layer and the original application task.

This paper describes a potential approach for further graphics pipeline functionality extension and automation. This is achieved by complete formalization of the visualization process.

2. PROBLEM ANALYSIS

In computer graphics the term “visualization” (or rendering) refers to the process of translation of some computer model into a raster image (**frame**). So in the very general sense, the **problem of visualization** can be described by the following expression:

$$Models \xrightarrow{A} Images\ , \qquad (1)$$

where *Models* – initial set of models, *Images* – a set of images and *A* – is a translation algorithm.

According to resolve the problem of visualization for some particular model and desired set of images we need to build the algorithm *A*. However, the problem itself is weakly structurized so the algorithm doesn't have a formal representation. And even though modern hardware devices and programming libraries provide a significant assistance in visualization systems development, the number and variety of visualization problems grow every year, making development more costly and less controllable.

To be able to analyze the visualization process, we will make an overview of its processing stages separating them by different levels of abstraction.

The first level of abstraction is the **physical level** (Fig.1):

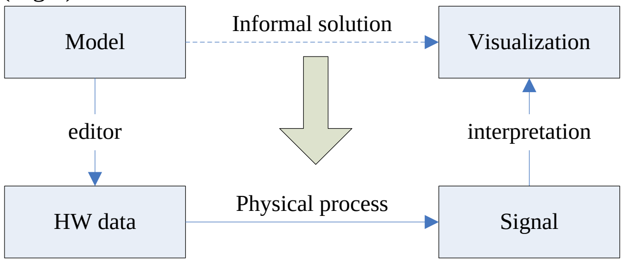

Fig.1 – Physical level of abstraction

At first, the model data is translated into hardware-specific data (HW data). Then it is processed by physical implementation of rasterization algorithm providing Signal as an output. The Signal is interpreted on computer display as resulting visualization of the initial Model.

This simple scheme was used in early computer graphics methodologies, and was very **difficult** (the algorithm was hard to implement using primitive hardware operations), **inflexible** (slight changes in input data would lead to significant changes in implementation) and **hardware-dependent** (the process couldn't be easily moved from one hardware system to another).

According to solve these problems, a new level of abstraction was introduced – the **application level**. On this level all the primitive hardware operations were standardized within special **graphics pipeline** methodology. Two main standard application libraries employing this technology are Direct3D and OpenGL. Taking into account this concept, the visualization process scheme can be represented by Fig.2:

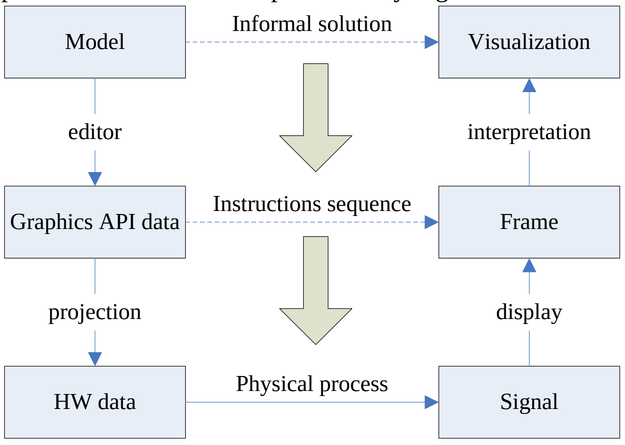

Fig.2 – Application level of abstraction

In this case model data is translated into formal graphics API (library Application Programming Interface) data. The algorithm is implemented as a sequence of instructions, and then automatically transformed into physical process.

Graphics API data is represented by the following data types (**graphical objects**): vertex buffers (geometry data), index buffers (topology data), textures (images which describe surface properties) and shader programs (geometry or image transformation procedures). This data is processed with three sets of operations: *Create* (allocate and initialize graphics data), *Set* (set graphics data input) and *Draw* (render data). Instructions sequence looks like:

$$(Set_1; Set_2 \ldots; Set_n; Draw_1) \ldots (Set_1; Set_2 \ldots; Set_n; Draw_m), \quad (2)$$

here each Draw instruction is preceded with a sequence of corresponding Set instructions.

Thus, using this notion, the problem of visualization can be reformulated as follows: having a computer model defined with a set of graphics objects, build a sequence of instructions which translate this model into a frame.

Graphics pipeline on application level lowers overall difficulty of visualization process implementation and removes the problem of hardware dependency by introducing Hardware Abstraction Layer (HAL) technology. However with the growing complexity of visualization problems, implementation flexibility and difficulty issues appear again.

These problems can be solved using special **graphics engines** [2] – programming libraries which abstract some functions of the graphics pipeline. Graphics engines are successfully used in a large number of practical visualization solutions and are usually much simpler and flexible in comparison to the graphics pipeline interfaces. However, these positive engine's features are local – the visualization systems are usually oriented on specific problem areas. Any attempt to use the engine beyond the frames of its applicability results in serious efforts for adaptation or complete replacement. Thus the problem of effective visualization process construction remains unsolved.

In this article we offer a formal methodology which helps to build general visualization algorithms and makes possible further intensive automation of the graphics pipeline.

3. ABSTRACT VISUALIZATION LEVEL

As it was said above, the problem of visualization process construction is still difficult on the application level. To be able to ease this difficulty we will introduce another process description level – the **abstract level** (see Fig.3).

At this level we are working with a computer model of some dynamic system. We will consider the model in object form [3], as this is currently the most common way for model representation.

If to classify the model's data by type, semantic and domain, we will get a number of categories which we'll call **attribute types** and specified values – **attributes**:

$$T_A = < Type, Domain, Semantic >, A = < T_A, Value >, \quad (3)$$

where T_A – attribute type, A – attribute, *Type* – data type, *Domain* – attribute domain, *Semantic* – the meaning of the attribute and *Value* – attribute's value.

Every model's object O_i can be defined with a set of corresponding attributes $O_i = \{A_j\}$. We will call a **scene** a subset of objects which is to be visualized in the model $S = \{O_i\}$.

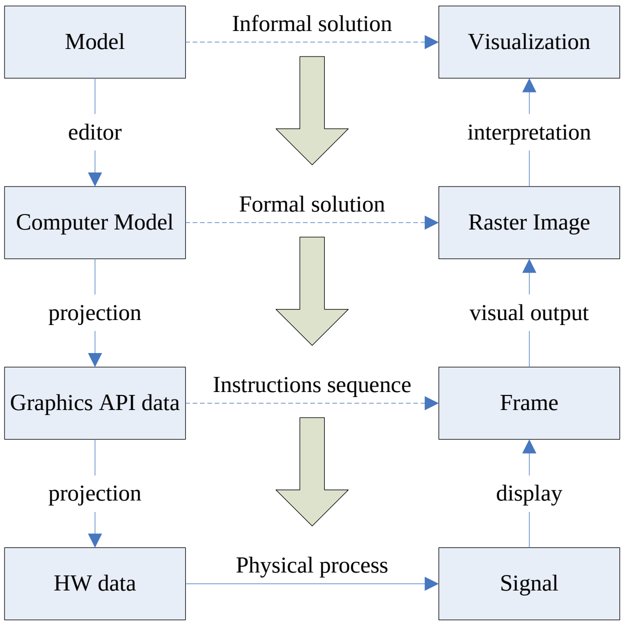

Fig.3 – Abstract level

Using this notation, the visualization problem can be reformulated in the following way: having a defined scene *S* of some computer model, build an algorithm *A* which translates this scene into a frame *F*:

$$S \xrightarrow{A} F \quad (4)$$

Taking into account processes at sibling abstraction levels, the problem of visualization can be represented in this way (Fig.4):

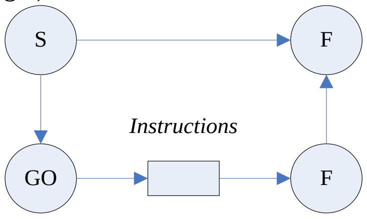

Fig.4 – Abstract and application levels

Here a scene *S* translates into a set of graphical objects *GO*, and then processed by an instructions sequence to get a frame *F*. We can label the whole instructions sequence here as *Render*:

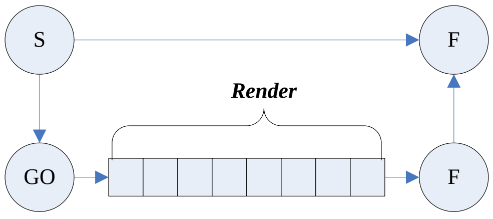

Fig.5 – Render procedure

If the scene is sophisticated enough, we can separate some instructions chunks $Render_i$ which correspond to visualization of specific objects (Fig.6). Each of such chunks can be viewed as a separate visualization process.

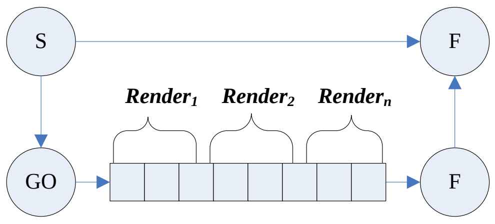

Fig.6 – Render subsequences

Instructions that do not produce any visual output and responsible only for data transformation will be marked as *Transform*. Frames generated by *Render$_i$* are later composed (or blended) together into a single output frame via instructions subsequences which we will mark as *Blend*.

Thus the instructions sequence on the application level can be marked with the following subsequence labels:

1. *Render* – frame generation for some particular set of objects
2. *Transform* – graphics object transformation without frame generation
3. *Blend* – frames composition

This mapping depends on the actual data entering the graphics pipeline. To separate these formalisms from data we will introduce a logical operation *Sample*.

The process of visualization is efficiently a step-by-step objects transformation which can be executed either in single or multiple threads. And using the introduced formalisms the process can be described with the next expression:

$$\begin{aligned}&GraphicsObjects \to Sample^n \to \\ &(Sample \vee Render \vee Transform \vee Blend)^n \to \\ &Blend^n \to Frame\end{aligned} \qquad (5)$$

So the whole graphics objects set is sampled onto subsets which are then processed by *Render*, *Transform* and *Blend* subsequences and finally all resulting frames are blended into a single frame.

In this way *Sample*, *Render*, *Transform* and *Blend* procedures can be used to describe any application level's instruction sequence. And now, according to formalize visualization algorithm itself, we need to move from technological procedures to abstract operations.

4. VISUALIZATION ALGEBRA

Let there is non-empty set of objects *A*, and a set of operations $\Omega_F = \{F_0, F_1, ...\}$ defined on *A*. We will call an **algebra** the object consisting of both these sets: $\Psi = < A, \Omega_F >$ [4].

Now we take a union of a set of all possible finite sets of scene objects and a set of all possible frames:

$$\begin{aligned}&\Theta = \{Objects\} \cup Frames, \\ &Objects = \{O_i\}, 0 \le i \le N, N \in \aleph\end{aligned}, \qquad (6)$$

where $O_i = \{A_j\}, 0 \le j \le M, M \in \aleph$, A_j – is an object's attribute. $Frames = \{F\}, F \in V^{n \times m}, V = \Re^k, n, m, k \in \aleph$ – a set of frames, which is efficiently a set of matrices of real vectors.

Using formalisms introduced above, we can define the following operations:

$$Sample: A, D \to B, A \in \Theta, B \subseteq A, D \in \Theta \qquad (7)$$

Sample initial set *A* into a subset *B* with respect to conditional set *D*.

$$Transform: A_k \to B, A_k \in \Theta^k, k \in \aleph_0, B \in \Theta \qquad (8)$$

Change and rearrange objects in initial tuple of sets A_k and get a resulting set *B*.

$$Render: A_k \to F_p, A_k \in \Theta^k, k \in \aleph_0, F_p \in Frames^p, p \in \aleph \qquad (9)$$

Translate initial tuple of sets A_k into a tuple of frames F_p.

$$Blend: F_p \to F, F \in Frames, F_p \in Frames^p, p \in \aleph \qquad (10)$$

Compose a tuple of frames F_p into resulting frame *F*.

Summarizing this, the following object will be called a **visualization algebra** (*VA*):

$$VA = < \Theta, Sample, Transform, Render, Blend > \qquad (11)$$

Any valid operators superposition in *VA* will be called an **algebraic expression**. For **complete algebraic expression** we will take an expression of the following type:

$$Visual\ Expression: \ S \to F, \qquad (12)$$

where S – is an initial scene and F – a resulting frame.

Now we will show the relationship of a complete algebraic expression in *VA* with some instructions sequence on the application level. Let we have an instruction sequence implementing a visualization algorithm in (4) form: $S \overset{A}{\to} F$. Then we can always make a corresponding algebraic expression in this form:

$$Render_0 \ : S \to F, \qquad (13)$$

where *Render$_0$* – operator encapsulating the whole visualization algorithm. This kind of complete visualization expressions $VisualExpression = Render_0$ we will call **degenerate**. In case if the instructions sequence can be decomposed, we can make more sophisticated algebraic expressions in its correspondence. Thus any instructions sequence implemented on the application level has a corresponding complete algebraic expression in *VA* (degenerate in the worst case).

Now we will show that reverse is true as well. For the sake of this, we will define a **projection** operation:

$$\begin{aligned}&\Pi(args): Transform(args) \to Sequence \\ &\Pi(args): Render(args) \to Sequence \\ &\Pi(args): Blend(args) \to Sequence \\ &\Pi(args): Sample(args) \to (empty) \\ &Sequence = (set_1, draw_1, set_2, draw_2, ..., set_i, draw_i)\end{aligned}, \qquad (14)$$

where *Sequence* – is application level's instructions sequence (2), *args* – corresponding operators' arguments.

Projection provides each of the operators (taken with proper arguments) with an instructions sequence. Now we will look, how we can apply projection to a complete algebraic expression.

A complete algebraic expression (12) can be represented by oriented weakly bound acyclic graph with *S* as initial node and *F* as an ending node. Intermediate nodes of this graph are corresponding operators. Edges correspond to superposition between operations. An example of such graph is visualized on the Fig.7:

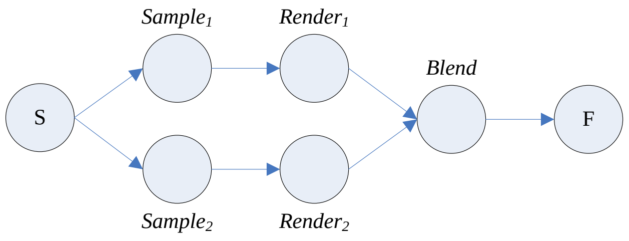

Fig.7 – Example of expression graph

Execution sequence of operators within expression can be obtained via topological sorting of the corresponding graph's nodes. In our case we will get:

$$Sample_1, Render_1, Sample_2, Render_2, Blend \tag{15}$$

Projection of operators sequence equals a concatenation of projections of each operator separately, so we get:

$$\begin{aligned}
&\Pi(S) : VisualExpression(S) \Leftrightarrow \\
&\Pi(S) : Sample_1(S), Render_1(S_1), Sample_2(S), \\
&Render_2(S_2), Blend(F_1, F_2) \Leftrightarrow \\
&\Pi(S) : Sample_1(S), \Pi(S_1) : Render_1(S_1), \Pi(S) : Sample_2(S), \\
&\Pi(S_2) : Render_2(S_2), \Pi(F_1, F_2) : Blend(F_1, F_2) \to \\
&(empty), Sequence_{Render1}, (empty), Sequence_{Render2}, \\
&Sequence_{Blend} = Sequence \Leftrightarrow \\
&\Pi(S) : VisualExpression(S) \to Sequence
\end{aligned} \tag{16}$$

Thus for any complete algebraic expression in *VA* we can get a corresponding instructions sequence via projection of topologically sorted sequence of operations from this expression.

Thus every instructions sequence has a corresponding complete algebraic expression and every complete algebraic expression has a corresponding sequence.

Now we will show, that every visualization problem can be resolved in visualization algebra in the form of complete algebraic expression.

5. VISUALIZATION PROBLEM SOLVABILITY

Having proven the fact that algorithm *A* (4) representations on the application and abstraction levels are mutually related, we can conclude that if the visualization problem is solvable using particular hardware and software on the application level, it can be solved in the terms of visualization algebra. Then visualization problem can be reduced to the following: having a computer model described in abstraction level's (scene) terms and a list of requirements for the final frame, build a complete algebraic expression in *VA*, which implements the process of translation of scene into a resulting frame.

The complexity of reformulated problem is lower than on the application level. And in the same time it is more flexible, because we use maximally generalized form of the initial computer model. Thus the problem of local flexibility, common for any graphics engine, is solved.

Now we will give an overview of the new methodology of visualization problem solution.

6. EXPRESSION BUILDING ALGORITHM

According to build a complete algebraic expression in *VA* (12) $VisualExpression : S \to F$ for the given scene *S* and resulting frame *F*, we need to make at least these basic steps:

1. Define subsets of objects from *S*, which can be visualized with uniform methods:

$$T = \{Objects_i\}, Objects_i \subseteq S, i \in \aleph \tag{17}$$

2. For each element from T define corresponding operator Renderi
3. For each Renderi define corresponding decomposition operator Samplei
4. Define composition operator Blend
5. Build a complete algebraic expression using the following nominal scheme:

Expression → (FirstExpression (SubExpression | λ) LastExpression) | **“F = Render(S);”**
FirstExpression → param **“=”** (**“Render”** | **“Sample”**) **“(S);”**
SubExpression → param **“=”** function **“;”** (SubExpression | λ)
LastExpression → **“F = “** function **“;”**
function → (**“Render”** | **“Sample”** | **“Blend”**) **“(“** (params | λ) **“)”**
params → param | param **“,”** params
param → **identifier**

Expression built using this scheme will be complete in visualization algebra.

7. PRACTICAL EXAMPLE

The following small example demonstrates usage features of the offered methodology.

Let we need to visualize a small scene consisting of a few geometrical objects lit by one light source with some interface handles for object attributes manipulation. An example of a resulting frame is shown on the Fig.8:

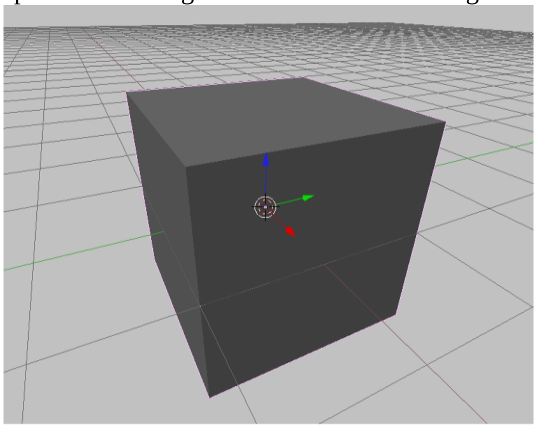

Fig.8 - Example scene

Here we have the following types of objects:

1. Geometrical objects, with attributes:
   a) Vertex buffer – data describing a surface
   b) Transform – transformation matrix
2. Light sources:
   a) Position – source position in the scene
   b) Color – source color
3. Camera
   a) Transform – transformation matrix
   b) Fov – angle defining field of view
4. UI elements:
   a) Vertex buffer – vertices for line ends

b) Transform – transformation matrix

c) Color – element color

Sample operators:

1. *$Sample_{geo}$(Objects, Camera)* – sample geometrical objects from *Objects* which are visible for the *Camera*

2. *$Sample_{light}$(Objects)* – sample light sources

3. *$Sample_{cam}$(Objects)* – sample cameras

4. *$Sample_{ui}$(Objects, Camera)* – sample UI elements visible for the *Camera*

Render operators are necessary only for visible objects:

1. *$Render_{geo}$(Objects, Lights)* – render lit geometrical objects

2. *$Render_{ui}$(Objects)* – render UI elements

Blending operator:

1. *$Blend_{over}$($Frame_1$, $Frame_2$)* – simple overlapping operation, pixels from *$Frame_2$* discard corresponding pixels from *$Frame_1$*

Now we can build the expression:

$Camera = Sample_{cam}(\mathbf{S});$

$Lights = Sample_{light}(\mathbf{S});$

$Geo = Sample_{geo}(\mathbf{S}, Camera);$

$Ui = Sample_{ui}(\mathbf{S}, Camera);$

$Frame_1 = Render_{geo}(Geo, Lights);$

$Frame_2 = Render_{ui}(Ui);$

$\mathbf{F} = Blend_{over}(Frame_1, Frame_2);$

This expression is the implementation of the visualization algorithm for the given visualization problem. And now, if the initial problem changes, the expression can be updated with minimal efforts.

8. CONCLUSION

Visualization problem is still a complicated development task today. The growing number and variety of applications requiring visualization makes traditional methodologies hard to use. According to resolve this issue, the article provides analysis of generalized visualization problem and gives a formal representation of corresponding processes. As a result, a special development methodology was proposed. Based on so-called visualization algebra, this methodology helps in visualization systems development and provides an opportunity for further extension and automation of graphics hardware.

9. REFERENCES

[1] Microsoft DirectX documentation (August 2009).

[2] J. Gregory. *Game engine architecture*. A K Peters, Ltd. Wellesley, 2009.

[3] G. Booch. Object-Oriented Development *IEEE Transactions on Software Engineering. Vol SE-12, NO.2*, February 1986.

[4] Maltsev A. *Algebraic systems*, M. Nauka, 1970, P. 46 – 47.