5
IBR on the GPU

In this chapter we describe our proposed implementation for IBR view synthesis, adapting the methods highlighted in the previous chapters. We begin by summarizing a basic conceptual algorithm and further develop it into a more efficient version by exploring current GPUs programmability.

5.1
Conceptual algorithm

Techniques explained in Chapters 3 and 4 lead to a complete solution for rendering novel views from depth images, whose conceptual algorithm is depicted in Figure 5.1. The first step is to read input depth images from the disk into main memory, along with cameras' matrices: calibration (K) and view (V). Then, view-dependent geometry is built for each input camera using provided depth maps, as described in sections 3.2 and 3.3.

Afterwards, virtual camera is configured using pair of adjacent cameras'

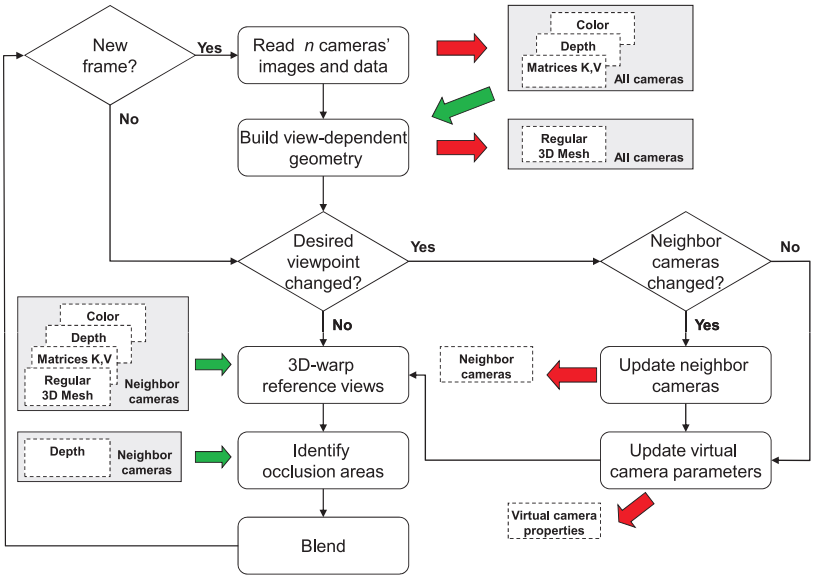

Figure 5.1: Conceptual algorithm for novel view synthesis.

data, using an interpolation process such as the one mentioned in Section 4.1.

Later, meshes for adjacent cameras can be warped into the virtual camera as highlighted in Section 3.3, and occlusion areas can be identified as shown in Section 3.4. Finally, the partial results are composited to generate the final rendered image as described in Section 4.2.

Following sections detail these steps and introduce modifications in the process so as to improve performance.

5.2 Creation of view-dependent geometry

The basic algorithm in the previous section suggests that view-dependent geometry be created for all cameras in a frame-basis. At every new frame, each of n input cameras' depth map with resolution WxH (assuming that all images have the same resolution) needs to be processed to yield a 3D mesh with WxH vertices, through method mentioned in Section 3.3. Later on the process, two out of those n meshes are sent to the GPU to be warped.

A key observation that lead to a economy both in memory footprint and in CPU-GPU transfer time follows. All generated meshes are regular and share the same X, Y coordinates for corresponding vertices, provided images resolution is the same for all input cameras. Only the Z coordinate, which comes from the depth map, change from mesh to mesh.

So, a straightforward optimization is to generate a single vertex buffer with a 2D mesh, with only X_i and Y_i coordinates (following notation defined in Section 3.3). Assuming images resolution is the same for every frame and every input camera, the mesh can be created only once and stored in the GPU memory.

At each new frame, color images and depth maps for pair of neighboring cameras are sent as textures to GPU, and the mesh defined in the vertex buffer is rendered once for each camera. A vertex shader is responsible for fetching the depth map at texture coordinates X_i, Y_i and determining the coordinate Z_w, through equations mentioned in sections 3.2 and 3.3.

In our implementation, we create a static OpenGL's vertex buffer object (VBO) [31] for storing the 2D mesh. With that simple modification, the mesh is transferred only once to the GPU in an initialization phase, yielding great improvement in rendering performance and transfer-time. Figure 5.2 summarizes the described process. In the initialization phase we also create a pair of target render textures (FBOs in OpenGL), which are used as temporary render targets for pair of input cameras warping results, which are blended later.

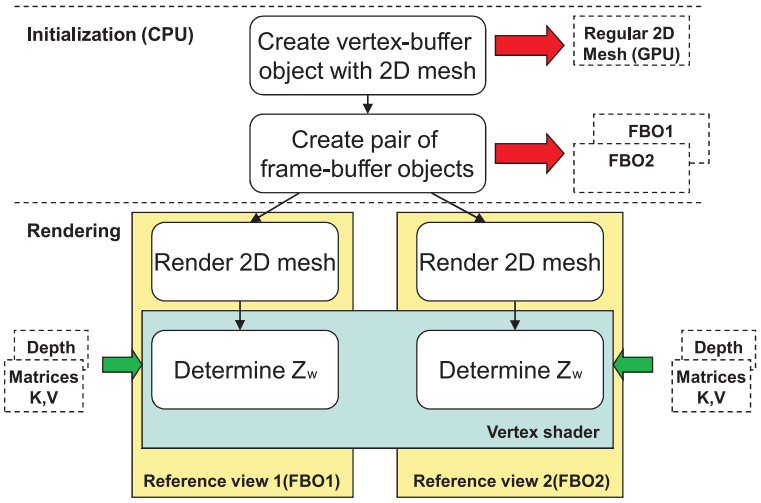

Figure 5.2: Vertex-buffer object used for improving performance during mesh generation.

5.3 Warping to novel viewpoint and occlusions identification

In our implementation, we use the same vertex shader mentioned in the previous section to also perform 3D warping and occlusion areas identification.

Inside the vertex shader, after the Z_w coordinate determination, we retrieve vertice's global coordinates (X_w, Y_w, Z_w) from (X_i, Y_i, Z_w) using expressions 3-7.

Finally, the vertice is projected into the novel viewpoint using virtual camera's calibration and view matrices, through equation 3-3, carrying the color information read from the color texture.

Besides, the same vertex shader is responsible for sampling the vertice neighbors for determining whether it belongs to an occlusion region or not, with the method defined in Section 3.4.

Therefore the input of the vertex shader is:

- Minimum and maximum depth values for Z_w unpacking (Section 3.2)
- Input camera projection matrix (K * V)
- Texture with color image
- Texture with depth map
- Virtual camera calibration and view matrix

It unpacks depth Z_w, unprojects vertice into global coordinate system, warps it into the virtual view and labels vertice as belonging to an occlusion area or not. In conclusion, the vertex shader output is:

- Occlusion label

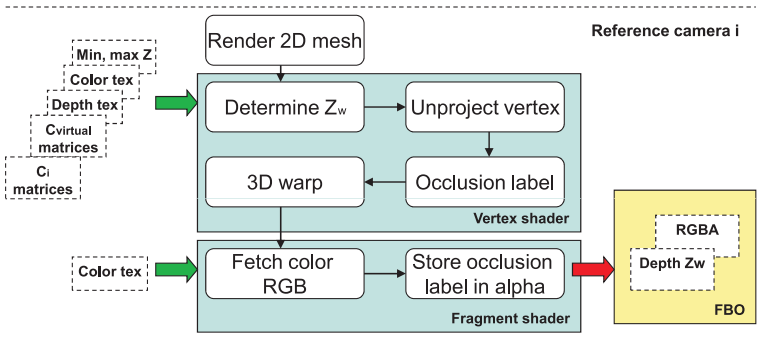

Figure 5.3: Reference view rendering into a FBO, using vertex and fragment shaders.

– Depth Z_w
– Texture coordinates for color texture, which will be interpolated by the hardware rasterizer.

A fragment shader in our implementation is responsible for rendering the final warped image inside a frame-buffer object, both color and depth data. It fetches the color texture with the input texture coordinates, stores occlusion label into color alpha channel and outputs depth Z_w, in global coordinates.

Figure 5.3 summarizes the process of rendering a single reference view into a temporary frame buffer.

5.4 Compositing on the GPU

After rendering reference views into separate buffers, we perform compositing: a full-screen quadrilateral is used to trigger a fragment shader, which in turn performs the necessary blending computations mentioned in Chapter 4, and outputs the pixel values directly to the screen.

The input of this blending fragment shader is:

– Virtual and references cameras position: used for angular distances computation (Section 4.2)
– Textures with color and depth data for warping results of reference cameras

Compositing is done in a similar fashion to the one described in Section 4.2. To meet the characteristics mentioned in that Section, here we propose and detail a penalty-based calculation of contribution weights for reference cameras.

The first desired characteristic for the compositing weights, namely angular distance influence on reference camera's weight, can be achieved by

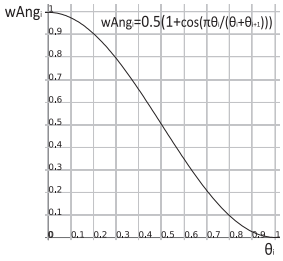

Figure 5.4: Influence of angular distance in reference camera's contribution during compositing.

using Equation 4-4 (Section 4.2). The behavior of that equation is depicted in Figure 5.4. The cosine-based equation guarantees a smooth variation of weights when changing the viewpoint, and also gives greater weight as the viewpoint gets closer to a reference camera.

Next we add a visibility test per-pixel, using a threshold τ to account for errors in calibration and depth estimation (refer to Section 4.2 for more details). By comparing depths for pixels p_i and p_{i+1} from cameras C_i and C_{i+1} (distance relative to the virtual camera), we can define visibility factors for both reference cameras:

$$closer(p_i) = \begin{cases} 1, & \text{if } Z_i - Z_{i+1} < -\tau \\ 0, & \text{otherwise} \end{cases}$$

$$closer(p_{i+1}) = \begin{cases} 1, & \text{if } Z_{i+1} - Z_i < -\tau \\ 0, & \text{otherwise} \end{cases}$$

The interpretation to use those factors is: when pixel p_i is much closer to the virtual camera than p_{i+1}, w_i should be increased, or decreased when p_{i+1} is much closer. Those observations yield the modified equation for weights (refer to Equation 4-4 for $wAng_i$ derivation):

$$w_i = clamp(wAng_i + closer(p_i) - closer(p_{i+1}), 0.0, 1.0) \qquad (5\text{-}1)$$

The missing part is the treatment of pixels marked occluded. For that we derive a penalty considering the following ideas:

- when pixel p_i is marked occluded, weight w_i should be penalized
- the penalty should not be applied when a reference camera almost coincides with the virtual camera, otherwise it should strongly affect that camera's weight

We propose a method based on angular distances to build a penalty term for pixels marked occluded. When the virtual camera is located very close to a

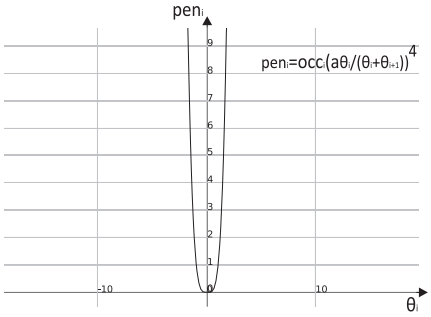

Figure 5.5: Penalty for pixels marked occluded based on angular distance.

reference camera, this camera's occlusion areas should not be heavily penalized, since they barely reveal any occluded areas. On the other hand, when virtual camera is far from a reference camera, areas marked as occluded certainly can expose unsampled areas, and therefore should be penalized in favor of the use of the other camera's data.

Equation 5-2 shows how this penalty is calculated in our implementation, using the same notation defined in Section 4.2:

$$\phi_i = occ_i(a\frac{\theta_i}{\theta_i + \theta_{i+1}})^4 \tag{5-2}$$

Its behavior is depicted in Figure 5.5. Coefficient a controls the increase due to angle distance (we used $a = 30$ in our implementation for the used datasets). occ_i corresponds to the occlusion label defined in previous Sections.

We incorporate this penalty to the weight equation to get the final weight value w_i for camera C_i, and Equation 5-1 turns into:

$$w_i = min(1.0, \phi_{i+1}+(1-\phi_i)clamp(wAng_i+\phi_i closer(p_i)-\phi_{i+1}closer(p_{i+1}), 0.0, 1.0)) \tag{5-3}$$

Basically the penalty for pixels marked occluded attenuates both the angular distance weight and the visibility test results. That equation summarizes the mentioned desired characteristics:

- when virtual camera is very close to a reference camera, the virtual image is almost identical to the reference one
- transitions are smooth and based on angular distances
- pixels marked occluded are treated differently to avoid rubber sheets on unsampled regions
- visibility ordering is enforced

Those weights are used to determine the final composited color as defined in equation 4-5 from Section 4.2 (since w_i is normalized, $w_{i+1} = 1 - w_i$).

Finally, we can summarize our proposed technique after considering all optimizations aforementioned in previous sections. Following algorithm overviews the steps involved in rendering of a novel view.

Phase 1: Initialization

1.1 Create and transfer vertex buffer object for a 2D mesh with the same resolution as input images

1.2 Create two frame buffer objects for temporary storage

1.3 Create textures, vertex shaders and fragment shaders

Phase 2: Rendering reference views separately

2.1 [**CPU**] Read input images and cameras' data

2.2 [**CPU**] Update virtual camera's position and orientation

2.3 [**CPU**] Determine neighbor cameras i and $i+1$

2.4 [**CPU**] Activate FBO1 as render target

2.5 [**GPU**] Render camera i (3D warping and occlusion labeling)

2.6 [**CPU**] Activate FBO2 as render target

2.7 [**GPU**] Render camera $i+1$ (3D warping and occlusion labeling)

Phase 3: Compositing

3.1 [**CPU**] Use screen as render target

3.2 [**CPU**] Draw a full-screen quadrilateral to trigger compositing fragment shader

3.3 [**GPU**] Compute angular distances, penalties and weights

3.4 [**GPU**] Composite color from reference views using final computed contribution weights