OpenGL 4.6 Deferred Renderer with Deferred Depth Peeling

~

Luigi Rapetta

Index

1. Introduction

The presented project is an OpenGL 4.6 deferred renderer with order independent transparency (OIT) based on depth peeling. The latter technique is fully integrated into the whole deferred rendering process and it exploits its advantages.

The following paragraphs will briefly introduce the techniques used, and will discuss their implementation and outcomes.

2. Order Independent Transparency

In literature, the problem of correctly rendering transparent surfaces has been dealt with several techniques. A fundamental factor for determining their ontology is the importance of sorting such surfaces, given their distance from the virtual camera.

Hence, there are two main categories for transparency techniques:

- *order dependent*: surfaces must be sorted considering their distance from the camera before being rendered;
- *order independent*: sorting of surfaces can be ignored.

Order dependent transparency can be challenging to implement, and it’s especially taxing for the CPU due to surfaces’ sorting. Order independent transparency (OIT), on the other hand, doesn’t require to draw transparent surfaces in order. This saves to the application all the computation involved with sorting.

1. Techniques overview

OIT techniques are generally categorized into:

- *exact*: accurate transparency, even in complex scenes, but more expensive;
- *approximate*: less accuracy in complex scenes, but less expensive.

Some exact OIT techniques include:

- *A-buffer*: lists of fragments is stored for each pixel, then sorted by the fragment’s distance from the camera – it may lead to GPU local memory limited occupancy during kernel execution;
- *depth peeling* [1]: depth buffer is used in multiple passes to peel a layer of fragments at each pass – explained in depth in section 2.2;
- *dual depth peeling* [2]: performance improvement from single depth peeling.

Among approximate OIT techniques, there’s *weighted, blended OIT*. Such technique uses a weighting function and three rendering passes in order to obtain an approximated, decent image with relatively low costs. It comes with the drawbacks of not supporting refraction and it cannot be implemented into a deferred renderer.

2. Depth peeling

Depth peeling is the OIT technique chosen for the project. It allows to render images oriented to quality, therefore displaying an exact transparency of surfaces.

Depth testing renders the fragment nearest to the camera. There's no way that standard depth testing can return the second nearest, the third nearest, and so on [1]. Depth peeling comes to aid for this problem.

Depth peeling is a multi pass process, in which the color and depth of the scene are rendered multiple times. In the first pass, the scene is rendered normally, and the depth test returns the nearest surfaces. By rendering the next pass, fragments with a depth less than or equal to the depth from the depth buffer of the previous pass are discarded (*peeled away*). The remaining fragments are then processed, and their depth is written on the depth buffer for this pass, which will be used in the next pass for the same process described before. With *n* passes over a scene, *n* layers deeper into the scene are extracted [1].

The *n* layers extracted from the scene can be blended together in order to get an image with correct transparency. The number of layers rendered does influence the quality of the final output.

The blending of such layers can occur in two ways:

- *back-to-front*: the layers are blended from the one with the farthest depths to the one with the closest depths;
- *front-to-back*: the layers are blended from the one with the closest depths to the one with the farthest depths.

Back-to-front blending uses a simple blending equation at the expense of storing in memory all the color buffers of every layer. On the other hand, front-to-back uses more complex blending settings, but saves the application from keeping all the color buffers by only requiring to store the depth buffers of current and previous layers.

3. Front-to-back blending

Given an output color buffer, layers extracted through depth peeling can be blended directly onto the output buffer, therefore in a front-to-back fashion. This can be done with a special blending configuration and a highly reduced number of depth and color buffers to keep during the whole process.

The buffers involved are solely the output color buffer and the depth buffers from the currently rendered layer and the previous one. If also opaque objects are to be rendered, then the depth buffer for opaque objects must be kept and used to occlude transparent objects.

For each pass, the layer is extracted by using the depth buffer of the previous pass. Eventually, the depth buffer of opaque objects is used to discard occluded fragments. The resulting fragments are then blended over the output color buffer and their depth is written onto the depth buffer of the current layer.

In the next pass, the two depth buffers switch each other: the one which was the depth buffer of the current layer, now it's the previous layer's, and vice versa.

The blending equation involved is the following [2]:

$$C_{dst} = A_{dst}\,(A_{src}\,C_{src}) + C_{dst}$$
$$A_{dst} = (1 - A_{src})\,A_{dst}$$

It is very common the notion that, regardless of the technique used, transparent objects must be rendered just after opaque ones. Yet, depth peeling with front-to-back blending requires the opposite: opaque objects must be rendered after all transparent layers are blended. The blending equation used for opaque objects is the same one shown above.

As mentioned above, rendering of transparent objects still requires the depth buffer of opaque objects. Hence, a depth pre-pass of opaque objects is required in order to acquire the respective depth buffer. This drawback can be highly mitigated with the application of deferred rendering, as explained in section 3.4.

3. Implementation

The proposed renderer implements depth peeling with front-to-back blending using deferred shading. To achieve this, some peculiar routes had to be taken. The solutions are discussed in this chapter.

1. Materials

The application displays a series of *entities*, whose appearance is defined by *material* structs. These materials are physically based, and support the metallic workflow. The material parameters are the following:

- diffuse;
- roughness;
- metalness;
- ambient occlusion.

It is required to separate transparent entities from opaque ones in two distinct lists to obtain a correctly rendered scene. There's an underlying system which facilitates so. Moreover, the application also implements a system which supports the dynamic change of material settings. The transparency of an entity can be modified at runtime, and the reference of such entity is correctly assigned to one of the two lists mentioned before.

2. Framebuffers

The application needs several OpenGL *framebuffer objects* (*FBOs*), listed below:

- two *transparent G-buffer FBOs*;
- one *opaque G-buffer FBO*;
- one *plain FBO*.

The two transparent G-buffer FBOs relates to the rendering of transparent entities, while the sole opaque G-buffer FBO relates to the rendering of opaque entities. Each G-buffer FBO (transparent and opaque) owns a proper depth buffer. The two transparent FBOs share a single set of color buffers used to store material information, positions and normals of entities. The opaque G-buffer FBO has its own set of color buffers, separated from the one of the two transparent FBOs.

The plain FBO is the one responsible of storing all the final rendering results of the lighting passes of both transparent and opaque entities. It only has a color buffer, therefore depth testing is not executed when this FBO is active. The blending equation showed in section 2.3 is used on the color buffer of the plain FBO.

3. Deferred depth peeling

The deferred pipeline has been slightly modified in order to take depth peeling into account. The pipeline is knowingly a multi pass process, defined by a geometry pass and a lighting pass. Depth peeling is a multi pass technique as well. The way the latter is implemented is that one depth peeling pass consists into a geometry and lighting pass, therefore one deferred pipeline execution is run at each depth peeling pass.

The geometry pass is responsible to extract from scene entities all relevant information useful for rendering, such as material information, positions and normals of entities (briefly: *geometry data*). In this application, the geometry pass is also responsible of performing the peeling of a single layer. The lighting pass is then responsible to blend the just extracted layer onto the color buffer of the plain FBO. This is done by using the blending equation showed in section 2.3.

The two transparent G-buffer FBOs do alternate at each layer. The active transparent G-buffer FBO is used for rendering geometry data (by writing on the shared G-buffer and the depth buffer of the active FBO), while the depth buffer of the other transparent G-buffer FBO is used as the depth buffer of the previous layer.

By alternating rendering on the two transparent G-buffer FBOs, peeling is perfectly doable. The only G-buffer shared between the two FBOs is sufficient to store all the geometry data needed to render the current layer. Data written on this sole G-buffer is then used to compute fragments in the lighting phase which are directly blended over the color buffer of the plain FBO in a front-to-back fashion.

The pseudo-code of this process is the following:

```
for i from 0 to (depth_peeling_passes – 1):
        currID = i % 2
        prevID = 1 – currID
        disable blending
        activate and clear transparentGBufferFBO[currID]
        pass depth buffer of transparentGBufferFBO[prevID] to geometry shader
        for each entity in transparentEntities:
                geometryPass(entity, ...)
        enable blending and set front-to-back equation
        activate plainFBO
        lightingPass(...)
```

4. Introducing opaque surfaces

As mentioned in section 2.3, front-to-back blending requires opaque surfaces to be blended last, after all transparent entities are rendered. Yet, rendering of transparent surfaces still requires the depth data of the opaque ones for correct occlusions. Traditionally, this leads to a depth pre-pass, in which opaque surfaces are sent down to the pipeline in order to obtain their depth, just to be sent back to it again after all the transparent surfaces are rendered.

The intrinsic nature of the deferred pipeline can be exploited to solve this inefficiency. Briefly, the workflow is:

- execute a geometry pass on opaque surfaces;
- execute depth peeling on transparent objects (geometry and lighting passes for each depth peeling pass), using the depth data of opaque surfaces;
- execute a lighting pass on opaque surfaces.

The geometry pass of opaque surfaces renders on the G-buffer of the opaque G-buffer FBO. It's important to remind that such framebuffer has a G-buffer and a depth buffer completely independent from the G-buffer shared between the two transparent G-buffer FBOs and their respective depth buffers.

Basically, the prevention of a wasteful depth pre-pass happens in this initial geometry pass. Depth of opaque surfaces is computed and later used in transparency rendering, while G-buffer data of opaque surfaces is kept and used in the final lighting pass for opaque surfaces. This way, opaque surfaces are sent to the pipeline just once.

Keeping this optimization in mind, the pseudo-code in section 3.3 can be updated:

```
// [1] Geometry pass for opaque entities
disable blending
activate and clear opaqueGBufferFBO
for each entity in opaqueEntities:
         geometryPass(entity, ...)

// [2] Geometry and lighting passes for transparent entities
for i from 0 to (depth_peeling_passes – 1):
         currID = i % 2
         prevID = 1 – currID
         disable blending
         activate and clear transparentGBufferFBO[currID]
         pass depth buffer of transparentGBufferFBO[prevID] to geometry shader
         for each entity in transparentEntities:
                  geometryPass(entity, ...)
         enable blending and set front-to-back equation
         activate plainFBO
         lightingPass(…)
```

```
// [3] Lighting pass for opaque entities
enable blending and set front-to-back equation
activate plainFBO
lightingPass(...)
```

5. Blending correctly

In order to execute front-to-back blending correctly in OpenGL, some tweaks to blending settings have to be made.

OpenGL blending settings cannot fully reproduce the blending equation in section 2.3. Just before running any deferred lighting pass, the following blending settings are used:

```
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
glBlendFuncSeparate(GL_DST_ALPHA, GL_ONE, GL_ZERO, GL_ONE_MINUS_SRC_ALPHA)
```

These setting produce the following equation:

$$C_{dst} = A_{dst}\,(C_{src}) + C_{dst}$$
$$A_{dst} = (1 - A_{src})\,A_{dst}$$

This is the closest possible equation obtainable by OpenGL blending settings. It differs from the one in section 2.3 due to the lack of the multiplication between C_{src} and A_{src}. To compensate such difference, the RGB component of the final color computed in the lighting pass shader must be multiplied by the alpha value of such color [2].

Another precaution to be taken is the clear color with which the G-buffers, both transparent and opaque, are cleared. For clearing G-buffer FBOs, the clear settings have to be:

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

By setting the alpha value to 0, the lighting pass shader can discard background fragments with an alpha test[1] on the diffuse buffer of the given G-buffer. If no alpha test was executed, these settings would still be relevant since they guarantee correct blending during the lighting pass.

1 A test which discards fragments if their alpha value is below a (usually near-zero) threshold.

4. Conclusions

The final application is the result of researching how to effectively implement depth peeling in a deferred renderer. It was not a straightforward task, since both traditional literature and other resources discuss depth peeling in the classic context of forward shading. The final consideration on the work for the renderer are discussed in this chapter.

Aside from the rendering, some underlying systems were implemented to coherently maintain relevant data and to provide the user a way to customize the scene. These are briefly discussed, since the main focus of this report is the pipeline itself.

1. Application’s outcomes

The application successfully implements a deferred renderer. Transparency is handled with a version of depth peeling which is fully integrated in the whole deferred rendering process (Fig. 1).

Fig. 1: Transparency with deferred depth peeling.

The application offers a *view mode* (Fig. 1) which allows the user to navigate into the scene. It also offers an *edit mode* (Fig. 2) which allows the user to customize scene elements and rendering settings, such as lighting, material parameters and the number of depth peeling passes.

Fig. 2: Editing tool window.

An underlying system allows the user to modify the transparency of entities dynamically, without the need of building the program again. These parameters can be useful to inspect the quality of transparency rendering, as well as the overall performance.

2. Performance

Performance highly depends on three main factors:

- depth peeling passes;
- number of dynamic lights;
- framebuffers resolution.

As mentioned in section 3.3, a depth peeling pass consists into a whole deferred pipeline execution. That translates into performance drops when the number of depth peeling passes gets high. Moreover, the number of fragments processed at each depth peeling pass is relevant. Despite the alpha test mentioned in section 3.5 drastically improves performance, there are instances in which processed fragments at each peel is, inevitably, still pretty high; for instance, when the camera is close to a transparent object, a big chunk of transparent fragments needs to be processed, therefore the performance drops.

The intensity of dynamic lighting also impacts performance. Yet, the application of deferred shading greatly helps to keep reasonable frame rates even with a considerable amount of dynamic point lights displayed on screen.

These observations lean into implying that the chosen OIT technique is quality oriented. In fact, depth peeling is known to be an exact OIT method. For real applications, the proposed technique can be used while being mindful about the number of peeling passes executed.

One important achievement of this implementation is that memory costs are kept constant due to a clever usage of front-to-back blending. Besides, the deferred nature of the whole rendering process also helped to optimize the depth pre-pass of opaque objects discussed in section 3.4, which improved performance.

3. Further improvements

Currently, the shader used for the lighting pass of the deferred pipeline is a traditional GGX. Despite it shows good looking results, it lacks back-face illumination. That is, Fresnel highlights don’t show up when a dynamic light illuminates a surface from its back. This is clearly noticeable with transparent objects. This issue is not related to the renderer itself and can be solved with a thoughtful rework of the lighting shader.

Performance can be improved by finding a way to determine the number of depth peeling passes dynamically.

Currently, the number of color buffers used in the G-buffer is not minimal. In fact, there’s a dedicated position color buffer in each G-buffer. If memory minimization is a concern, this buffer can be omitted in the lighting shader by using the current layer depth buffer in conjunction with an inverse projection. Though, this route was not taken in order to minimize shader computations and to simplify data retrieval.

Lastly, the integration of dual depth peeling into a deferred renderer can be investigated, and compared to the single depth peeling implemented for such project.

5. Bibliography

[1] C. Everitt, *Interactive Order-Independent Transparency*, NVIDIA Corporation, 2001

[2] L. Bavoil, K. Myers, *Order Independent Transparency with Dual Depth Peeling*, NVIDIA Corporation, 2008