A MODEL TO BLEND RENDERINGS

Vincent Boyer and Dominique Sobczyk
L.I.A.S.D.-Université Paris 8
February 8, 2013

Abstract.
We propose a model to blend renderings. It consists in mixing different kind of rendering techniques in the same frame to enhance the visualisation of informations on large scenes. This method is achieved in real-time during the rendering process using the GPU programming. Moreover rendering techniques used and key points defined by the user can be interactively changed. In this paper we present the model, a new non-photorealistic rendering technique and images produced by our method.

Key words:
Non-Photorealistic Rendering GPU

1. Introduction

This paper presents a method to blend different renderings. The aims of this model are both a better looking and a more informative image. Rendering is the process of generating an image for a given model. This can be achieved using software programs (generally through a graphic programing library) or/and GPU programs. Generally renderings are classified in two main categories: photo-realistic and by opposition non-photorealistic. Photo-realistic renderings do not allow to enrich the image produced. By opposition non-photorealistic renderings try to enhance the transmission of information in a picture [12], [2]. Based on this idea that a new rendering will help the user to visualize 3D models, a lot of previous works have been done. We present some of these ones organized in different topics:

• deformation of the 3D mesh according to the viewpoint. Rademacher [10] has proposed to create a view dependent model. It can be used especially by cartoonists. A generalization of view-dependent deformations was presented by Martin et al. [7]. Other works have been presented and for example Wood et al. [14] proposed the generation of panoramas for a given 3D scene and a camera path.

• edges extraction. A lot of well-known works have been done to extract edge features like silhouettes, boundaries and creases from 3D models [6], [1], [4], [11]. Generally, a preprocessing operation should be done to convert a 3D model in another data structure in order to obtain edges more easily (for example half-edge data structure maintains edge adjacency). The main application of these techniques are the architectural and technical illustrations.
• artistic shading. Hatching [13], [3], cartoon shading [5] and celshading are examples of non-photorealistic shading. They replace Gouraud or Phong shading to convey three-dimensional structure of objects in an image. Note that celshading includes often also outline shading.

As one can see each of these techniques have its favorite applications, advantages and drawbacks. The motivation of this paper can be summarized by two questions: why do we use one and only one rendering to produce an image? Is the visualization better if the user choose different rendering techniques and apply them simultaneously?

Previous works have been done to mix different renderings techniques[9], [8]. For example, Myszkowski et al.[8] used simultaneously a combination of a hybrid ray tracing and image-based rendering technique and a novel perception-based antialiasing technique. This accelerates rendering of high quality walkthrough animation sequences along predefined paths. But for these methods, the rendering techniques mixed are pre-defined and can not be changed dynamically. In fact these methods are not used to improve the visualization but are used to choose rendering techniques according to their qualities and human eye sensitivity to accelerate the rendering process.

This paper presents a model to combine different renderings in the same frame. This model is designed to enhance the visualization of large scenes. For this reason each rendering and its influence on the scene will be chosen by the user and can be changed dynamically. In the following, we present the general algorithm of our model and detail the solutions to solve each problem. Then images produced by our system are detailed and in conclusion the limitations and future works are presented.

2. The model

Our model mixes different renderings at the same time. The user should specify which and where renderings will be used. The main process of our model is realized on fragments. For a given fragment, different renderings can be applied simultaneously and should be blended. Due to the real-time constraint, the computation should be done on GPU. The specifications given by the user are:

1. **key points** which are points of the 3D space. In the following \( n \) is the number of key points given by the user;

2. renderings used and for each of them four distances:

   • \( d_{r_0} \): the minimal distance in order to use this rendering;
   • \( d_{r_1} \): the minimal distance where we maximize the use of this rendering. Between \( d_{r_0} \) and \( d_{r_1} \) the rendering will be shaded;
   • \( d_{r_2} \): the maximal distance where we maximize the use of this rendering;
\[ \bullet d_{r3}: \text{the maximal distance in order to use this rendering. Between } d_{r2} \text{ and } d_{r3} \text{ the rendering will be shaded.} \]

Left part of figure 1 presents four key points given by the user and their visualization on the scene and the right part shows the distances for a texture rendering given by the user on the interface and the result on the scene. The distances \( d_{r0} \) and \( d_{r1} \) are equal to 0.0, \( d_{r2} \) is equal to 0.5 and \( d_{r3} \) is equal to 0.8. Thus texture is applied in the range of 0.0 to 0.5 from the key points and in the range between 0.5 to 0.8 the texture is applied shaded with black. In this example we give four key points, one rendering and its associated distances. Note that the number of key points, the coordinates of key points, the number of rendering used, the renderings chosen and the distances used could be modified dynamically by the user. The general form of our algorithm is:

For a given fragment \( F \)

1. Compute the closest distance \( d \) to the key points
2. For each rendering
   - Compare the distances given by the user and distance \( d \) and compute a weighted value \( v \)
   - Compute the color and weight it by \( v \)
3. Sum the colors previously obtained and clamp it between 0 and 1 for each component.

These computations are realized only in the GPU and we obtain real-time renderings. Remark that at the present time, fragment shaders do no permit to make loops on data which do not reference a texture. So even if we present the algorithms including loops for the reader, we unfold it. This limit should disappear with future versions of graphic cards and shaders.
**Distance to key points**

We propose three modes to consider key points:

- **single points**: the distance \( d \) is the minimal euclidean distance between fragment and each key point(s);
- **points of a polyline**: the key points form a polyline and \( d \) is the minimal distance between fragment and this polyline. The algorithm used to compute \( d \) for a fragment \( F \) is:

  For each segment \( P_i P_{i+1} \) \((i \in [0; n-2])\) of the polyline
  
  If \( P_i \vec{P}_{i+1} \odot \vec{P}_i F < 0 \) then \( d_i \) is the euclidean distance between \( P_i \) and \( F \)
  
  Else if \( P_i \vec{P}_{i+1} \odot \vec{P}_i F > \|P_i \vec{P}_{i+1}\|^2 \) then \( d_i \) is the euclidean distance between \( P_{i+1} \) and \( F \)
  
  Else \( d_i = \|P_i \vec{P}_{i+1} \odot \vec{P}_i F\| \|P_i P_{i+1}\| \)

  \( d = \min d_i \)

  where \( \odot \) represents the dot product of two vectors, \( \otimes \) the cross product of two vectors and \( \|P_i P_{i+1}\| \) the norm of the vector \( P_i \vec{P}_{i+1} \).

- **points of a polyline loop**: in this case, we consider the \( n \) key points as \( n \) points of a polyline but \( P_{n-1} P_0 \) is considered as a polyline segment. The distance \( d \) is computed using the algorithm described for the polyline mode including one more segment (i.e. \( P_{n-1} P_0 \)).

Figure 2 shows the influence of the mode used in the computation of the distance \( d \). The polyline mode is presented at the left and the polyline loop mode is shown on the right while the single points mode is presented on the left of figure 1.

**Compute the weighted value for each rendering**

For each rendering used, we compute a coefficient \( v \) depending on its four associated distances \((d r_0, d r_1, d r_2, d r_3)\). \( v \) is then applied to weight the rendering. The sum of weighted rendering on a fragment allows us to mix renderings on it. The algorithm is:

For each rendering

If \((d \in [d r_0; d r_3])\) then

If \((d < d r_1)\) \( v = \frac{d - d r_0}{d r_1 - d r_0} \)

Else if \((d \leq d r_2)\) \( v = 1.0 \)

Else \( v = \frac{d r_3 - d}{d r_3 - d r_2} \)

Else \( v = 0 \)

Remark that for a given fragment \( f \), it is possible to have zero, one or more renderings. Each rendering is weighted independently and \( v \) can be viewed as the alpha channel for a rendering on a fragment. Finally the fragment color is clamped between 0 and 1.
1. We include the ability to blend the image with the background. Figure 3 illustrates this part: the top right screenshot shows the use of blending; on the bottom image, the distance $d_{r3}$ of texture rendering has been changed and fragments are rendered with texture, $wbdist$ and material when $d = 0.55$.

**Rendering**

Texture rendering, material rendering, toon shading and celshading (using texture and outline) have been implemented in GPU fragment shader. Vertex shader is used to transform vertex coordinates and to compute necessary data in the fragment shader. We implement a new rendering, named $wbdist$ for “white to black distance”. This rendering is very simple and is based on the coefficient previously computed. The fragment color components received the coefficient $v$. Then, if the fragment is in $[d_{r1}; d_{r2}]$, the color is white and alpha is 1, else the fragment color is an achromatic color and alpha is in $[0; 1]$ depending on the distance $d$. For example, it allows us to make a halo effect when $d_{r1}$ is closest to $d_{r2}$. This can be used to mark a key point or a path as shown in figure 4. Also, other effects like NASA aerogel can be easily produced.

**3. Images and Results**

We present images produced with our model. Those have been produced on a PC pentium IV 3.6Ghz with 1Go of memory and a nvidia graphic card Quadro FX 1400. The top of figure 5 presents a first part of the path described with key-points. As one can see, the halo rendering and the mix of renderings help the user to obtain the essential information needed. The bottom of the figure presents two other views where the used renderings have been changed. Indeed, in the first of these, textured rendering then
Fig. 3. Compute the coefficients and render the scene

Fig. 4. Halo and aerogel rendering
material rendering are used according to the distance of the polyline and in the second one this is the opposite (i.e. material then textured rendering). The minimal frame rate obtained for this scene composed by 20 000 triangles is 27 frames per second with 4 renderings (textured, material, wdist and celshading) used simultaneously and 4 key-points. In fact, the number of key-points used has no influence on the frame rate (i.e. time computation needed for the distance computation is negligible).

4. Conclusion

We have proposed a model to blend renderings based on GPU programing. A collection of rendering shaders have been implemented and a new rendering has been included but other ones can be easily added. The user can apply particular renderings on each part of the scene. This permits to have only advantages of renderings and never their drawbacks. Moreover every parameters can be modified by the user dynamically and it is very intuitive. We can apply immediately, as shown in examples, this model to a car journey. Future works will be done to propose other interpolations in order to compute
the coefficients (non-linear interpolations) and to automatically import other renderings.

References