Computers & Graphics 35 (2011) 1023–1034

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

A survey of raster-based transparency techniques ☆

Marilena Maule[a], João L.D. Comba[a,*], Rafael P. Torchelsen[b], Rui Bastos[c]

[a] UFRGS, Brazil
[b] UFFS, Brazil
[c] NVIDIA Corporation, USA

ARTICLE INFO

Article history:
Received 31 December 2010
Received in revised form
1 July 2011
Accepted 25 July 2011
Available online 3 August 2011

Keywords:
Transparency
Fragment sorting
Order-independent transparency

ABSTRACT

Transparency is an important effect for several graphics applications. Correct transparency rendering requires fragment-sorting, which can be more expensive than sorting geometry primitives, and can handle situations that might not be solved in geometry space, such as object interpenetrations. In this paper we survey different transparency techniques and analyze them in terms of processing time, memory consumption, and accuracy. Ideally, the perfect method computes correct transparency in real-time with low memory usage. However, achieving these goals simultaneously is still a challenging task. We describe features and trade-offs adopted by each technique, pointing out pros and cons that can be used to help with the decision of which method to use in a given situation.

1. Introduction

Transparency is the physical property of materials that allows light to pass through objects. This property is important to estimate the appearance of real objects, and it is largely used to denote the relationship among structures in visual interaction. The focus of this survey is to summarize and describe specific techniques for rendering transparent objects using raster-based pipelines.

In raster-based pipelines, the computation of transparency is simplified based on the assumption that there is no refraction when light passes from one medium to another. In this formulation, computing the correct transparency requires properly blending fragment colors, considering their surface opacities and their distances to the viewpoint, as described by Porter and Duff [1]. For transparency to be correctly computed, fragments must be composed in the same order of their distance to the camera, either in front-to-back (FTB) or back-to-front (BTF)[1] ordering.

Fragments combined in unsorted depth order might not have their contributions properly evaluated. For example, consider the computation of a pixel color involving three fragments. Assume that in correct depth order the closest and farthest fragments are transparent, while the middle fragment is opaque. If the opaque fragment is combined last (after the transparent ones), the incorrect contribution of the farthest fragment cannot be discarded anymore.

Fig. 1(a) illustrates the expected result after rasterization of three triangles facing the camera. In this example, the green triangle is the closest to the camera, while the red one is the farthest. The BTF blending equation expects triangles in the following order: red, blue, and green (as in Painter's algorithm [2]). This way, the green contribution is correctly identified as the nearest to the camera. Fig. 1(b) illustrates color blending in incorrect depth order. Note that the green triangle is not drawn in front of the others, due to the out-of-order blending. Tables 1 and 2 give numerical examples for both situations.

As we can conclude that sorting is the main topic of the papers we survey, and we use it as the criteria to classify methods into the following categories:

- *Geometry-sorting*: sort geometry (meshes or primitives) before rasterization;
- *Fragment-sorting*: sort rasterization fragments before blending, using buffer-based or depth peeling;
- *Hybrid-sorting*: combine geometry-sorting with fragment-sorting;
- *Depth-sorting-independent*: blend fragments without considering their depth order;
- *Probabilistic*: estimate visibility without sorting.

We organize our presentation using the above classification in the sections that follow. To clarify the discussion, parameters used by the methods are given in Table 3.

2. Geometry-sorting methods

The geometry-sorting methods render transparent objects (usually composed of triangle meshes) by either sorting the objects

☆ This article was recommended for publication by E. Reinhard.

* Corresponding author. Tel.: +55 51 3308 6930; fax: +55 51 3308 7308.
E-mail addresses: joao.comba@gmail.com (J.L.D. Comba), rafael.torchelsen@gmail.com (R.P. Torchelsen), rbastos@nvidia.com (R. Bastos).

[1] Back-to-front blending equation [1]: $C'_{dst} = (1-\alpha)C_{dst} + \alpha C_{frag}$.

0097-8493/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2011.07.006

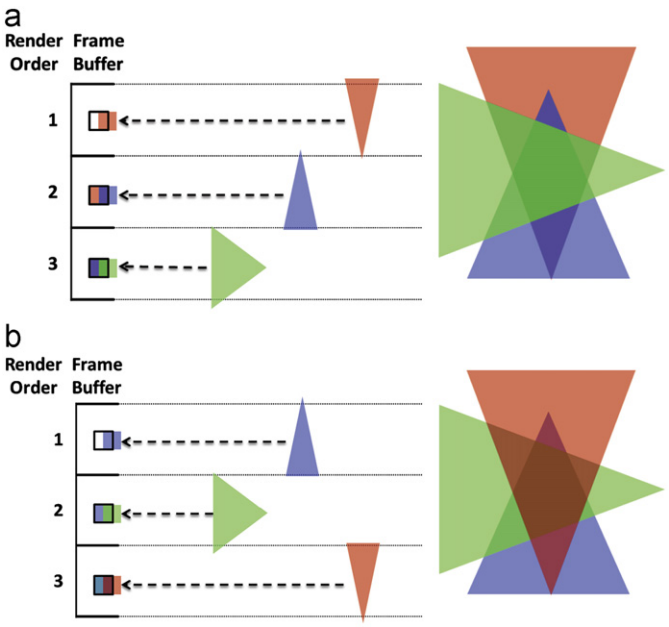

Fig. 1. Blending evaluation for green, blue, and red triangles (in viewing order). On the left, we illustrate the blending of fragments generated during the rasterization of each triangle. The arrow length represents the fragment depth. On the right, we show the resulting image after blending. Primitives processed in-depth order (a) produce different results compared to objects processed in out-of-depth order (b), whereas blending in-depth order produces the expected result. (a) Depth ordering: correct blending and (b) Out-of-order: incorrect blending. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1
Numerical example of a pixel evaluated with the BTF order. Each line represents a fragment blending order (top to bottom).

Fragment	RGBα	Depth	$BTF_{blended\ result}$
Background	(1, 1, 1, 1)	∞	(1, 1, 1, 1)
1	(1, 0, 0, 0.4)	3	(1, 0.6, 0.6, 0.76)
2	(0, 0, 1, 0.4)	2	(0.6, 0.36, 0.76, 0.616)
3	(0, 1, 0, 0.4)	1	(0.36, 0.616, 0.456, 0.53)

Table 2
Numerical example of a pixel evaluated out-of-depth order with the BTF blending equation [1]. Each line represents a fragment and the blending order (top to bottom).

Fragment	RGBα	Depth	$BTF_{blended\ result}$
Background	(1, 1, 1, 1)	∞	(1, 1, 1, 1)
1	(0, 0, 1, 0.4)	2	(0.6, 0.6, 1, 0.76)
2	(0, 1, 0, 0.4)	1	(0.36, 0.76, 0.6, 0.616)
3	(1, 0, 0, 0.4)	3	(0.616, 0.456, 0.36, 0.53)

Table 3
Parameters used by different methods.

W	Screen width
H	Screen height
S	Number of samples for super sampling
m	Number of objects
n	Number of geometry primitives ($n \gg m$)
p	Pixel size: 12B (8B: RGB 8b per channel, 8b alpha, 4B depth value)
l	Number of transparent layers
d	Average number of transparent layers ($d < l$)
k	Buffer entries per pixel (in fragments)
f	Min(k, d)
s	Samples per pixel

or at a finer granularity, by sorting their primitives (i.e. triangles). Both approaches are simple to implement and can directly leverage the alpha-blending support on GPUs (graphics processing units). These methods sort objects or primitives before rasterization and are arguably the most widely used transparency technique in games.

Two situations can lead to image artifacts when using geometry-sorting methods: interpenetrating geometry and out-of-order arrivals. Interpenetrating geometry makes it difficult to properly sort objects and primitives, which leads to blending artifacts. Out-of-order arrival can arise when the technique sorts entire meshes instead of triangles, which can also lead to blending artifacts. Both situations can be properly handled by sorting at the fragment level.

2.1. Object sorting

This approach sorts objects as single entities, usually by the centroid of their meshes. It is prone to artifacts due to the out-of-order arrival of fragments, since object ordering is approximate and does not guarantee correct ordering at the primitive level.

Object sorting methods first sort all the m objects in $O(m \log m)$, followed by rasterization of object primitives using $O(n+WH)$ operations, and blending using $O(WHd)$ operations. The scene is rendered in one geometry pass, with no need for extra memory, apart from the color and depth buffers, which require $O(WHp)$ bytes.

2.2. Primitive sorting

Sorting geometry at the primitive level is the finest granularity that can be obtained in object space. It allows solving the out-of-order problem when no interpenetration occurs. One way to solve interpenetration cases is to split primitives, which might have a high computational cost and still generate z-fighting problems. Another option is to solve interpenetration analytically at even higher costs.

Primitive sorting requires sorting n primitives (triangles) in $O(n \log n)$ operations. The remaining steps and analysis are analogous to the object-sorting given above.

3. Fragment-sorting methods

Fragment-sorting methods compute transparency by z-sorting at fragment level before blending. Fig. 2 shows an example of out-of-order rendering of triangles, where fragments are sorted before blending. We further subdivide the fragment-sorting methods into two categories:

- *Buffer-based* methods store and sort the fragments before blending;
- *Depth peeling* methods extract depth order implicitly through a multi-pass rendering approach.

The main advantage of fragment-sorting methods is the quality of the image, often superior to all other methods. On the other hand, the computational cost and/or memory footprint, due to sorting, is considerably higher.

3.1. Buffer-based methods

Buffer-based methods use a buffer to store fragments while they are generated. After rasterization, a sorting step computes the correct blending ordering. The advantage of these methods is image quality, when compared to sorting-based methods like geometry-sorting, and performance, when compared to *depth peeling* methods,