Efficient Scene Image Synthesis Based on Pipeline Technology

Dzmitry Mazouka
Department of Information Management Systems
Belarusian State University
Minsk, Belarus
mazovka@bk.ru

Victor Krasnoproshin
Department of Information Management Systems
Belarusian State University
Minsk, Belarus
krasnoproshin@bsu.by

***Abstract*—Modern computer graphics is a large and complex area of technology. In this paper we analyse the problem of scene image synthesis. We propose a general approach to using graphics pipeline for rendering that preserves its flexibility while improving the efficiency of development.**

Keywords—computer graphics, graphics pipeline, graphics engine

I. INTRODUCTION

In today's world of rapidly developing technology, the capabilities of graphics hardware are reaching unprecedented heights. We can visualize immersive 3D worlds and simulate complex mathematical models. At the heart of these advances are two fundamental pillars: the graphics pipeline and graphics engines. And while these technical components are the basis for our ability to create images, they also present significant technical challenges for developers and designers.

At first glance, the concept of a graphics pipeline may seem simple – it is a sequence of operations that transform source data into an image. However, the low logical level of abstraction of these operations quickly grows into a complex web of interconnected steps, each of which requires high attention to detail. Building a specialized graphics pipeline for a specific rendering task requires a deep understanding of graphics programming [1].

In addition, the complexity of programming the graphics pipeline increases with the constant development of graphics hardware. Modern GPUs include multiple processors that execute multiple tasks in parallel [2]. And the correct and optimal use of this capability requires advanced skills in building a pipeline. Over time, new technologies push for constant change in established processes and make graphical programming a continuous learning experience [3].

Due to the difficulties of programming the graphics pipeline, many developers use graphics engines to solve their problems. Graphics engines such as Unity [4], Unreal Engine [5] or Godot [6] provide a set of tools and frameworks to simplify the development process. They promise more efficient development cycles, a standard rendering pipeline, and usable models and assets. However, these benefits also come at a price.

Despite the undeniable advantages of graphics engines, their use can impose restrictions on the flexibility and specialization of the project. Not every visualization problem can easily fit within the engine's standard constraints. Users may find it difficult to work with a rigid system that does not support the unique requirements of the problem. This question of balancing flexibility and convenience often confronts developers when considering using engines in their projects.

In this article we propose an approach to solving visualization problems that reduces the impact of the limitations described above. This approach reduces the complexity of using the graphics pipeline and at the same time fully maintains its flexibility. To understand the essence of this method, let's first analyze the composition of the visualization problem.

II. ANALYSIS OF VISUALIZATION PROBLEM

Given some abstract model M, it is required to develop an algorithm that, based on abstract objects of the model, builds a two-dimensional graphic image *Image*. We will call such an algorithm a model visualization algorithm *ModelRender*:

$$ModelRenderer: M \rightarrow Image \quad (1)$$

Let us formalize the described visualization problem.

A set of visualized objects of model M we will call a scene: $Scene = \{Object_i\}$. Model M in general is not limited to the scene and can contain arbitrary processes, parameters, and objects that do not produce visual images. An example of such processes is physics simulation, which affects the change of *Image* over time, but is not directly involved in rendering.

By *SceneRender* we will denote the algorithm for visualizing the scene at time t:

$$SceneRenderer: Scene_t \rightarrow Image_t \quad (2)$$

This rendering model is easy to analyse, but it does not completely cover *ModelRender*, since rendering the entire model may involve iterative changes. That is, the result of previous visualizations can be used as input for subsequent ones. Taking this into account, let's clarify the definition of *SceneRender*:

$$SceneRenderer: Scene_t, Image_{t-1}, \ldots \rightarrow Image_t \quad (3)$$

Let us consider the state of the scene at a certain point in time; in general, the scene visualization algorithm can be decomposed into smaller visualization algorithms $Render_i$:

$$Render_i: Objects, Images \rightarrow Image_{i,t}$$

$$Objects \subseteq Scene_t$$

$$Images = \{Image_1, Image_2, \ldots\}$$

$$SceneRender = Render_1 \cdot \ldots \cdot Render_n : Scene_t, Image_{t-1}, Image_{t-2}, \ldots \rightarrow Image_t \quad (4)$$

The possibility of decomposition depends on the specific model M, with two degenerate cases possible. First: the scene visualization algorithm is radically decomposed in such a way that each object is visualized by a separate small algorithm:

$$SceneRender = Render_1 \cdot \ldots \cdot Render_n$$

$$Render_i: Object_i \rightarrow Image_i \quad (5)$$

And the second case is when the scene visualization algorithm cannot be presented as a composition of smaller algorithms:

$$SceneRender = Render \quad (6)$$

In further analysis, we will assume that decomposition of the scene visualization algorithm is possible, and the number of small algorithms is less than the number of scene objects.

Specific smaller algorithms, and the scene visualization algorithm as a whole, can be viewed as projections from a set of objects and images into a set of images:

$$Render: Objects, Images \rightarrow Images \quad (7)$$

And the goal of the visualization problem is to construct a scene visualization algorithm in the form of a composition of smaller visualization algorithms. The resulting *SceneRender* is a solution to the visualization problem.

The general scheme for solving the visualization problem can be presented as follows:

1. Identify dependencies between the results of scene visualization in a sequence of model iterations.
2. For each iteration, identify subsets of scene objects that can be rendered in a consistent way.
3. For each of these subsets, construct a corresponding visualization algorithm.
4. If the scene rendering has intermediate dependencies between algorithms for rendering subsets of objects, determine such dependencies.
5. Construct the final algorithm for visualizing the scene as a composition of algorithms for visualizing subsets of objects, taking into account dependencies on intermediate results and the results of previous iterations of the model.

The above steps are extremely general. This is primarily due to the fact that the number of practical visualization problems and their variations is extremely large. Any visualization problem can be solved in many different ways, so often some kind of optimization function is applied to many different solutions. For example, if the solution is supposed to be implemented on some hardware platform with limited resources. Or, when optimizing for the speed of execution, it becomes necessary to take into account the operating features of the selected platform.

III. Procedural Graphics Pipeline Primitives

Let's consider the visualization problem in the standard form described above. In order to further analyse the solution, we need to bring the visualization algorithms to the formalisms of the graphics pipeline. That is, given that the implementation of visualization algorithms will be carried out using a graphics pipeline, we need to show how the objects and methods of the pipeline relate to the algorithmic solution of the visualization problem.

In general, graphics pipeline can be thought of as a black box, the input of which is data and instructions, and the output is an image.

To put it simply, we can assume that every time we need to generate an image, we need to provide the necessary data and instructions and start the pipeline. In practice, between different pipeline calls its state is not completely cleared. For example, it would be impractical to load gigabytes of geometry data into graphics card memory for each pipeline run. And also, when using multithreading, it makes no sense to idle the pipeline waiting for each image to be generated. But even taking into account such features, the software interface of the pipeline is modelled according to the principle outlined above and logically we will consider image generation on the pipeline as the following function:

$$Pipeline: Data, Instructions \rightarrow Image \quad (8)$$

Where *Data* is the input data of the pipeline, *Instructions* is the sequence of instructions, and *Image* is the resulting image.

We can map scene objects onto pipeline datasets in the following way:

$$Object_i \rightarrow (Data_1, Data_2, ..., Data_n) \quad (9)$$

That is, each scene object is associated with an ordered set of pipeline data. Let's denote the set of all data of all scene objects *Scene* by *SceneData*:

$$Scene = \{Object_1, Object_2, \ldots\}$$

$$Object_i \rightarrow (Data_{i1}, \ldots, Data_{in})$$

$$SceneData = \{Data_{11}, \ldots, Data_{1n}, ..., Data_{mn}\} \quad (10)$$

Images are also represented by pipeline data, specifically frame buffers, when output is produced to them. If an image is used for rendering, then it is represented by a texture.

Rendering algorithms for a visualization task in standard form in a pipeline are represented as independent sequences of instructions:

$$Render_i \rightarrow (Set_1, \ldots, Set_{1n}, Draw_1, ..., Draw_k) \quad (11)$$

In two degenerate cases we will have either each *Render* containing a single *Draw* call, and one single *Render* containing all *Draw* sequences for the entire scene.

Note that on a pipeline, executing such a sequence of instructions does not automatically generate a frame, or image, that could be used for subsequent operations. So, unlike the *Object* and *Render* above, the *Image* on the pipeline can be represented using:

1. Nothing, if the result of the rendering algorithm on the pipeline is trivially combined with the rest of the results.
2. An execution buffer, if the rendering algorithm is entirely placed in such a pipeline object, and is subsequently trivially combined with other results.

3. Frame buffer and texture, if the result is used by other algorithms or in subsequent iterations.

That is, the representation of an *Image* on the pipeline depends on how the rendering algorithms interact with each other in solving the visualization problem.

As an example, consider the deferred rendering algorithm. The deferred rendering algorithm allows you to efficiently render a large number of light sources simultaneously. To do this, the visualization process is divided into several stages: geometric pass, lighting pass, and combination. During the geometry pass, objects in the scene are rasterized and auxiliary depth, normal, and material maps are generated for each pixel in the image. Each rendered light source then uses these maps to calculate that source's contribution to the illumination of each pixel. The contribution of each light source is accumulated in the light buffer. Ultimately, the auxiliary buffers and the lighting buffer are combined to produce the final pixel colour of the image.

1. For all objects in the scene, calculate the depth, normal and material maps for each screen pixel.
2. For each light source, output the lighting contribution to the lighting buffer in turn.
3. Using the resulting buffers, calculate the final image.

See schematic representation on Fig. 1.

And as an expression:

$$Scene = \{Objects, Light_1, \ldots, Light_n\}$$

$$Render_A(Objects) \rightarrow DepthMap, NormalMap, MaterialMap$$

$$Sum_{i=1,n}(Render_i(Light_i, DepthMap, NormalMap, MaterialMap)) \rightarrow LightBuffer$$

$$RenderFinal(LightBuffer, MaterialMap) \rightarrow Image \quad (12)$$

Similar diagrams and expressions can be produced for all kinds of visualization algorithms, including post-processing, screen space ambient occlusion (SSAO), reflections, shadows, and the like. In each case, you can notice similar patterns, which can be described as follows:

1. Select a subset from the original set of objects.
2. Using this subset, execute a set of instructions on the pipeline.
3. The result of execution on the pipeline can either be used in subsequent operations or output as the final result.

The class of operations corresponding to object subset selection will be denoted as *Sample* procedure. Rendering operations will be denoted as *Render* procedure. In principle, even these two procedures cover the rendering expression above, however, you can notice that in some cases rendering operations work exclusively with images or frames [7]; this type of operation can be separated into a different class – the *Blend* procedure.

This set of procedures logically breaks the visualization problem into smaller fragments using the decomposition principle as on Fig. 2.

With this classification, by using *Sample*, *Render*, and *Blend* procedures, the complexity of the problem, which hides in the relationships between the various subproblems, can be written explicitly. And such connections can be formally standardized in the future, such as, for example, using the concept of object shaders, which we discussed in our previous works [8, 9, 10].

IV. Conclusions

The problem of visualization is a pressing problem nowadays. In this article we explored an approach to solving it that has two contradictory properties: it maintains the flexibility inherent in the graphics pipeline, and also reduces the complexity of development, which makes it similar to using a graphics engine.

We have shown how the analysis of a visualization problem and pipeline tools in general generates a set of procedures by which the solution to the problem can be expressed in a simpler form.

The proposed approach forms the basis for an extension of the programmable graphics pipeline that we considered in our previous works [7, 8, 9, 10].

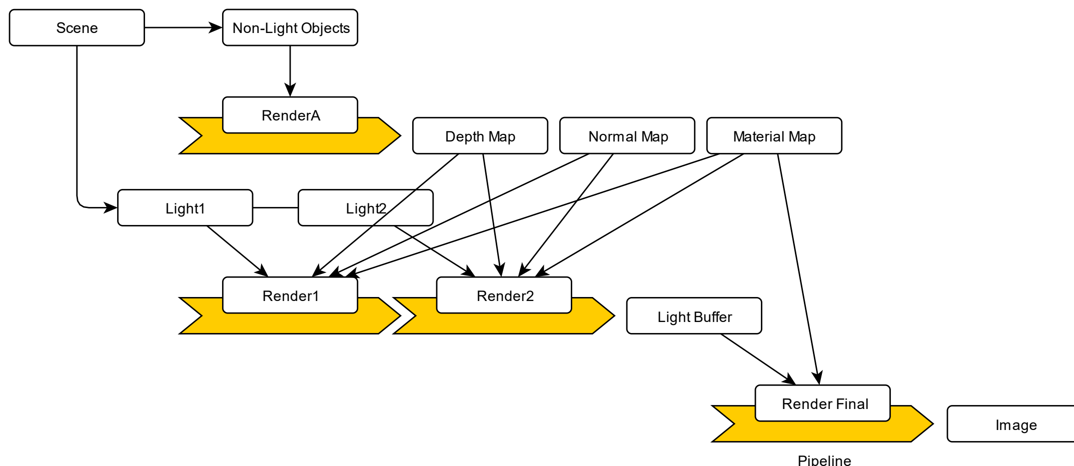

Fig. 1. Schematic representation of the deferred rendering algorithm

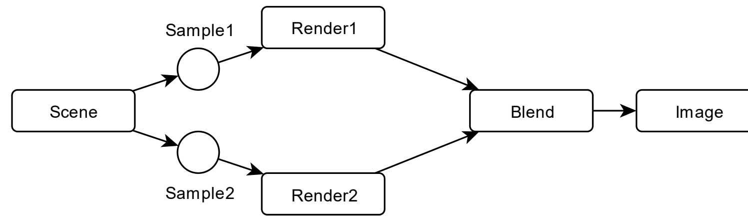

Fig. 2. Problem decomposition using procedures

References

[1] Microsoft, DirectX 12 Programming Guide, https://docs.microsoft.com/en-us/windows/win32/direct3d12/what-is-directx-12-.

[2] David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd ed., Morgan Kaufmann, 2013, pp. 23–39.

[3] Ron Fosner, Real-Time Shader Programming, Morgan Kaufmann, 2003, pp. 88–111.

[4] Unity, Unity Technologies. https://unity.com/.

[5] Unreal Engine, Epic Games. https://www.unrealengine.com/en-US/.

[6] Godot, Godot Foundation. https://godotengine.org/.

[7] V. Krasnoproshin and D. Mazouka, "Frame Manipulation Techniques in Object-Based Rendering" Communications in Computer and Information Science, vol. 673: "Pattern Recognition and Information Processing", Springer, 2017, pp. 97–105.

[8] V. Krasnoproshin and D. Mazouka, "Graphics Pipeline Evolution Based on Object Shaders" Pattern Recognit. Image Anal. 30, 2020, pp. 192–202, https://doi.org/10.1134/S105466182002008X.

[9] V. Krasnoproshin and D. Mazouka, "Data-Driven Method for High Level Rendering Pipeline Construction", Neural Networks and Artificial Intelligence. Communications in Computer and Information Science, vol. 440, pp. 191–200, 2014.

[10] Krasnoproshin, V., Mazouka, D. A New Approach to Building a Graphics Pipeline for Rendering. Pattern Recognit. Image Anal. 32, 282–293 (2022). https://doi.org/10.1134/S1054661822020134.