공학박사 학위논문

GPU 기반의
실시간 포비티드 렌더링

2019년 8월

부경대학교 대학원

IT 융합응용공학과

권 오 석

공학박사 학위논문

GPU 기반의
실시간 포비티드 렌더링

지도교수 김 영 봉

이 논문을 공학박사 학위논문으로 제출함.

2019년 8월

부경대학교 대학원

IT융합응용공학과

권 오 석

권오석의 공학박사 학위논문을 인준함.

2019년 8월 23일

위 원 장 공학박사 김 종 남 (인)

위 원 공학박사 신 봉 기 (인)

위 원 이학박사 김 형 석 (인)

위 원 공학박사 조 장 우 (인)

위 원 공학박사 김 영 봉 (인)

목 차

그 림 목 차

표 목 차

GPU based Real-Time Foveated Rendering

Oh-Seok Kwon

Dept. of IT Convergence and Application Engineering,
The Graduate School,
Pukyong National University

Abstract

Today's Head-Mounted Displays and high-resolution display above 4K provide a high visual experience and an immersive experience for users. However, the rendering on high resolution displays needs efficient rendering techniques enough to give a real-time. One of such techniques is a foveated rendering combined with gaze-tracking that minimizes the loss of perceptual detail.

Therefore, we propose a GPU based real-time foveated rendering technique. The proposed method operates in a way that imitates the human visual system. We designed the sampling map considering the human retina, visual attention, and efficiency of reconstruction. Especially, it can be designed to effectively work on the GPU, and also reduces the computational cost that has been improved by previous foveated sampling. In addition, we use a combination of ray tracing and temporal reprojection to generate a complemented sparse pixel image. Finally, we generate a dense image through the reconstruction process. We also show real-time foveated rendering pipelines for various purpose that are effectively mapped on the GPU.

As a result, the proposed method allows high-quality generated images in real time on a high-resolution display with minimizing loss of perceptual detail.

Ⅰ 서론

넓은 시야(FOV) 및 고해상도를 갖춘 HMD(Head-Mounted Display) 장치의 개발은 가상 현실(Virtual Reality: VR)을 통한 사용자 경험의 기회를 획기적으로 증가 시켜왔다. HMD를 기반으로 하는 많은 VR 어플리케이션이 상용화되어 서비스되고 있으며, 더욱 다양한 분야, 다양한 컨텐츠의 형태로 사용이 증가하고 있다.

기본적으로 VR 어플리케이션에서는 어지러움을 피하기 위해 높은 프레임 속도와 낮은 대기 시간을 요구한다 [1]. 또한 오늘날의 HMD는 높은 해상도 또는 스테레오 디스플레이로 구성되며, 추가적으로 모니터 등의 다양한 디스플레이 장치는 4K를 넘어 더욱 높은 해상도로 고품질의 영상을 제공하고 있다. 그러나 이러한 고품질의 영상은 뛰어난 시각적 경험을 제공하지만, 높은 해상도로 인해 렌더링에 소요되는 계산비용 또한 비례하여 증가한다.

실시간을 보장하기 위해 전통적으로 래스터화 기반의 렌더링(Legacy Rendering) 기술이 발전되어왔지만 최근에는 GPU의 발전과 더불어 물리기반 렌더링(Physically Based Rendering: PBR)이 개발되었다. 그러나 PBR은 여전히 고성능을 요구하기 때문에 인간 시각 시스템과 유사한 포비티드 렌더링 기술을 접목하면 이를 개선할 수 있다. 따라서 적응적인 해상도를 생성하는 포비티드 렌더링 기술은 계산 비용과 전력 소비를 줄일 수 있는 장점으로 인해 많은 연구가 진행되고 있다 [1]-[4].

지금까지 HMD 및 일반 디스플레이 장치에서의 렌더링 기술은 주로 계산 성능으로 인한 이유로 래스터화 기반의 방법이 적용되어왔다.

반면에 광선 추적법은 매우 사실적인 고품질 이미지를 생성 할 수 있는 장점을 가진다 [3]. 또한 여러 물리적인 광학적 기법을 쉽게 구현할 수 있는 장점이 있다. 그러나 광선 추적 방법은 래스터 화에 비해 높은 계산 비용을 필요로 하기 때문에 실시간 렌더링에는 활용되지 못했다. 그러나 Koskela [4]등의 연구에 따르면 망막의 포베아 영역에 시세포가 밀집되어 있다는 사실을 활용하는 방법을 찾기 시작하였다. 또한 중심시와 주변시로 구분하여 전체 해상도의 약 94 %가 생략되어도 인간의 시각 시스템은 이것은 인지할 수 없는 경우가 존재한다는 사실을 확인하였다. 따라서 광선 추적법은 시선추적과 결합한 포비티드 렌더링으로 HMD 및 고해상도 디스플레이의 렌더링 요구 사항을 충족시킬 수 있다. 또한 고성능을 요구하는 많은 장치에 포비티드 렌더링을 적용하면 계산 성능을 향상시킬 수 있다.

포비티드 렌더링은 인간 시각 시스템과 밀접한 관련이 있다. 건강한 성인 인간의 시야각은 눈당 수평으로 150°, 수직으로 135°이며, 중심시에 해당하는 망막의 중심에서부터 18°까지의 영역에는 광 수용체가 매우 밀집해 있는 중심와(Fovea: 포베아)가 존재한다 [5]. 광 수용체 세포는 원추 세포와 간상세포로 구분되며, 먼저 색상을 구분하는 원추 세포는 포베아 영역에서 높은 밀도로 존재하며, 주변부로 갈수록 급격히 감소한다. 한편, 주변부 영역은 원추 세포의 낮은 밀도로 인해 색상을 구분하는 능력은 떨어지나, 빛의 세기에 대해 민감하게 반응하는 간상세포가 있다 [6]. 주변시 영역의 간상세포는 약한 빛에도 민감하게 반응하며 사물의 형태 및 빛의 세기를 감지하는 역할을 하며, 특히 작은 깜박임에도 민감한 반응성을 보인다 [6]. 따라서 이러한 해부학적 사실에 근거하여, 주변시 영역을 대략적으로 표현하더라도 인간의 눈은 이를 쉽게 구분하기 어렵다 [7]. 이러한 인간 시각 시스템을 모방

한 포비티드 렌더링에 대해 많은 연구가 진행되어 적응적인 해상도로 렌더링하는 방법을 통해 계산 비용을 줄이게 되었다. 그러나 이러한 장점과 더불어 Patney [8]는 사용자 연구에 의해 따르면 주변시 지역이 지나치게 대략적으로 렌더링할 경우 갈수록 시야가 좁아지는 터널시야 현상이 발생한다는 시각적 이슈를 확인하였다.

이러한 문제에도 불구하고 포비티드 렌더링으로 인한 적응적인 해상도는 샘플링 확률에 따라 비례적으로 성능향상을 기대할 수 있다. 그러나 샘플링 확률에 따른 해상도의 감소와 더불어, GPU의 작동원리에 따른 추가적인 최적화 가능성이 있다. 현대의 GPU는 warp(스레드 그룹) 단위로 작업을 수행하도록 설계되어 이 작업 단위를 고려하면 동일한 샘플링 상황에서 계산 시간을 더욱 줄일 수 있다 [9].

따라서 본 논문에서는 GPU 기반의 실시간 포비티드 렌더링 기법을 통한 VR을 위한 고품질 영상 생성 방법을 제안한다. 제안한 방법은 인간의 시각 시스템을 모방하는 방식으로 동작하며, 인간의 망막 및 시각적인 주의, 그리고 재구성 단계에서 드는 계산 비용을 고려하여 샘플링 맵을 설계하였다. 특히 GPU에서 더욱 효과적으로 동작할 수 있도록 설계된 샘플링 맵은 이전의 포비티드 샘플링으로 개선된 계산 비용을 추가적으로 줄일 수 있다. 이는 포비티드 샘플링에 의해 수행되는 광선 추적법을 보다 효과적으로 수행할 수 있는 계산상의 이점을 제공한다. 포비티드 샘플링의 각 픽셀에서 병렬적으로 수행되는 광선 추적 스레드를 Compressed Sparse Row(CSR)의 형태로 압축하는 방법으로 최적화하였다. 또한 광선 추적법(ray tracing) 뿐만 아니라 템포럴 리프로젝션(temporal reprojection) [10]을 복합적으로 사용하여 샘플링으로 인한 희소 픽셀 이미지를 보완하고 마지막으로 재구성 단계를 통해 최종적으로 빽빽한 이미지를 출력한다. 세포의 구조가 다각형 형태로 가

정하여, 희소한 픽셀의 빈 부분을 보로노이 (voronoi) 다이어그램 형태로 채우고, 실시간 자연 이웃 보간법을 통해 보로노이 셀 간 경계를 부드럽게 처리하는 접근법을 통해 재구성하였다. 제안한 방법을 통해 터널시야 현상을 막으면서 고비용의 광선 추적법을 실시간으로 사용할 수 있는 VR을 위한 고품질의 영상을 기대할 수 있다.

이 논문은 다음과 같이 구성된다. 2장에서는 관련 연구에 대해 논의하고, 3장에서는 제안한 방법에 대해 상세히 기술한다. 4장에서는 구현에 대해 서술하고, 실험 결과와 결론은 각각 5절과 6절에서 설명한다.

II 관련 연구 및 배경

본 장에서는 인간 시각 시스템과 광선 추적법에 대해 기술하고, 포비티드 렌더링의 이전 연구들을 자세히 살펴보도록 한다.

1. 인간 시각 시스템

인간의 시각 시스템을 파악하기 위해서는 우선적으로 그림 1의 인간의 눈의 해부학적 구조와 작동원리에 대한 이해가 필요하다.

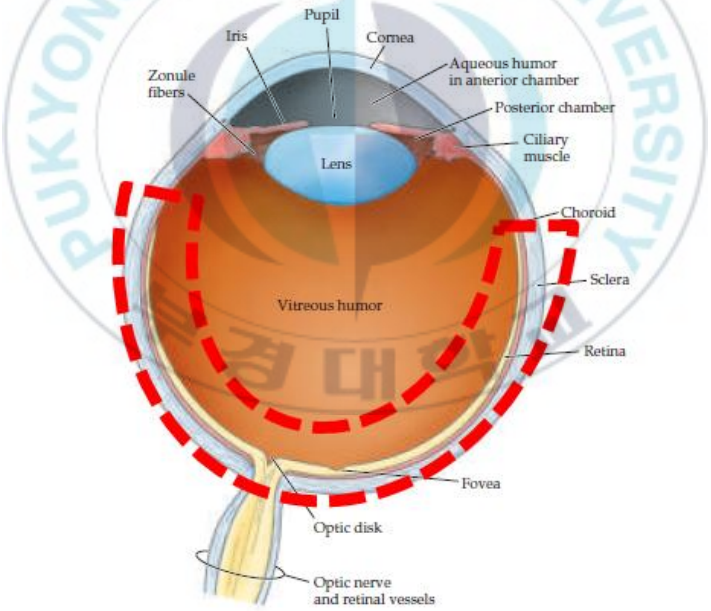

그림 1. 인간의 눈의 해부도, 빨간색 영역은 망막

눈은 크게 렌즈와 센서로 구분된다. 먼저 렌즈의 역할에 해당하는 각막과, 수정체로 구성되며 홍채를 통해 동공의 넓이를 조절하여 눈을

통과해 들어오는 빛의 양을 조절한다. 그리고 센서는 망막으로써 광수용체 세포로 구성되어있다. 인간의 망막이 디지털 카메라와의 다른 점은 디지털 센서는 균일하게 배치되는 것에 비해 망막은 초점이 맺히는 중심와(Fovea: 이하 포베아)을 기준으로 그림 2와 같이 비균일하게 분포한다 [11].

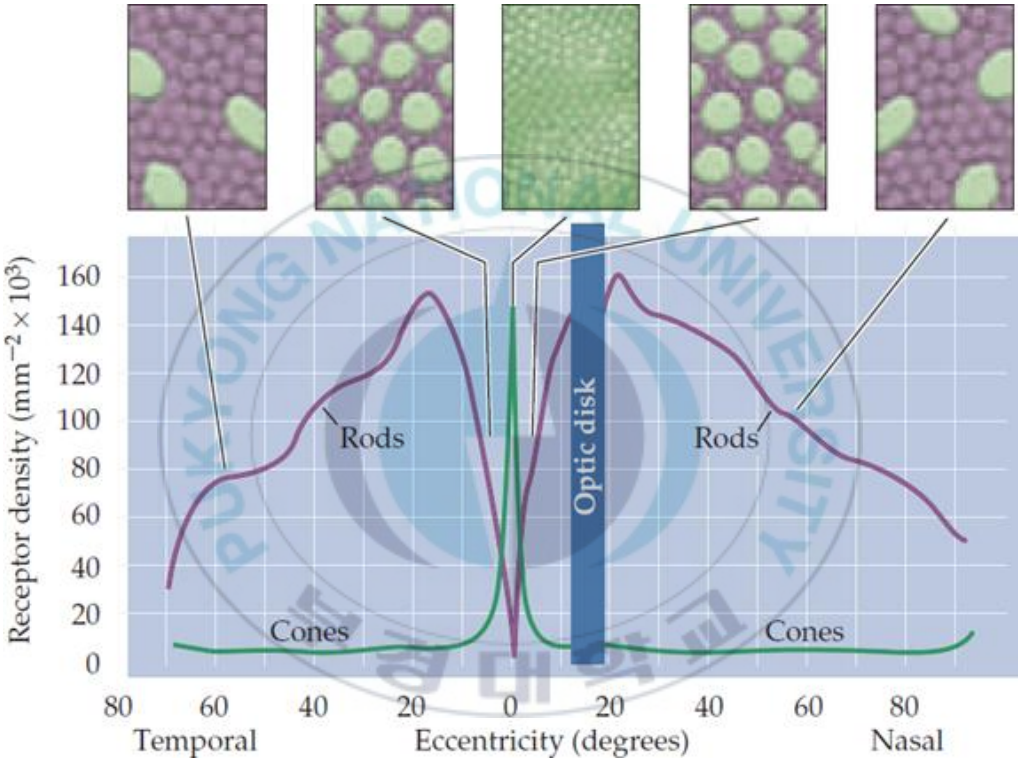

그림 2. 이심률에 따른 광수용체 세포의 밀도

이 망막을 구성하는 광수용체 세포는 간상(rods)과 원추(cones) 세포의 두 종류로 구분된다. 먼저 원추세포의 경우 적, 청, 녹을 감지하는 세 타입으로 구성되고, 그림 2와 같이 대부분은 포베아 부분에 분포하며 주변부로 갈수록 그 수가 급격하게 감소한다. 그리고 막대세포는 약한 빛에도 민감하게 반응하는 역할을 수행하며 중심와 주위의 약 18

도 부근에서 링형으로 최고로 밀집하며 주변부로 갈수록 최대 밀도의 20%까지 감소한다. 따라서 인간의 시각 시스템은 초점의 중심부를 기준으로 20도 내에 대부분의 시세포가 밀집된 포베아 영역에 상이 맺히도록 렌즈의 수축도를 조절하는 방식으로 동작한다 [12]. 아래 그림 3은 이심률에 따른 중심시와 주변시로 구분된 눈의 해상도를 묘사한 그림이다.

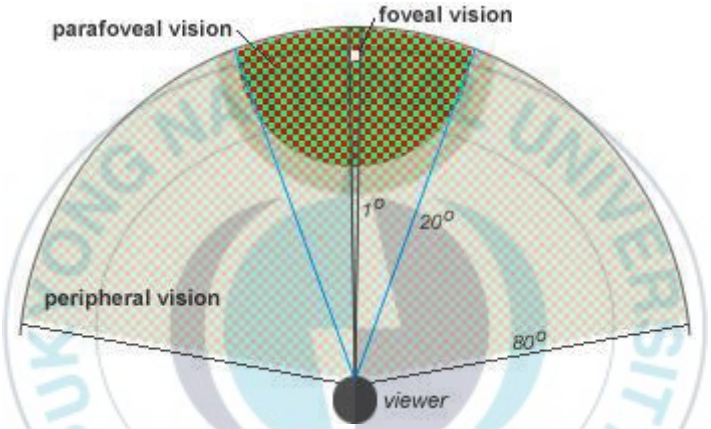

그림 3. 눈의 해상도

2. 광선 추적법

광선 추적법은 그림 4와 같이 사실적인 렌더링 이미지를 생성할 수 있는 효과적인 방법이다. 광원을 물리적으로 추적하여 물체의 성질을 고려하여 카메라로 들어오는 방사량을 수식 1와 같이 계산한다.

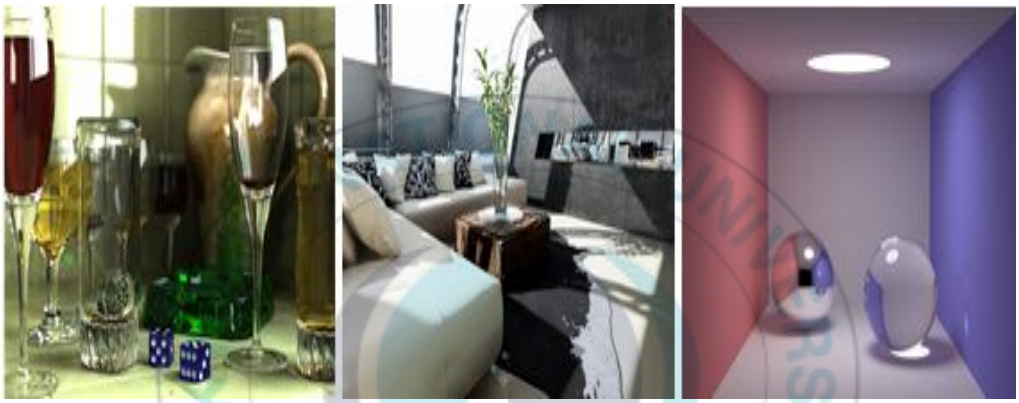

그림 4. 광선 추적법을 통한 다양한 장면의 렌더링 결과

$$L_o(x,\vec{w}_o) = L_e(x,\vec{w}_o) + \int_S f(x,\vec{w}_o,\vec{w}_i) L_i(x,\vec{w}_i) \left|\cos(\theta_i)\right| dw_i \qquad \text{수식 1}$$

수식 1의 각각의 변수를 정리하면 다음과 같다:

x	:	표면의 위치,
w_o	:	광원으로 나가는 방향,
w_i	:	광원으로부터 들어오는 방향,
L_o	:	표면에서의 나가는 방사도,
L_e	:	방출되는 방사도,
L_i	:	들어오는 방사도,
f	:	양방향 반사 분포 함수 (BRDF),
S	:	표면을 중심으로한 구,
θ_i	:	입사광 방향과 표면 법선사이의 각도

수식 1에서 $L_e(x,\overrightarrow{w}_o)$는 해당 표면의 물체가 스스로 발산하는 반사광을 나타내며, $L_i(x,\overrightarrow{w}_i)$는 광원으로부터 들어오는 입사광을 나타낸다. 위 수식은 오직 반사 성질을 가지는 물체를 표현하는 수식이며, 그림 5에 묘사된 그림과 같이 설명되어진다.

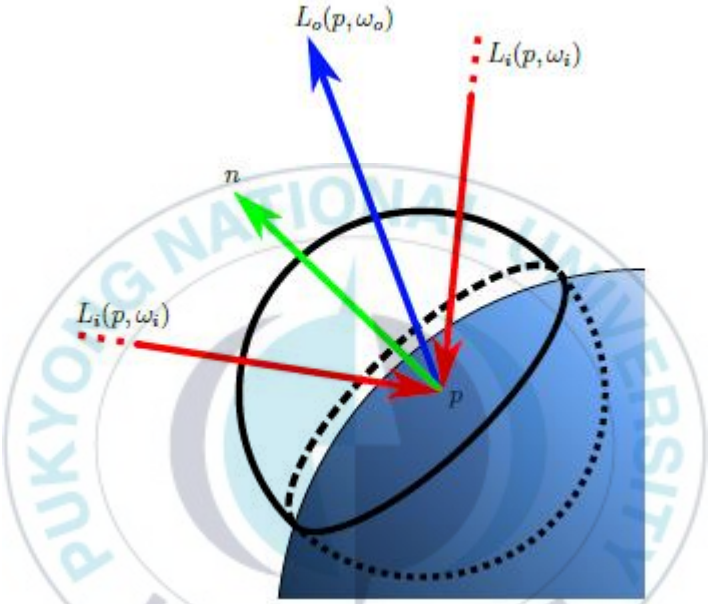

그림 5. 양방향 분포 함수

위 수식의 적분 안에서 양방향 반사 분포 함수 (BRDF)는 표면에서부터 각 입·출 방향으로 빛이 산란되는 방법을 결정한다. 이것을 통해 여러 가지의 표면의 성질을 모델링할 수 있게 된다. 그리고 입사광은 입사광의 방향과 표면의 법선 사이의 각도에 의해 조절된다. 여기서 난반사 표면 성질을 가지는 물체를 표현하려면 L_e 또한 적분하여 구하면 된다. 결과적으로 수식 2.n을 통해 카메라로 들어오는 빛의 세기를 계산할 수 있고, 최종적으로 3차원의 물체를 2D 이미지로 렌더링 할 수 있다.

그러나 위 수식의 적분에 대한 일반적인 닫힌 해는 존재하지 않기 때문에 분석적 방법을 통해서는 해를 구할 수 없다. 따라서 확률론적 접근법을 통해 이 문제를 해결할 수 있으며, Kajiya [13]의 경로 추적법을 사용하면 적분 문제를 해결할 수 있다. 경로 추적법은 무작위로 생성된 광선을 추적하고, 광선과 표면의 교차점에서 계산한 방사도의 평균을 통해 근사하는 방식으로 동작한다. 그러나 경로 추적법은 단순하고 일반적인 접근법이지만 결과에 수렴하기 까지는 느릴 수 있다. 또한 확률적인 방법으로 동작하기 때문에 교차점까지 도달하는데 소요되는 처리시간이 각 광선마다 다를 수 있다.

3. 포비티드 렌더링

포비티드 렌더링의 기본적으로 샘플링과 셰이딩 그리고 재건과정으로 구분된다. 먼저 이전 연구에서 다루어진 샘플링은 인간 시각 시스템를 기반으로한 Adaptive Sampling과 특정 패턴을 기반으로 한 Fixed-based Sampling, 인지를 기반으로 하는 Perception-based Sampling으로 나뉜다.

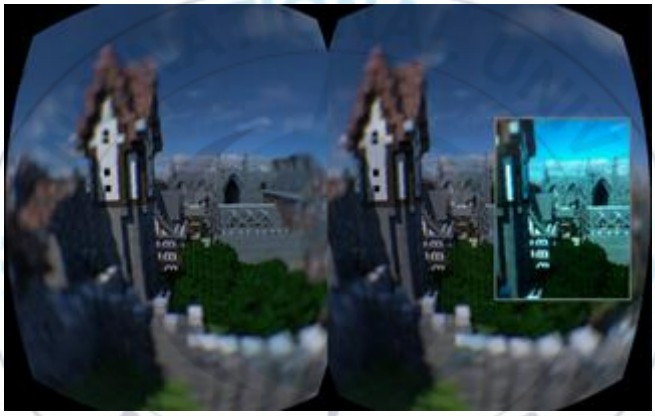

그림 6. Fujita(2014)의 스테레오 렌더링 파이프라인의 결과

Fujita(2014) [14]는 사전 계산된 고정 샘플링 패턴을 사용하여 광선 추적법의 계산 비용을 줄이고, 보로노이 구조화 후 인접 셀과의 k-NN 보간법을 통해 희소 이미지에서 빽빽한 이미지로 재구성합니다. 여기에 사용 된 사전 계산 된 샘플링 패턴은 보로노이 구조에서 k-NN 보간법의 성능을 보존하기 위해서 인접 시드의 정보를 미리 알고있는 방법을 통해 계산 비용을 절감한다. 또한 그림 6와 같이 양안을 고려해 스테레오를 위한 렌더링 파이프라인을 제안하였다. 그러나 고정 샘플링

패턴을 사용하기 때문에 템포럴 리프로젝션과 같은 기술을 사용하여 얻은 데이터로는 보로노이 구조를 형성할 수 없다는 단점이 있다. 따라서, 추가적인 샘플링을 수행하더라도 k-NN 보간을 위해 보로노이 구조로 재구성되어야하는데, 이는 많은 비용이 소요될 수 있다.

그림 7. Patney(2016)의 대비 보존 포비티드 렌더링

Patney(2016)는 [8] 시각적 인지를 기반으로 가우시안 필터링으로 인한 블러 효과가 터널 시야 효과를 유도한다는 것을 확인했다. 인간의 주변 시야에서 시각적 품질을 유지하는 것에 접근하여 이 문제를 해결했다. 이 문제를 해결하기 위해 그림 7와 같이 주변시 영역에서의 대비를 보존하는 방법을 통해 이러한 문제를 해결하였다. 또한 대비를 보존하는 방법을 통해 주변시 부분에서 더욱 강한 블러 효과에도 사용자는 큰 차이를 알 수 없음을 발견하였다.

그림 8. Weier(2016)의 렌더링 결과

그리고 Weier(2016) [15]는 포비티드를 통해 실시간 광선추적법을 적용하고, 재구성시의 품질 향상을 위한 파이프라인을 제안하였다. 그림 8의 빨간 사각형의 주변시 영역에서 낮은 샘플링 확률로 인한 모델의 경계에서 발생하는 잡음을 해결하였다. 그러나 경계가 매우 많이 드러나는 모델을 렌더링할 경우에는 리샘플링 대상이 늘어나 계산 비용이 증가한다는 문제가 존재한다. 또한 포비티드 샘플링으로 인한 성능 향상을 기대하지만 역설적이게도 GPU의 작업단위를 고려하지 않기 때문에 계산 비용이 절약되지 않을 가능성이 있다.

III Our Foveated Rendering

1. 개요

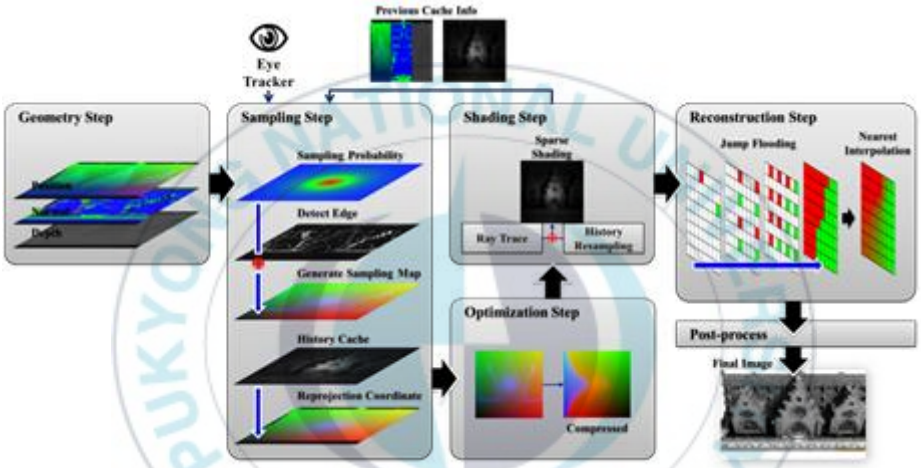

그림 9. 렌더링 개요

우리는 그림 9와 같은 절차를 통해 최종적으로 포비티드 이미지를 생성한다. 각 단계는 샘플링, 최적화, 셰이딩, 재구성의 절차로 구분되어 수행되며, 입력 데이터로는 시선 추적의 스크린 좌표와 동공의 너비를 사용하였다.

포비티드 렌더링에서 최종적으로 생성할 이미지는 인간 시각 시스템을 모방하여 중심시와 주변시로 구분된다. 이와 같은 방법은 시선추적을 통해 중심시는 고품질로, 주변부는 이미지의 품질을 낮춤으로써 렌더링에 필요한 계산시간을 줄일 수 있다. 따라서 먼저 시선추적을 바

탕으로 중심시와 주변시를 구분하여 처리할 수 있도록 샘플링 맵을 생성한다. 이때 지연렌더링(Deferred Rendering)과 같은 절차로 그림 10의 G-buffer의 각 버퍼를 활용한다. 그런 다음 샘플링 맵를 바탕으로 셰이딩을 수행하여 중심시와 주변시로 구분되어 적응적으로 픽셀이 채워진 이미지를 생성한다. 광선추적법을 통해 픽셀의 값을 구하기 전 GPU에서 효율적으로 동작할 수 있도록 최적화 단계를 통해 샘플링 맵을 재정렬한 후 수행한다. 이 단계를 통해 획득한 이미지는 샘플링 확률에 의해 데이터 측면에서 희소한 특성을 가지므로 재구성 절차를 통해 최종적으로 채워진 포비티드 이미지를 생성한다. 각 단계에 대한 자세한 설명은 다음 각 절에서 기술한다.

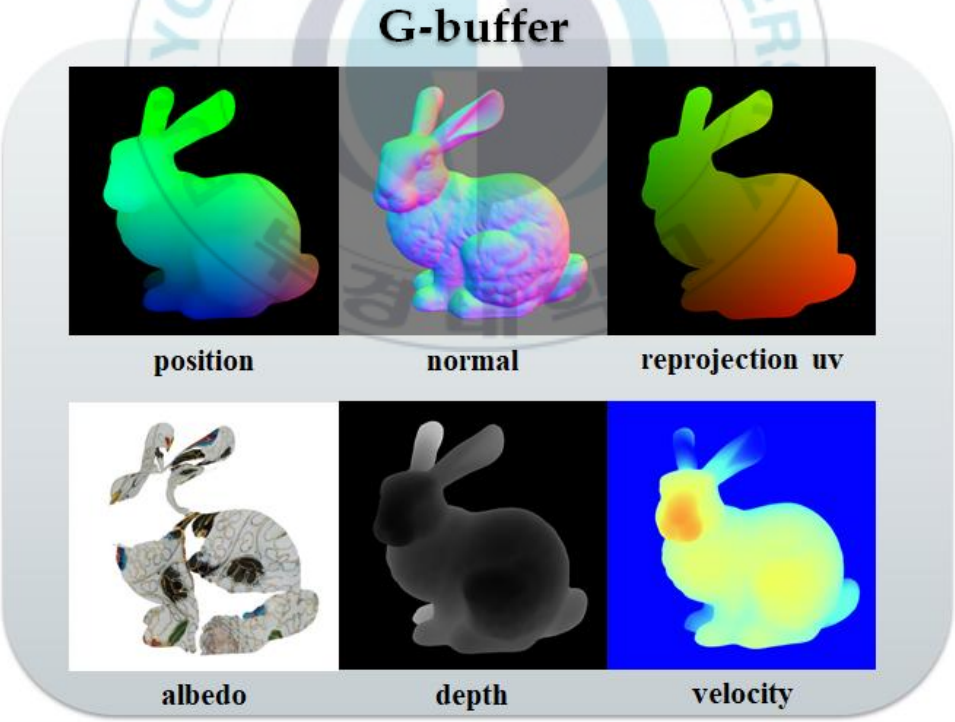

그림 10. 지연렌더링 방식으로 구한 G-buffer

2. 샘플링

샘플링 단계에서는 포베이션 함수가 적용된 샘플링 맵을 생성한다. 추가로 템포럴 리프로젝션을 수행하기 위한 리프로젝션 맵을 이 단계에서 함께 수행한다.

2.1. 샘플링 맵

포비티드 렌더링을 위한 샘플링 맵을 생성하기 위해서는 시선추적이 필수적이다. 따라서 시선추적 장비를 통해 얻은 스크린 좌표 상의 시선의 위치를 기준으로 다음 그림 11와 같은 절차를 통해 샘플링 맵을 구한다. 제안한 파이프라인에서 사용한 샘플링 맵은 시선을 중심으로 포베이션 확률에 따른 중심시와 주변시를 구분하고, 시각적 인지를 고려한 세일리언시 맵을 합성하여 생성한다.

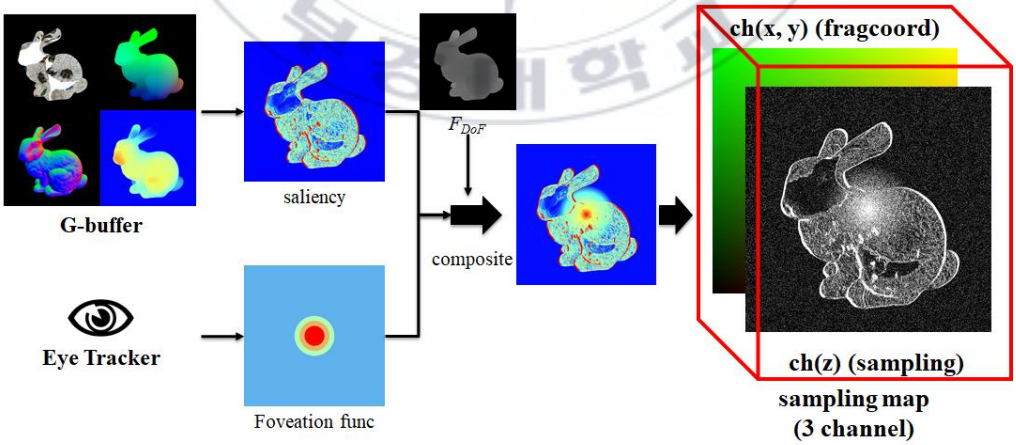

그림 11. 샘플링 맵 생성 과정

(가) 포베이션 확률

샘플링 맵은 인간이 망막을 통해 인지하는 시력의 해상도를 나타낸다. 인간의 눈의 해상도는 일반적으로 망막에 위치하는 시세포의 분포로 결정된다. 그러나 인간의 시력은 Strasburger [16]의 연구에 따르면 해부학적 데이터(광수용체 밀도)뿐만 아니라 최소 해상각(MAR: Minimum Angle of Resolution)에 따라서 인간의 시력이 결정된다. 따라서 망막의 중심을 기준으로 이심률의 변화에 따른 MAR은 아래 그림 12와 같이 선형 모델로 표현할 수 있다.

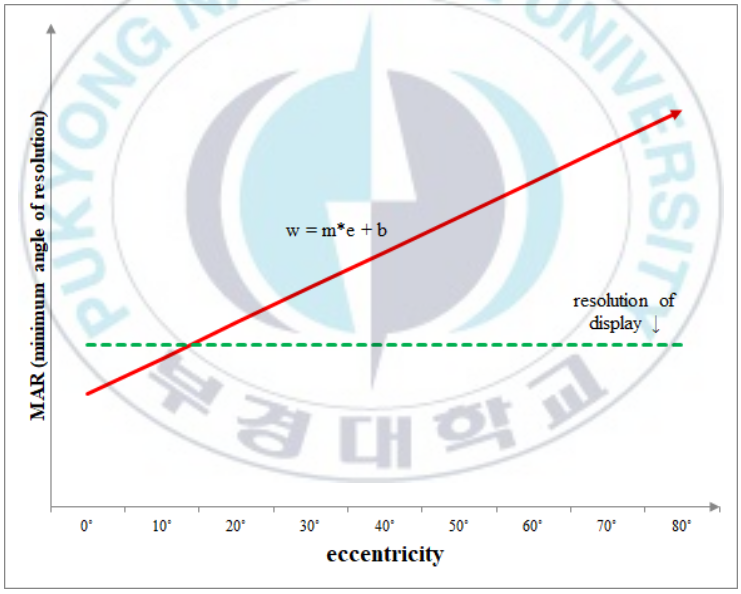

그림 12. 이심률에 따른 최소 해상각

위 그림 12에서 w는 이심률에 따른 해상각이고, e는 이심률, m는 MAR 기울기, b는 포베아(e = 0)에서의 최소 해상각이다. 여기서 이심률 e는 다음과 그림 13과 같이 구할 수 있다.

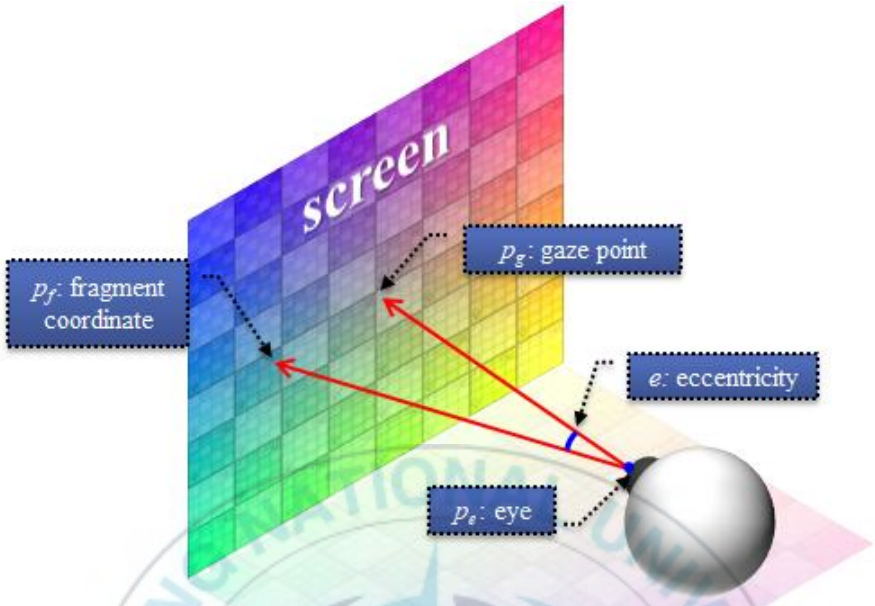

그림 13. 프래그먼트에서의 이심률 e의 계산

위 그림 13와 같이 시선 추적위치 p_g와 눈의 위치 p_e를 통해 모든 프래그먼트 좌표 p_f와의 사이각을 계산하여 중심시에서부터 주변시까지의 모든 이심률을 GPU에서 병렬적으로 계산한다. 따라서 그림 13을 통해 구한 최소 해상각 w는 두 픽셀을 구별하는 최소 분해능을 나타내므로 아래의 수식 2로 샘플링 확률을 구한다.

$$p(w) = clamp(1 - \frac{(w - w_{\min})}{w_{far}}, \xi, 1.0) \qquad \text{수식 2}$$

이심률에 따른 해상각 w을 입력으로 디스플레이의 최소해상각 $w_{\min}$과 최고 이심률에 따른 해상각 w_{far}으로 정규화하여 처리한다. ξ는 최소 확률 값이다. 이 최소 확률은 차후 수행할 재구성 단계의 성능을

고려하여 선택적으로 지정한다. 아래 그림 14는 위 수식 2를 나타낸 그림으로 중심시(central vision)와 주변시(mid, far-peripheral)로 구분되는 것을 확인할 수 있다.

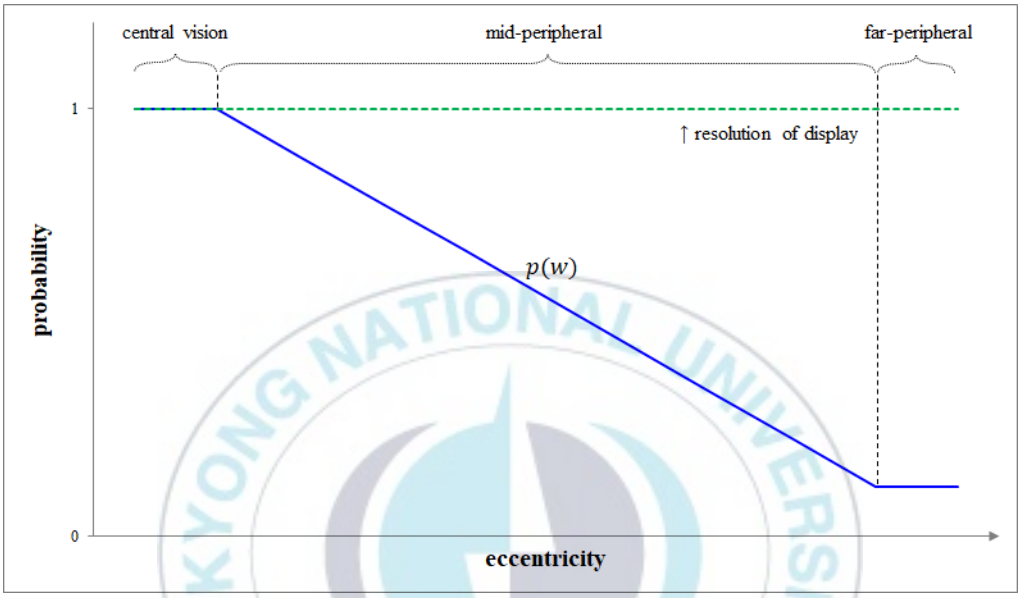

그림 14. 이심률에 따른 샘플링 확률

이 포베이션 확률 함수는 광수용체 세포의 밀도와 광수용체 세포로 진입하는 빛의 각도에 의해 결정된다는 사실에 근거한다 [17]. 따라서 샘플링을 수행하기 위해 시선의 위치, 사용자 정의 된 계수 값을 기반으로 한다. 아래 그림 15은 샘플링 수행의 결과를 보여준다.

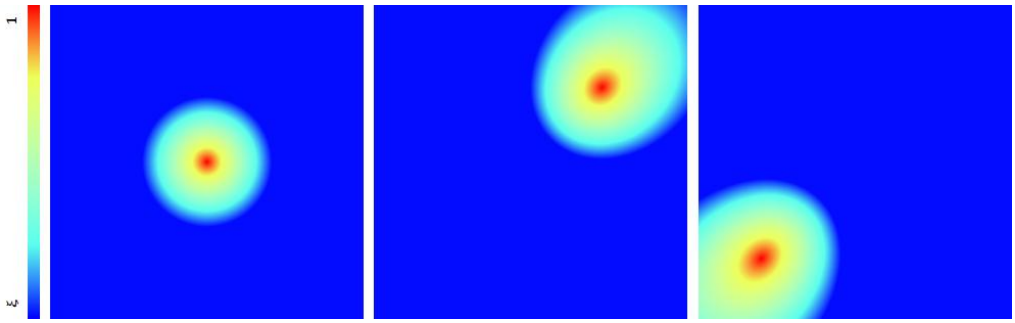

그림 15. 포베이션 확률함수의 결과

(나) 시각적 인지

주변시 영역에서의 언더 샘플링은 렌더링의 계산시간을 감소시킨다. 그러나 이러한 장점에도 불구하고 Patney의 연구에 따르면 주변시 영역에서의 언더 샘플링으로 인한 블러효과는 시각적으로 터널시야 현상이 발생할 수 있음을 확인하였다 [8]. 이러한 터널시야 현상은 주변시 영역의 사물이 흐리게 처리되어 더욱 시야가 좁아지게 된다. 따라서 이러한 터널시야 현상을 줄이기 위해서 지각적 모델인 세일리언시 맵을 적용하였으며, 생성절차는 다음 그림 16와 같다.

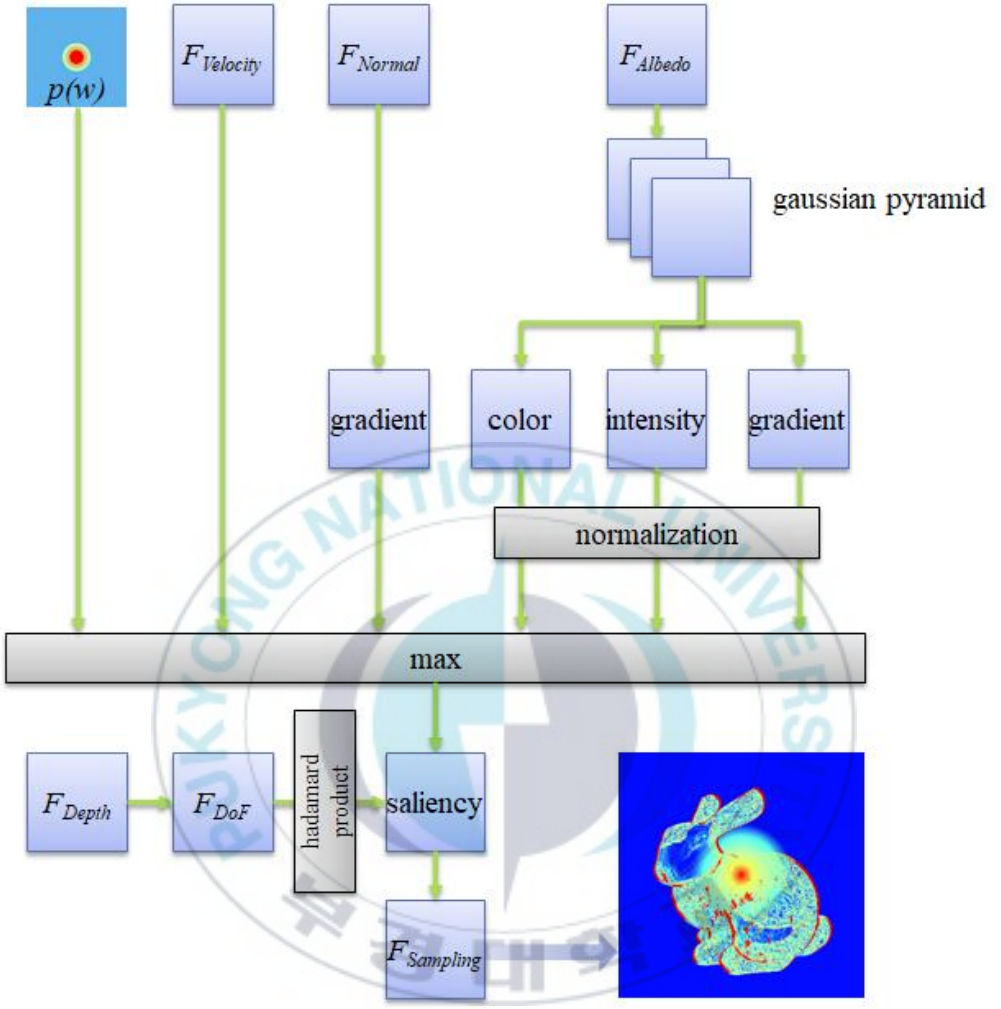

그림 16. 샘플링 맵 $F_{Sampling}$의 생성 절차

세일리언시 맵을 생성하기 위해서 G-BUFFER로부터 알베도, 노멀, 깊이, 가속도 버퍼의 정보를 활용한다. 각 버퍼로부터 색상, 밝기, 윤곽선 등의 특징을 합산하여 시각적 주의를 집중시키는 특징 정보를 얻는다. 이때의 연산은 인간의 시각체계와 유사한 CIE-Lab 컬러 공간으로 변환하여 수행한다. 세일리언시는 수식 3와 같이 각각 알베도, 노

멀 맵에 대해 3x3 커널 크기의 그라디언트 필터를 통해 편미분의 최대 값으로 세일리언시를 구한다.

$$F_{Sampling} = \max(\Delta F_{Albedo}, \Delta F_{Normal}, F_{Velocity}, p(w)) * F_{DoF} \qquad \text{수식 3}$$

여기서 인간의 눈은 빠르게 움직이는 물체에 대해 민감하게 반응하므로, 가속도 버퍼 ($F_{Velocity}$)를 사용하여 추가적으로 샘플링 확률을 높인다. 또한 사용자의 몰입감을 높이기 위해 깊이 버퍼를 통해 생성한 피쳐 맵 F_{DoF}를 통하여 경우 피사계 심도 효과를 간단하게 처리한다. 피사계 심도는 초점 거리가 멀수록 기하급수적으로 증가하기 때문에 정밀한 계산보다는 다음 그림 17와 같이 처리하여 계산시간을 줄인다.

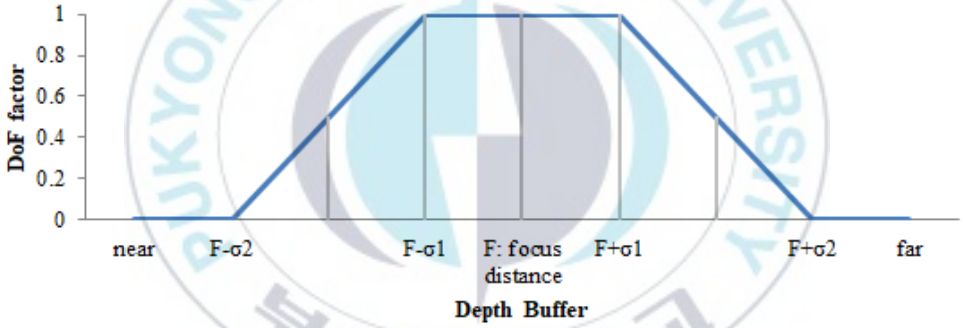

그림 17. 피사계 심도 계수 함수

여기서 초점 위치 F를 기준으로 초점 범위 계수 $\sigma 1$와 선형보간 범위 계수 $\sigma 2$로 피사계 심도 계수를 결정한다. 정규화된 F_{DoF}를 세일리언시에 곱함으로써 초점범위 밖의 세일리언시에 대해서는 제외하는 전략을 통해서 포비티드 렌더링의 성능을 유지한다. 그림 18는 시각적 주의와 피사계 심도가 적용된 샘플링 맵과 셰이딩 결과이다. 샘플링 맵은 최소 확률 [ξ ~ 1]로 구성되며 최종적으로 0 또는 1로 샘플링 맵의 3번째 채널에 저장한다.

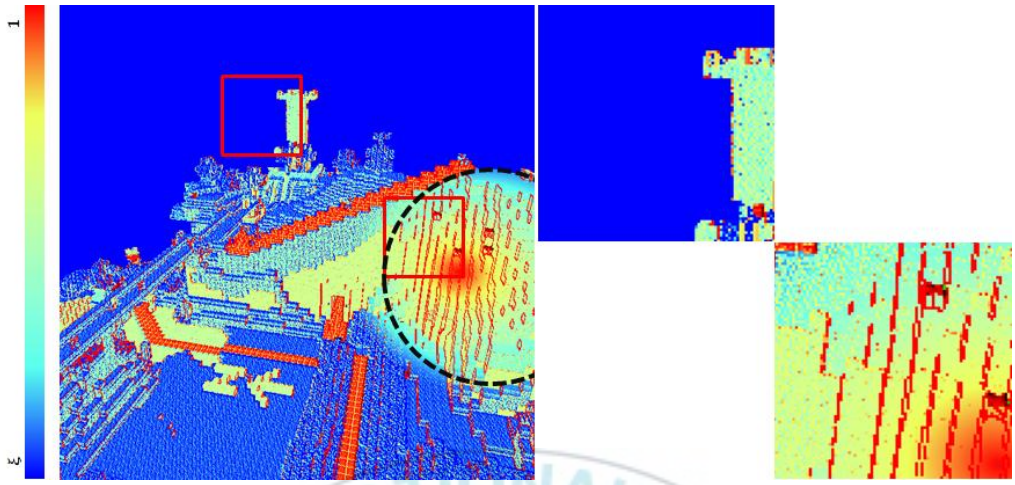

그림 18. 샘플링 맵의 결과

2.2. 리프로젝션 맵

파이프라인의 결과는 희소한 이미지를 생성하기 때문에 이전 프레임의 정보를 재사용하는 템포럴 리프로젝션 [10]을 적용하여 보강한다. 그림 19는 G-BUFFER에서 생성한 포지션과 이전 프레임의 깊이 버퍼를 사용하여 현재 위치 (t)에서 이전 프레임 (t-1)의 MVP (모델, 뷰 및 프로젝션) 행렬을 사용하여 현재 프레임에 투영될 픽셀 좌표를 계산한다.

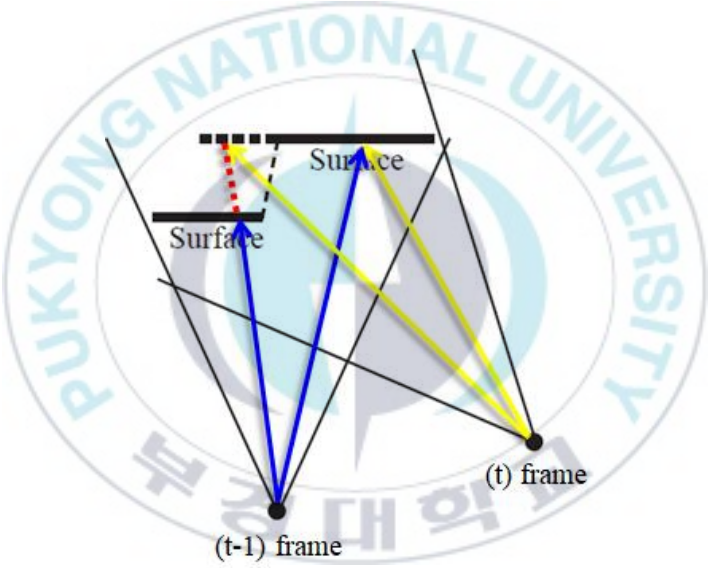

그림 19. 리프로젝션 에러 검사

여기서, 붉은 점선으로 표시된 캐시 미스는 이전 프레임 t-1의 깊이 버퍼와의 비교를 통해 수행된다. 또한, 포비티드 샘플링으로 인해 이전 프레임 버퍼에 캐시에 정보가 없을 수도 있기 때문에 아래 수식 4를 통해 캐시 여부를 계산한다.

$$x = depth(\blacksquare)^{t-1} - |pos(\blacksquare)^{t} - eye^{t-1}| \quad \cdots\cdots\blacktriangleright \quad f(x) = \begin{cases} 1, & x < \varepsilon \rightarrow \text{cache hit} \\ 0, & x \geq \varepsilon \rightarrow \text{cache miss} \end{cases}$$ 수식 4

현재 프레임에서의 포지션 버퍼()와 이전 프레임의 카메라 위치로 이전 프레임에서의 깊이를 계산하고, 이를 이전 프레임의 깊이 버퍼와의 오차로 캐시 여부를 결정한다. ξ는 깊이 버퍼의 최소 변화량으로 최소 변화량보다 작을 경우를 캐시 실패로 처리한다. 결과적으로 그림 20의 오른쪽과 같이 3채널로 구성된 리프로젝션 버퍼 ($F_{Reprojection}$)에 z 채널에 캐시 히트 여부를 저장한다. (z = 0은 캐시 미스, z = 1은 캐시 히트).

그림 20. 캐시 여부 저장된 리프로젝션 버퍼 ($F_{Reprojection}$)

3. GPU 최적화

포비티드 렌더링에서는 샘플링을 통해 계산시간을 향상시킨다. 이러한 장점으로 인해 광선추적법을 실시간으로 수행하는 방법에 대해서 서술한다. 포비티드 샘플링은 최대 80% 이상 감소된 샘플링 맵을 만들 수 있다. 그러나 샘플링 맵은 시선의 위치 및 시각적 주의를 집중시키는 특징정보의 따라 분포되어 있다. 따라서 특징정보가 분산된 샘플링 맵으로 광선추적법을 수행하는 것은 GPU의 쓰레드 실행단위가 고려되지 않아 드라마틱한 성능향상을 기대할 수 없다. 따라서 GPU 최적화 단계를 통해 더욱 효율적으로 동작할 수 있도록 샘플링 맵을 재정렬 한

다.

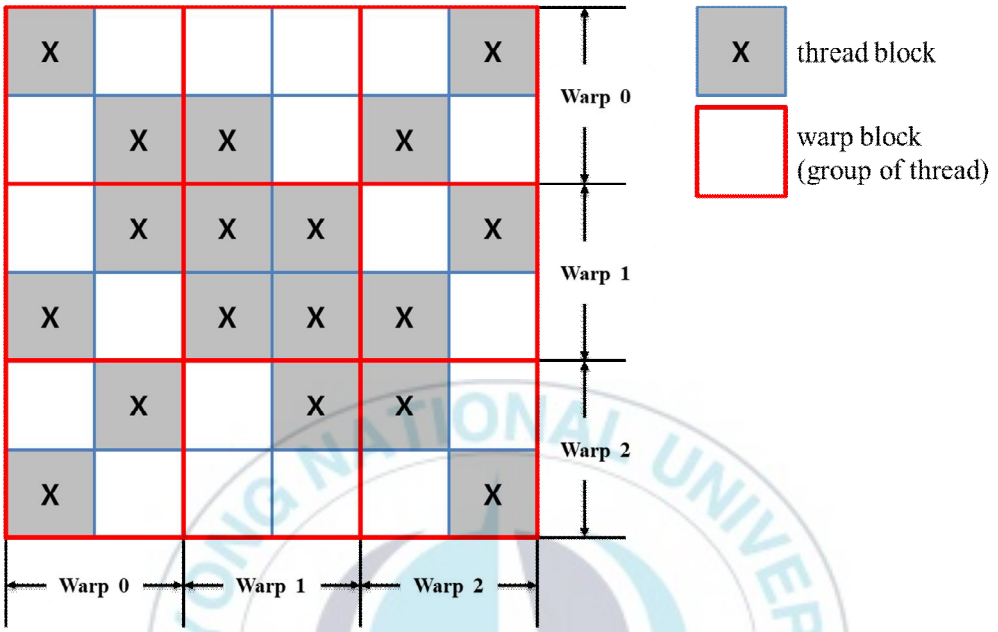

그림 21. 샘플링 맵의 GPU 작업 맵핑

그림 21는 가운데 위치에 사용자의 시선이 추적된 결과를 가정하고 생성된 샘플링 맵을 GPU에 맵핑한 그림이다. 광선추적법을 수행할 픽셀은 모두 18개로 총 36픽셀로 구성된 해상도에서 50%로 샘플링 된 결과이다. 그러나 위의 예시에서는 불행히도 모든 픽셀에서 광선추적법을 수행한 결과와 성능차이가 없다. 따라서 GPU의 멀티코어를 효과적으로 활용하기 위해서는 동일한 스레드를 그룹화하여 실행하는 것이다. 먼저 샘플링 맵은 광선추적법 또는 템포럴 리프로젝션의 두 가지 작업을 수행하는 것으로 정의한다. 여기서 템포럴 리프로젝션의 작업은 단순히 히스토리 버퍼에서 이전 픽셀 값을 캐싱하여 재사용하므로 광선추적법보다 작업 시간이 현저히 짧기 때문에 여기서는 광선추적 작업만 고려한다. (이때 광선 추적법의 작업 시간은 모두 같다고 가정, 샘플링

맵을 희소 행렬 S로 정의)

제안한 전략은 다음과 같이 2가지로 구성하며, 작업 수행을 위해 동일한 크기의 핑퐁 버퍼 R을 준비한다. 먼저 전략 1의 경우 단순히 Compressed Sparse Row(CSR)의 형태로 압축하는 것이다.

전략 1. 입력된 희소 행렬 S에서 각 행의 첫 번째 픽셀에 대해
각 행에서 열의 개수만큼 반복

• S.z = 1이면 핑퐁 버퍼 R의 왼쪽에 저장한다.

→ R (u : 0, v : v).z를 1 증가.

• S.z = 0 인 경우 재정렬 버퍼 R의 오른쪽에 저장한다.

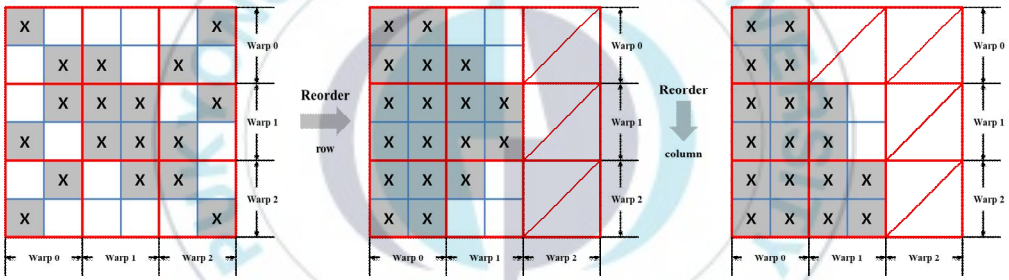

그림 22. 전략 1에 따른 재정렬 절차

두 번째의 경우는 GPU의 쓰레드 그룹 수행단위인 워프단위로 묶어서 처리하는 것이다.

전략 2. 입력된 압축된 희소 행렬 R에서 블록의 각 행의 첫 번째 픽셀에 대해

• 블록의 모든 행에서 첫 번째 픽셀의 R.z에서 최대/최소를 찾는다

→ min과 max의 R.z = 1인 마지막 픽셀을 교환합니다.

• 가장 작은 R의 첫 번째 픽셀 z를 1 증감.

• min과 max의 차이가 1일 때까지 반복수행.

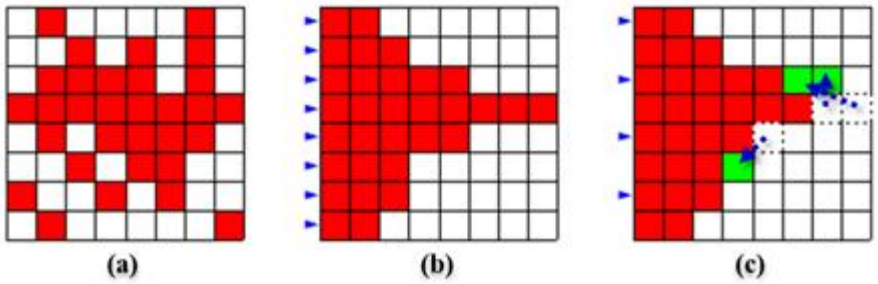

그림 23. 재정렬 수행결과. (a) 입력 행렬, (b) 전략 1을 통한 압축, (c) 전략 2를 통해 (b)를 추가. 압축

샘플링 맵에서 광선추적을 위한 작업은 빨간색 사각형 (S.z = 1)으로 정의하고, 템포럴 리프로젝션은 붉은색과 흰색 사각형 모두에서 수행된다. 그림 21와 같이, 정렬되지 않은 그림 23의 (a) 희소 행렬을 전략 1을 통해 그림 23의 (b)와 같이 CSR 형태로 재정렬한다. 각 행은 GPU에서 독립적으로 수행되며 희소 행렬의 행의 수 만큼 재정렬 작업을 수행하여 압축한다. 그런 다음 전략 2를 통해 압축된 CSR을 블록 높이를 고려한 추가 압축을 통해 최적화 성능을 향상시킨다. 그림 24는 재정렬에 의한 작업 실행결과를 비교한 그림이다. 글로벌 동기화를 위한 대기시간이 감소 되어있음을 확인할 수 있다. 제안한 방법에 의한 성능 평가는 5장에서 논의한다.

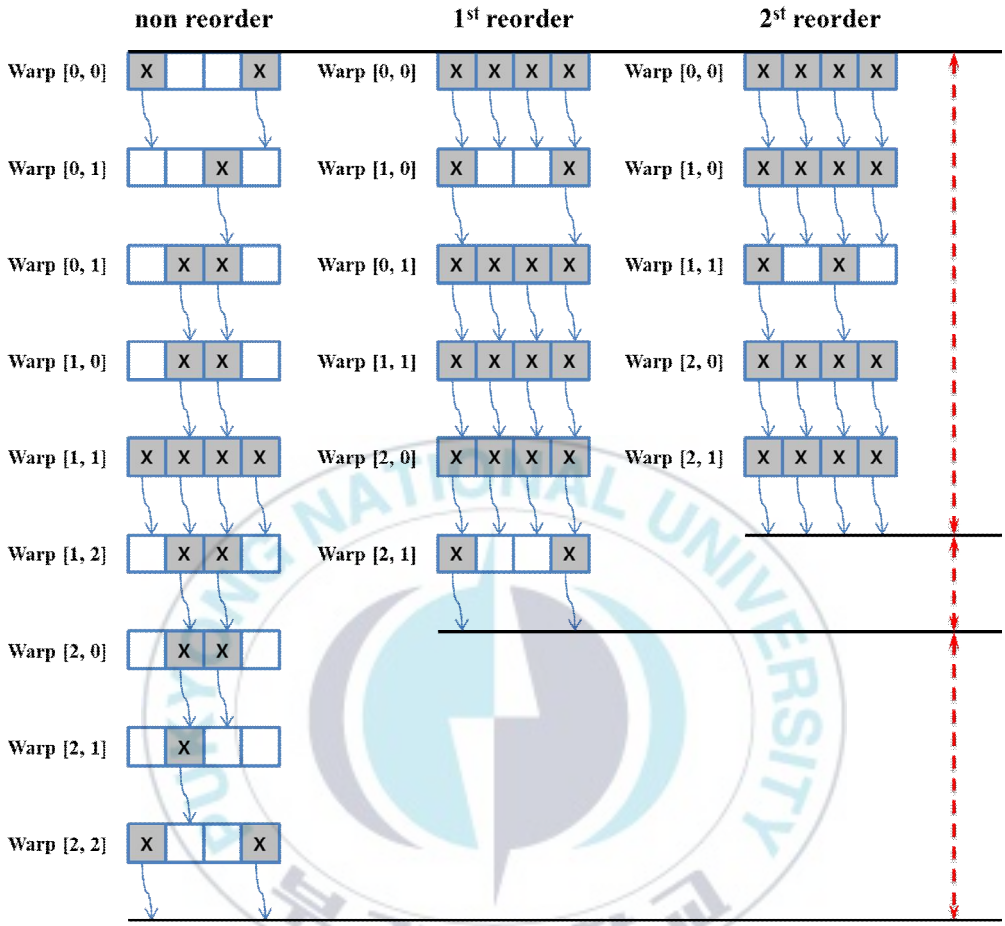

그림 24. 재정렬에 의한 쓰레드 실행 결과

4. 셰이딩

셰이딩 단계는 광선 추적법과 템포럴 리프로젝션을 통해 각 픽셀의 음영 값을 계산한다. 샘플링 맵을 바탕으로 광선 추적법을 통해 해당 픽셀의 색상 값을 획득하고, 광선 추적을 수행하지 않는 픽셀은 템포럴 리프로젝션을 통해 히스토리 버퍼에 저장되어있는 색상 값을 재사용하여 광선 추적법만을 사용하는 것보다 보강된 결과물을 생성하도록 한다. 먼저 작업을 수행하기 전에 프래그먼트에서 사용할 픽셀의 좌표는 이전의 GPU 최적화 단계를 지나면서 맵핑이 되지 않는다. 따라서 FragCoord를 사용하는 대신 재정렬된 샘플링 맵에서 픽셀위치가 저장된 두 x, y 채널을 사용한다. 그리고 샘플링 여부가 저장된 3번째 z 채널의 값으로 광선 추적법과 템포럴 리프로젝션으로 구분되어 수행된다.

4.1. Ray Trace

샘플링 맵을 통해 광선 추적을 수행하는 절차는 그림 25와 같다.

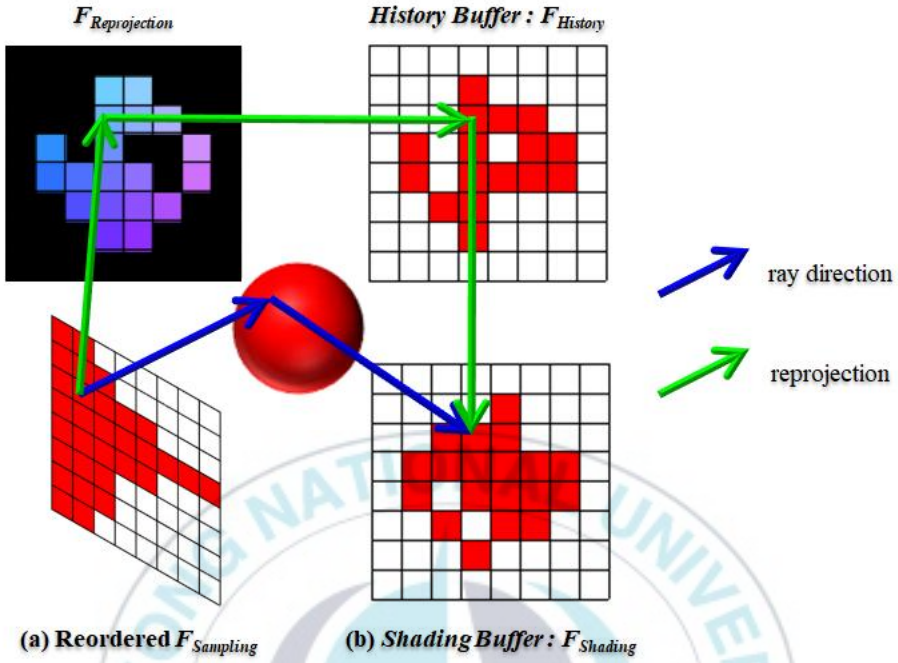

그림 25. 재정렬된 샘플링 맵으로 광선추적 작업수행 절차

재정렬된 샘플링 맵은 프래그먼트의 좌표와 샘플링 맵의 2채널(x,y)에 저장된 좌표가 서로 일치하지 않는다. 셰이딩 프로그램에서의 프래그먼트 좌표는 샘플링 맵의 2채널을 읽어오는데 사용한다. 따라서 프래그먼트 좌표(uv)로 샘플링 맵의 2채널 (x, y)값을 캐싱하여 광선을 생성한 후 광선추적을 수행하고 셰이딩 버퍼의 프래그먼트 좌표(x, y)에 저장한다. 광선추적의 수행 절차를 기술하면 다음과 같다.

- 프래그먼트 좌표 (u, v)로 재정렬된 샘플링 버퍼 (x, y, z)를 읽어온다.
- z = 1 인 경우 x, y의 방향으로 광선을 생성한다.
- 광선 추적 수행.
- 리프로젝션 버퍼의 (x, y)위치를 캐싱하여 캐시 히트 여부를 검사한

후 히스토리 버퍼의 값을 가져온다.

- 셰이딩 버퍼 (x, y)위치에 광선 추적과 히스토리 버퍼의 값을 더하여 저장한다.

4.2. Temporal Reprojection

앞서 수행한 광선 추적의 결과로 채워진 셰이딩 버퍼는 샘플링 확률에 따라 희소한 이미지를 생성한다. 따라서 우리는 재구성 절차를 수행하기 전에 샘플링을 수행하지 않는 픽셀에 대해서도 템포럴 리프로젝션을 통해 이전 프레임 버퍼를 재활용 하는 방법으로 셰이딩 버퍼를 보완한다. 템포럴 리프로젝션의 수행 절차는 다음 그림 26와 같다.

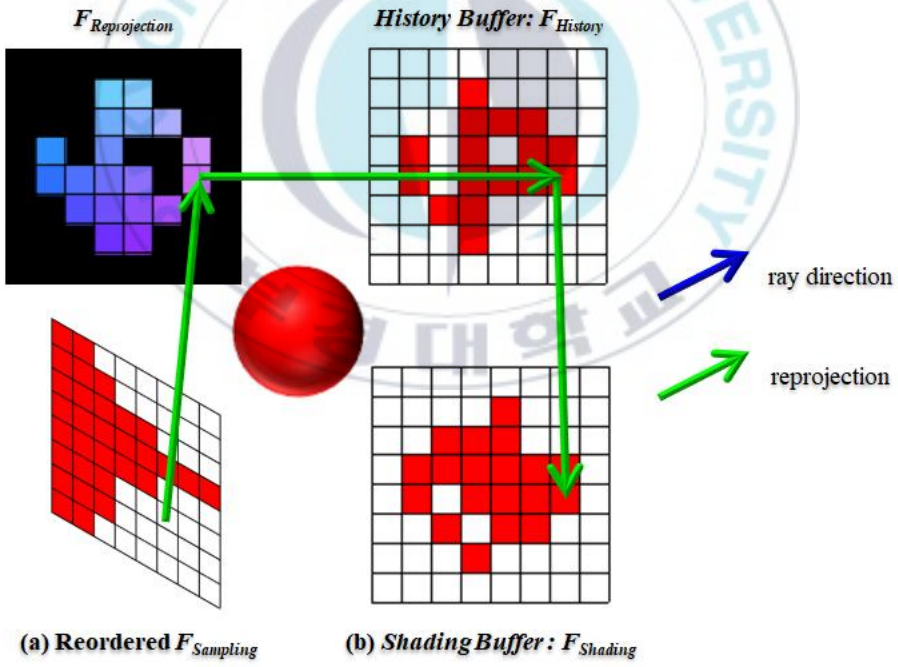

그림 26. 재정렬된 샘플링 맵으로 템포럴 리프로젝션 작업수행 절차

템포럴 리프로젝션 역시 재정렬 샘플링 맵을 기반으로 수행되며,

리프로젝션 좌표 맵과 히스토리 버퍼가 추가로 사용된다. 템포럴 리프로젝션을 수행하는 절차는 다음과 같다.

- 프래그먼트 좌표 (u, v)로 재정렬된 샘플링 버퍼 (x, y, z)를 읽어온다.
- z = 0인 경우 리프로젝션 맵 ($F_{Reprojection}$)의 x, y위치를 캐싱하여 리프로젝션 맵의 z채널에 저장된 캐시 히트 여부를 검사한 후 히스토리 버퍼($F_{History}$)의 값을 캐싱한다.
- 셰이딩 버퍼의 (x, y)위치에 히스토리 버퍼의 값을 저장한다.

위와 같은 절차를 통해, 광선 추적의 셰이딩 버퍼를 보완하여 보다 덜 희소한 결과를 만들 수 있다. 뿐만 아니라 낮은 spp의 광선추적에서도 점진적으로 이미지 품질이 향상되어 결과에 수렴할 수 있다.

5. Reconstruct

이전 단계에서 광선 추적법 뿐만 아니라 템포럴 리프로젝션을 통해 희소한 프레임 버퍼를 보강했음에도 불구하고 결과 이미지는 채워지지 못한 영역이 존재하는 희소 이미지이다. 따라서 최종적으로 디스플레이하기 위해서는 빈 영역을 채우는 절차가 필요하다.

5.1. 보로노이 구조화

포비티드 렌더링은 망막의 광수용체 세포의 밀도를 모방하여 적응적인 해상도로 표현하는 방식으로 동작한다. 이 광수용체 세포는 정형화 되지 않은 다각형 형태의 모양을 나타나며, 이는 보로노이 구조와 유사하다 [18]. 따라서 제안한 절차는 컬러 정보를 획득한 픽셀을 시드

로 하여 빈 영역을 보로노이 구조로 채운다. 셰이딩 버퍼에 채워진 데이터의 분포는 희소하지만 최소 샘플링 확률 ξ에 따라 최소 개수를 보장하였다. 따라서 시드 수와는 독립적이면서도 실시간으로 동작하는 알고리즘인 Rong(2006)의 Jump Flooding Algorithm (JFA) [19]을 사용하여 희소한 셰이딩 버퍼로 보로노이 구조화된 이미지를 생성한다. JFA는 시드 수에 관계없이 항상 동일한 성능으로 보로노이 다이어그램을 생성하며 모든 래스터 픽실에서 $O(1)$안에 가장 가까운 사이트를 찾을 수 있기 때문에 kd-트리를 구성하는 방법보다 GPU에서는 매우 효과적이다. 다음 그림 27은 좌측 상단의 희소 이미지로부터 JFA를 통해 우측 하단의 빽빽한 이미지를 만드는 과정을 나타낸다.

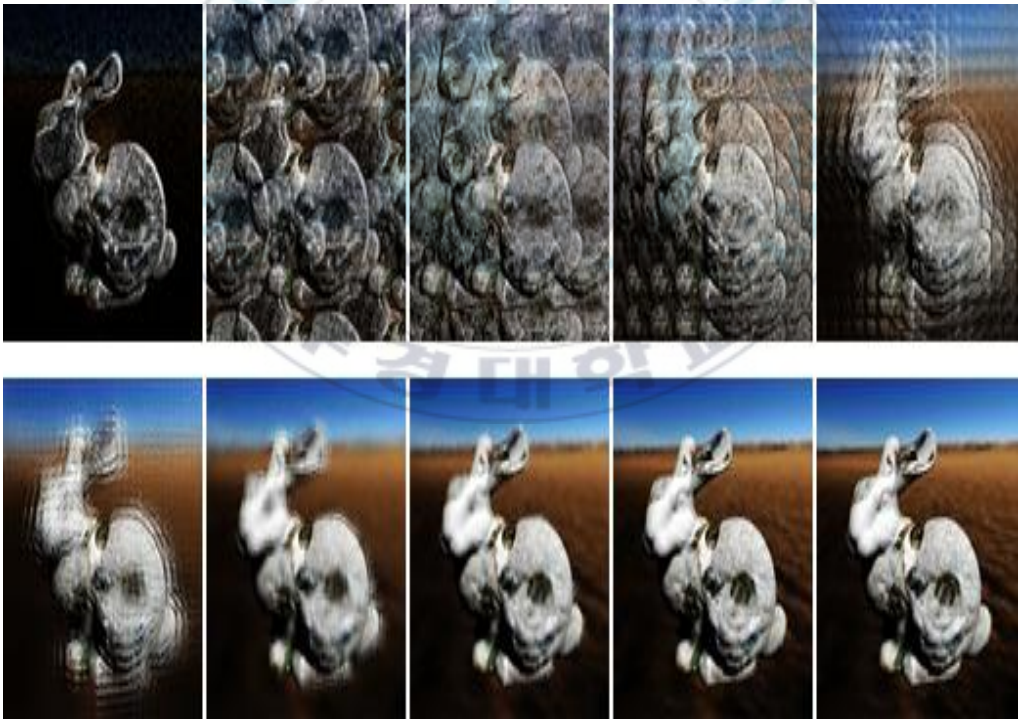

그림 27. JFA의 적용 결과

5.2. 보로노이 경계 보간

JFA를 적용하여 생성한 결과는 보로노이 형태의 특성상 엄격한 경계로 인한 아티팩트가 있다. 또한, 재구성 프로세스에서 채워지는 픽셀은 인접한 시드와 같은 값이므로 엄밀하게는 해당 픽셀의 광선 추적 값이 아니다. 따라서 보로노이 경계를 부드럽게 처리하기 위해 Sibson이 제안한 자연 이웃 보간법(Natural Neighbor Interpolation) [20]을 수행한다. 자연 이웃 보간법은 보로노이 구조에서의 이상적인 보간 방법이다. 그러나 Sibson 보간법은 비실시간으로 동작하기 때문에 GPU에서 동작하는 Park(2006)의 Discrete Sibson Interpolation을 적용하였다 [21]. 그러나 Park이 제안한 보간법은 처리 대상의 해상도가 높을수록 알고리즘의 동작원리에 의해 실시간이 보장되지 않는다. 따라서 다음과 같은 절차를 통해 실시간으로 동작하도록 하였다. 절차는 다음과 같다.

- 입력으로 JFA의 핑퐁 버퍼 (T1 : 컬러 버퍼, T2 : 좌표 버퍼)
- 모든 출력 래스터 위치에 대해
  - 》 T1버퍼에서 동일한 레스터 위치 p에 존재하는 컬러 정보 c1를 찾는다.
  - 》 T2버퍼에서 동일한 래스터 위치 p에 저장된 시드 정보 s1을 찾는다.
  - 》 최소 jitter size를 통해 bbox를 설정
  - 》 bbox범위 내에서 반복적으로 이동하며 이웃 시드 정보 s2를 찾는다.
  - 》 탐색한 이웃 시드 s2와 검색 래스터 위치 q와의 거리가 래스터 위치 p와 bbox내의 래스터 위치 q의 거리보다 크면, 이웃 시드

s2의 컬러 정보를 합산

- 모든 출력 레스터 위치 p에 대해서 합산한 결과를 정규화 하여 출력한다.

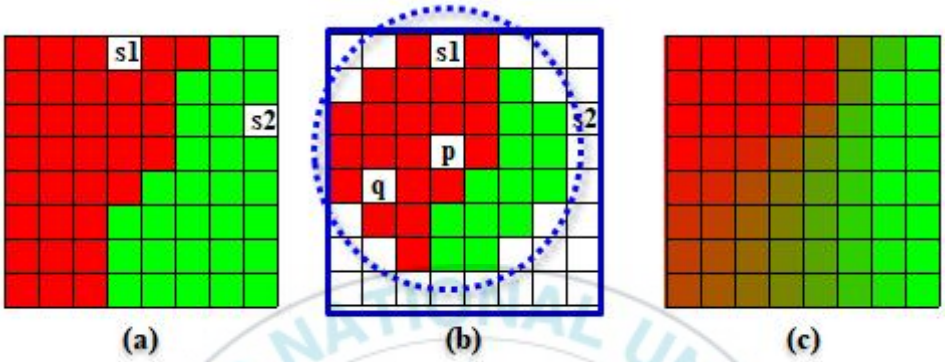

그림 28. 래스터 위치 p에서 이웃 시드 s2를 검색하고 보간하는 절차. (a) JFA의 T1버퍼와 T2버퍼 (T1은 컬러정보, T2는 좌표정보 동일한 컬러에는 동일한 좌표가 저장되어있음), (b) 검색 및 보간 과정, (c) 인접 보간의 결과

제안한 방법은 탐색 윈도우의 크기가 결정되기 때문에 Park의 Discrete Sibson Interpolation보다 빠르게 수행된다. 탐색 윈도우의 크기를 결정하는 jitter block size는 최소 샘플링 확률을 결정하기 때문에 보간의 계산시간과 광선추적법의 계산시간 사이의 트레이드 오프를 고려하여 결정하여야 한다.

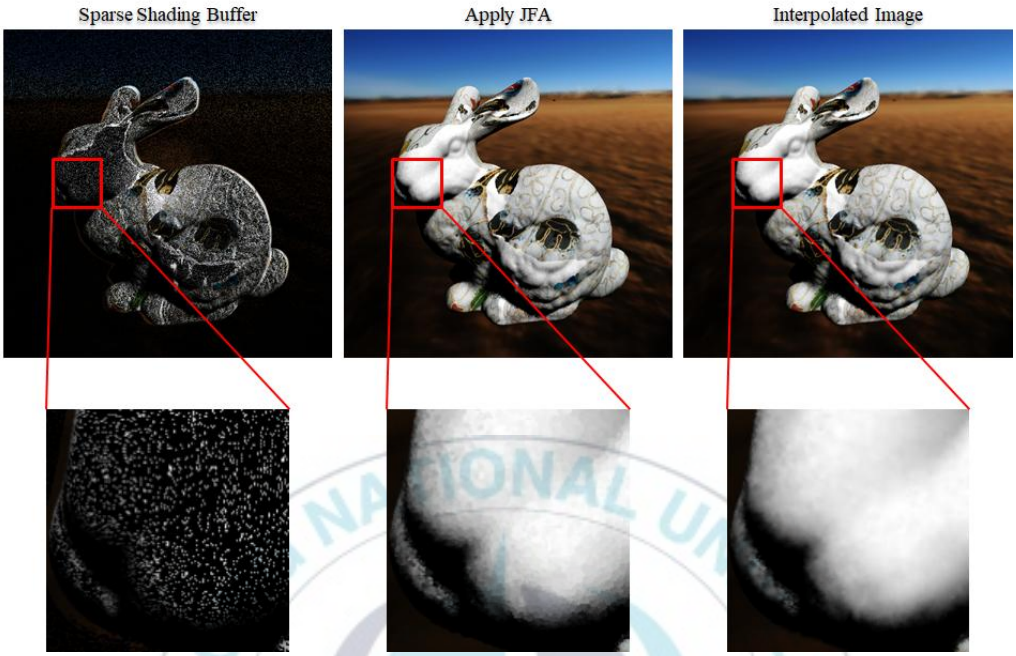

그림 29. 보로노이 구조화 및 보간 결과

그림 29는 보로노이 구조화 및 자연 이웃 보간의 결과를 나타낸다. 제안한 재구성 절차를 통한 희소 셰이딩 버퍼로부터 빽빽하게 채워진 결과를 확인할 수 있다.

IV Implementation

이 챕터에서는 실제 구현을 위한 렌더링 파이프라인을 기술한다. 시선추적 장비로부터 스크린 좌표상의 시선(gaze)의 위치를 입력받는 것을 전제로 설계하였다. 듀얼 디스플레이의 경우 각각 해당되는 눈에 영상이 투사되므로 해당 렌더링의 소요시간의 최소화를 위해 몇 가지의 절차가 추가된다.

1. 개요

먼저 구현을 위한 포비티드 렌더링 파이프라인은 다음 그림 30과 같다.

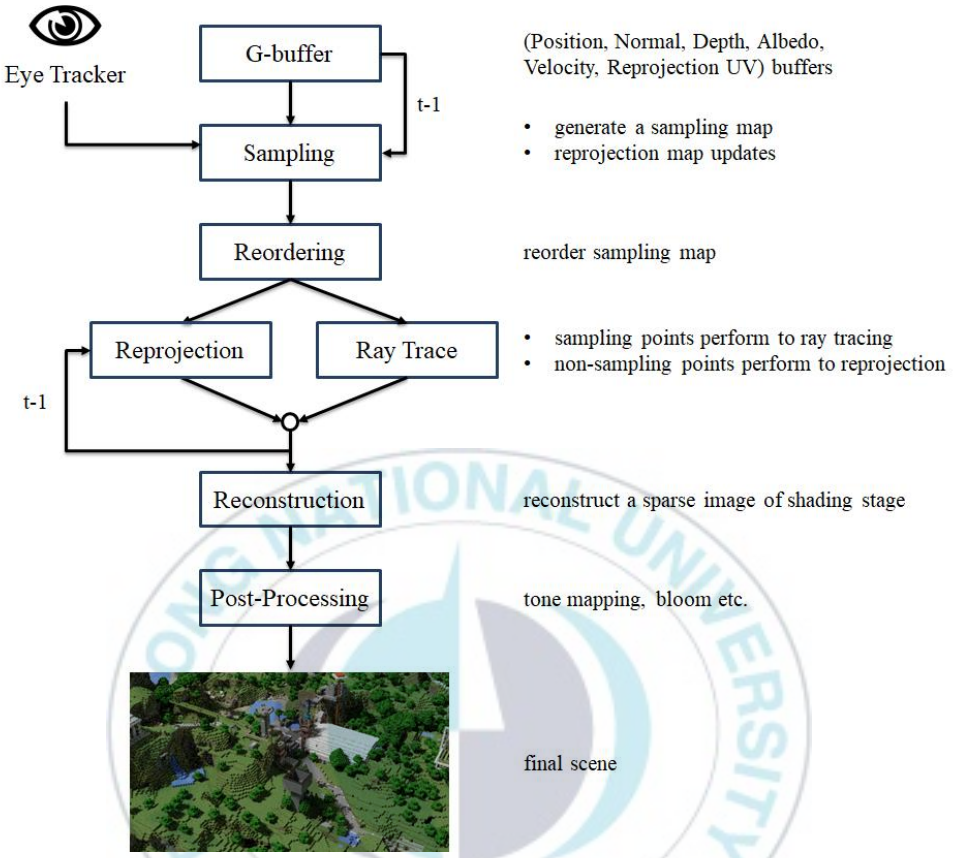

그림 30. 포비티드 렌더링 파이프라인

먼저 G-buffer stage에서는 Position, Normal, Depth, Albedo, Velocity, Reprojection UV 버퍼를 생성한다.

2. Sampling Stage

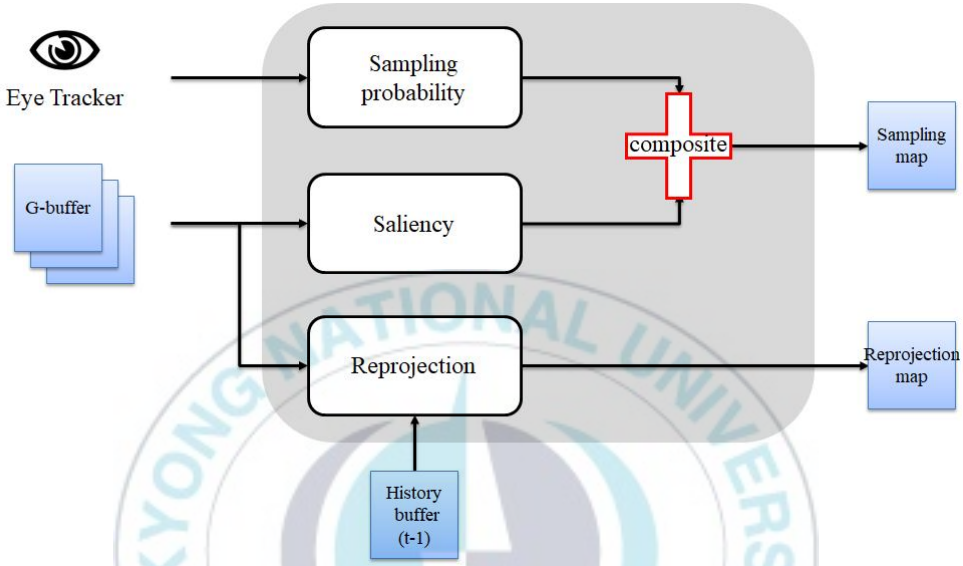

그림 31. 샘플링 단계의 절차

샘플링 단계에서는 G-buffer와 시선의 위치와 히스토리 버퍼를 사용해 sampling map과 reprojection map을 생성한다. 우선 sampling map은 해당 픽셀의 값을 구하기 위해 광선추적법과 템포럴 리프로젝션의 작업에 대한 정보를 저장하는 3채널 버퍼로 구성하며, (x, y) 채널에는 픽셀의 좌표와, z 채널에는 샘플링 확률을 저장한다.

reprojection map은 재투영을 통해 히스토리 버퍼에 저장되어있는 픽셀정보를 재사용하여, 포비티드 샘플링으로 인한 희소한 픽셀 정보를 보완한다. 3채널 버퍼로 정의하며 (x, y) 채널에는 재투영이 수행될 t-1의 재투영 좌표가 저장되고 z에는 재투영 오류검사를 통한 캐시 여부를 저장한다. 캐시히트 검사를 수행하기 위해서는 이전 프레임의 깊

이 버퍼정보가 필요하므로, 깊이 버퍼의 초기화는 수행하지 않는다. 샘플링 단계는 셰이더 프로그램으로 동작하며, 그 코드는 다음과 같다.

Code 1: GLSL **SAMPLING** fragment shader

Input: G-buffers
output: outFovSample, outTemporalReproj

```
layout(binding = 0) uniform sampler2D positionTex;
layout(binding = 1) uniform sampler2D normalTex;
layout(binding = 2) uniform sampler2D albedoTex;
layout(binding = 3) uniform sampler2D depthTex;
layout(binding = 4) uniform sampler2D prev_depthTex;
layout(binding = 5) uniform sampler2D reprojectionTex;
layout(binding = 6) uniform sampler2D velocityTex;

layout(location = 0) out vec4 outFovSample;
layout(location = 1) out vec4 outTemporalReproj;

layout(location = 0) uniform vec2 uf_screenSize;
layout(location = 1) uniform float uf_depth_epsilon;
layout(location = 2) uniform vec2 uf_gaze;
layout(location = 3) uniform vec4 uf_eye;
layout(location = 4) uniform vec4 uf_prev_eye;
layout(location = 5) uniform float uf_mar_slope;
layout(location = 6) uniform float uf_mar_min;
layout(location = 7) uniform float uf_mar_far;

void main()
{
    vec2 FragCoord = gl_FragCoord.st / uf_screenSize;

    vec4 pos = texture2D(positionTex, FragCoord);
    vec4 normal = texture2D(normalTex, FragCoord);
    float depth = texture2D(depthTex, FragCoord).x;
    vec2 reprojection_coord = texture2D(reprojectionTex, FragCoord).xy;
```

```
    vec4 velocity = texture2D(velocityTex, FragCoord);

    float prev_depth = texture2D(prev_depthTex, reprojection_coord).x;
    float hit = func_isCacheHit(prev_depth, length(pos - uf_prev_eye), uf_depth_epsilon);

    float e = func_eccentricity(uf_gaze, uf_eye, FragCoord);
    float fov_sample_rate = func_fov_probability(e, uf_mar_slope, uf_mar_min,
uf_mar_far);

    float saliency = func_saliency(albedoTex, FragCoord);
    float normal_gradient = func_gradient(normalTex, FragCoord);
    float DoF = func_depth_of_field(depthTex, FragCoord, uf_gaze);
    float usingRay = func_sampling_probability(fov_sample_rate, saliency, normal_gradient,
DoF);

    outFovSample = vec3(FragCoord, usingRay);
    outTemporalReproj = vec3(reprojection_coord, hit);
}
```

3. Reordering Stage

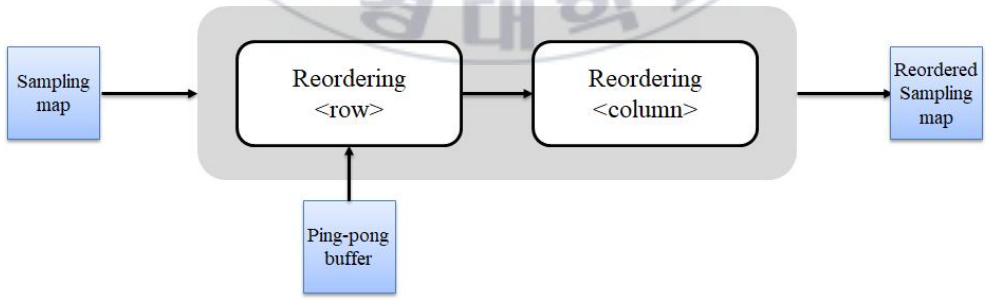

그림 32. Reordering 단계를 통한 GPU 작업을 위한 최적화 절차

이 단계에서는 이전 단계에서 구한 샘플링 맵에서 수행될 광선 추적법이 GPU에서 효율적으로 동작할 수 있도록 재정렬을 수행하며 그

림 32의 절차를 통해 2단계로 수행된다. 재정렬을 수행하기 위해서 샘플링 맵과 동일한 사이즈의 백버퍼인 ping-pong buffer가 필요하며, 버퍼의 사이즈가 너무 클 경우 재정렬 수행의 성능을 고려해 분할 처리할 수 있도록 segment 정보가 필요하다. 이 segment를 바탕으로 block단위로 분할 압축을 수행하여 재정렬을 수행하게 된다. 또한 재정렬 과정에서 읽는 영역과 쓰는 영역이 다르므로, 여기서는 컴퓨트 셰이더(compute shader)를 사용하여 처리하며 해당 셰이더 프로그램의 코드는 다음과 같다.

Code 2: GLSL **REORDER ROW COMPRESS** compute shader

Input: samplingMap, bufferSize
output: dstMap

```
layout(binding = 0) uniform sampler2D samplingMap;
layout(location = 0) uniform vec2 bufferSize;
layout(binding = 0, rgba32f) coherent uniform image2D dstMap;

layout(local_size_x = 1, local_size_y = 32) in;
void main()
{
    ivec2 idx_start = ivec2(gl_GlobalInvocationID., gl_GlobalInvocationID.y);
    ivec2 idx_end = ivec2(bufferSize.x - 1, gl_GlobalInvocationID.y);

    for (int i = 0; i < bufferSize.x; i++)
    {
        ivec2 idx_search = ivec2(i, gl_GlobalInvocationID.y);
        vec4 data = texture(samplingMap, idx_search / bufferSize);

        if (data.z > 0)
        {
            imageStore(dstMap, idx_start, data);
```

```
            vec4 first_sample = imageLoad(dstMap, ivec2(0, gl_GlobalInvocationID.y));
            first_sample.z++;
            imageStore(dstMap, ivec2(0, gl_GlobalInvocationID.y), first_sample);
            idx_start.x++;
        }
        else
        {
            imageStore(dstMap, idx_end, data);
            idx_end.x--;
        }
    }
}
```

4. Shading Stage

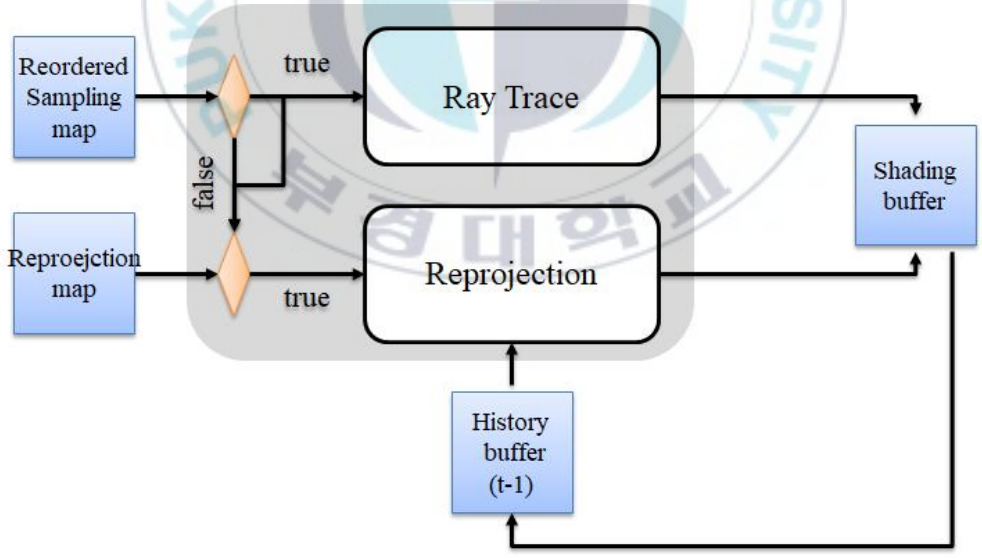

그림 33. 셰이딩 단계의 절차

셰이딩 단계에서는 재정렬된 샘플링 맵과 리프로젝션 맵의 두 텍스

쳐가 입력으로 주어지며 광선추적과 리프로젝션의 2가지의 작업을 수행한다. 재정렬된 샘플링 맵의 z채널에 저장된 값을 바탕으로 그림 33와 같이 광선추적법과 재투영을 작업을 선택하여 수행한다. 이전의 재정렬 단계로 인해 광선의 생성을 위한 좌표정보가 프래그먼트 셰이더의 좌표와는 다르게 뒤섞였으므로, 컴퓨트 셰이더를 사용하여 처리한다. 셰이더 코드는 다음과 같다.

Code 3: GLSL **SHADING** compute shader

Input: samplingMap, reprojectionMap, shadingTex, historyTex
output: outShadingTex, outHistoryTex

```
layout(location = 0) uniform sampler2D samplingMap;
layout(location = 1) uniform sampler2D reprojectionMap;
layout(location = 2) uniform sampler2D shadingTex;
layout(location = 3) uniform sampler2D historyTex;

layout(location = 0) uniform vec2 uf_screenSize;
layout(location = 1) uniform float seed;
layout(location = 2) uniform float coef_temporal;

layout(binding = 0, rgba32f) coherent uniform image2D outShadingTex;
layout(binding = 1, rgba32f) coherent uniform image2D outHistoryTex;

layout(local_size_x = 32, local_size_y = 32) in;
void main()
{
    vec2 FragCoord = gl_FragCoord.st / uf_screenSize;

    vec3 thread_work = texture2D(samplingMap, FragCoord);
    vec2 thread_uv = thread_work.xy;
    vec4 reprojection_uv = texture2D(reprojectionMap, thread_uv);

    vec4 temporal_color = vec4(0.0f);
```

```
    if (c_weight.z > 0.0f) {
        temporal_color = texture2D(historyTex, reprojection_uv);
    }

    bool usingRay = thread_work.z;
    if (!usingRay) {
        float acc_frame = fabs(frame - imageLoad(shading_buffer, ivec2(thread_uv)).w;
        imageStore(outHistoryTex, ivec2(thread_uv),
            acc_frame < coef_temporal ? temporal_color : vec4(0.0f));
        imageStore(outShadingTex, ivec2(thread_uv), acc_frame <
            coef_temporal ? vec4(color_to_accumulated(temporal_color), temporal_color.w *
shading_buffer[thread_uv].w) : make_float4(0.0f);
        return;
    }

    vec2 pixel = vec2(thread_uv) / uf_screenSize * 2.0f - 1.0f;
    vec4 tmp = vec4(pixel, -1.0f, 1.0f);
    tmp = mvp * tmp;
    vec3 nearPos = vec3(tmp) / tmp.w;

    vec3 ray_origin = eye;
    vec3 ray_direction = normalize(nearPos - eye);

    Ray ray(ray_origin, ray_direction);
    trace(model_object, ray);

    vec3 result_color = ray.result;

    result = Uncharted2ToneMapping(result);
    float4 final_result = make_float4(result, 1.0f) + temporal_color;

    imageStore(history_buffer, ivec2(thread_uv), final_result);
    imageStore(shading_buffer, ivec2(thread_uv), vec4(color_to_accumulated(final_result),
frame + 1);
}
```

5. Reconstruct Stage

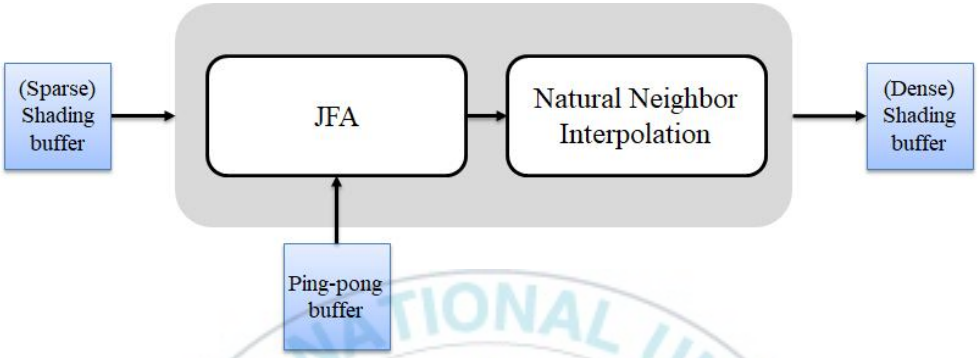

그림 34. 재구성 단계의 절차

재구성 단계에서는 이전 셰이딩 단계에서 구한 희소 셰이딩 버퍼에서 빈 영역을 채우는 역할을 수행한다. 그림 34와 같이 샘플링 맵을 통해 생성된 셰이딩 버퍼를 JFA를 통해 보로노이로 구조화한 후 자연 이웃 보간법을 통해 최종 결과를 출력한다. JFA를 수행하기 위해서는 2개의 핑퐁 버퍼라고 부르는 컬러 정보를 저장하는 버퍼와 시드의 위치 정보를 저장하는 버퍼가 필요하다. 또한 보간의 실시간을 고려해 최소 샘플링 확률을 보장하는 jitter 윈도우의 크기를 넘겨준다. Rong의 JFA의 셰이더 코드는 생략하며, 재구성 단계의 셰이더 코드는 다음과 같다.

Code 4: GLSL **RECONSTRUCT** fragment shader

Input: coordTex, colorTex
output: outColor

```
layout(binding = 0) uniform sampler2D coordTex;
layout(binding = 1) uniform sampler2D colorTex;
```

```
layout(location = 0) out vec4 outColor;

layout(location = 0) uniform vec2 screenSize;
layout(location = 0) uniform float jitterSize;

void main() {
    vec2 FragCoord = gl_FragCoord.st / screenSize;
    vec4 closest = texture2D(coordTex, FragCoord.st);
    vec4 closestColor = texture2D(colorTex, closest.st);

    vec2 min_box = FragCoord - jitterSize;
    vec2 max_box = FragCoord + jitterSize;
    vec2 increment = vec2(1.0f) / screenSize;

    vec4 incrementColor = vec4(0.0f);
        for (float h = min_box.y; h < max_box.y; h += increment.y) {
            for (float w = min_box.x; w < max_box.x; w += increment.x) {
                vec2 ref_uv = vec2(w, h);
                if (ref_uv.s < 0.0 || ref_uv.s >= 1.0f || ref_uv.t < 0.0 || ref_uv.t >= 1.0f)
                    continue;
                closest = texture2D(coordTex, ref_uv.st);
                float radius = distance(closest.st, ref_uv.st);
                float dist_i_to_p = distance(FragCoord.st, ref_uv.st);
                if (radius < dist_i_to_p)
                    continue;

                vec4 ref_closestColor = texture2D(colorTex, ref_uv.st);
                incrementColor += vec4(ref_closestColor.xyz, 1.0f);
            }
        }

        if (incrementColor.a > 0.0f)
            outColor = vec4(incrementColor.rgb / incrementColor.a, 1.0f);
        else
            outColor = closestColor;
}
```

V 평가

파이프라인의 성능 및 결과 이미지의 품질을 평가하기 위해 그림 35에 표시된 6가지의 모델 (상세 스펙: 표1 참조)으로 실험을 수행하였다. 셰이딩 모델은 하나의 그림자 광원으로 설정하였다. 먼저 이미지의 품질에 대한 평가는 전체 해상도로 광선 추적을 수행한 것을 기준으로 평가를 수행하였다. 그리고 성능평가는 전체 해상도로 광선 추적을 수행한 것과 샘플링 맵을 재정렬하지 않은 버전과 재정렬한 버전으로 구성하였다. 사용한 하드웨어 시스템은 Intel Core i5-6600 CPU, 8GB Ram 및 NVIDIA GeForce GTX 970으로 구성하였으며 평가의 정확성을 위해 각 장면당 300번의 반복 수행의 평균을 나타내었다.

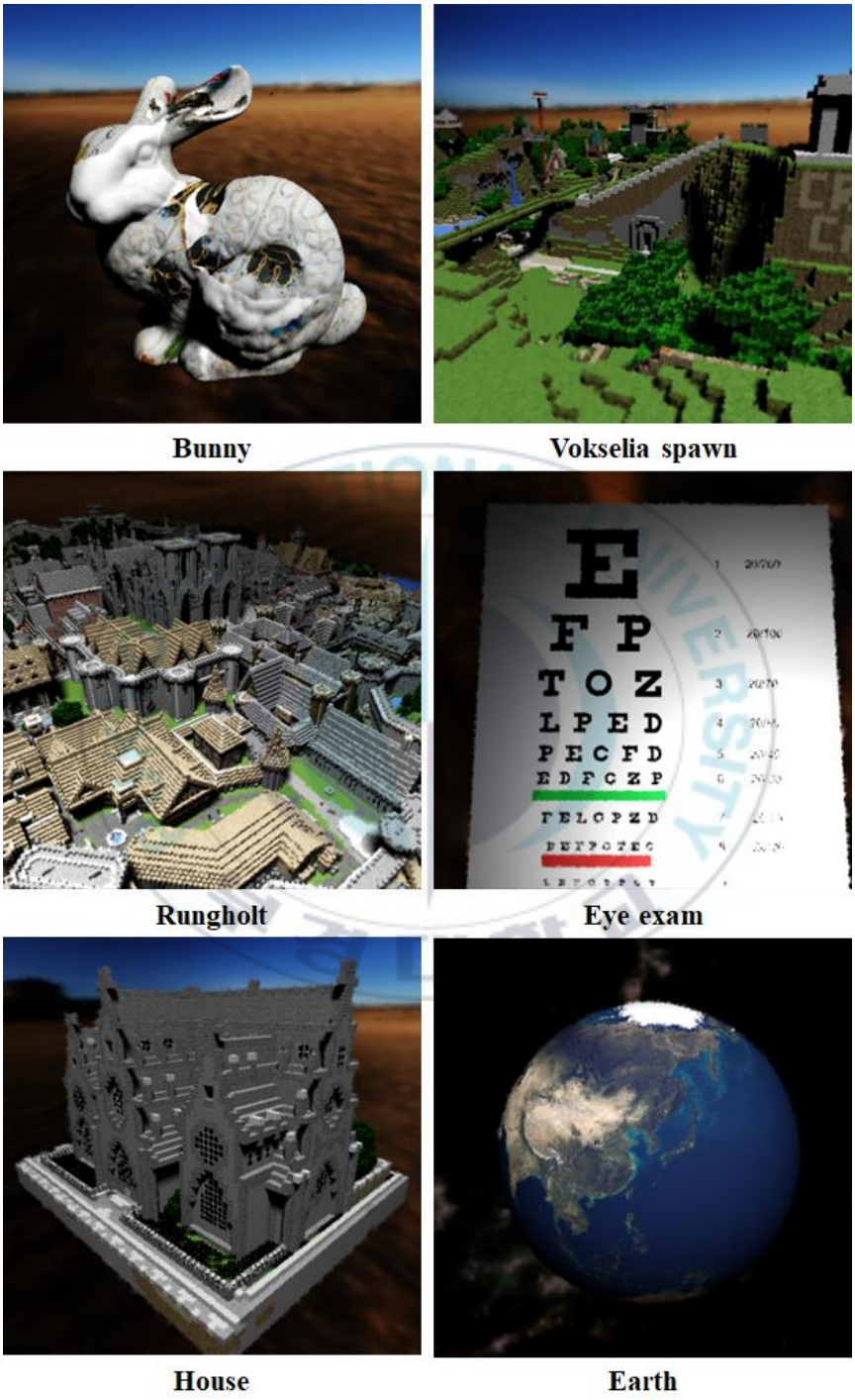

그림 35. 평가에 사용된 모델

Name	Vertex	Triangles
bunny	72,027	144,046
Vokselia spawn	863,521	1,875,632
Rungholt	3,289,722	6,704,264
Eye exam	25	32
House	85,900	126,248
Earth	482	960

표 1. 데이터 셋의 상세스펙

1. 이미지 품질 평가

모든 테스트 장면에 대해 제안한 파이프라인에서 실행된 결과를 보여준다. 품질을 평가하기 위해서 전체 해상도로 광선 추적을 수행한 것과 제안한 방법과의 시각적 품질을 비교하였다. 평가 척도는 피크 신호대 잡음비(PSNR)와 구조적 유사도 (SSIM)로 나타냈으며 그림 36과 37은 각 조건에 대한 품질의 결과를 보여준다.

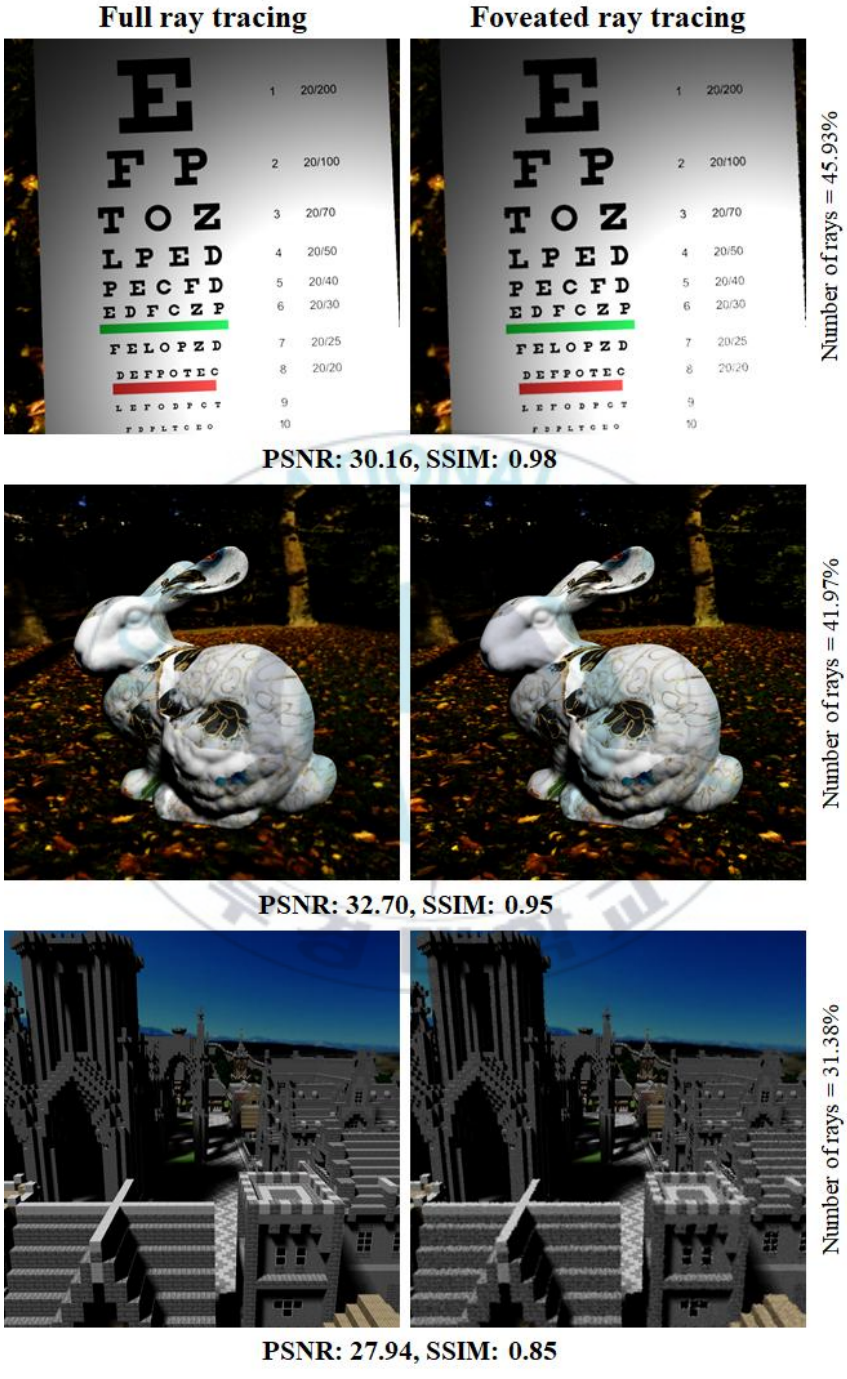

그림 36. 이미지 품질 평가 1(좌: 전체 해상도 광선 추적 후애, 우: 제안한 샘플링으로 인한 광선추적 결과)

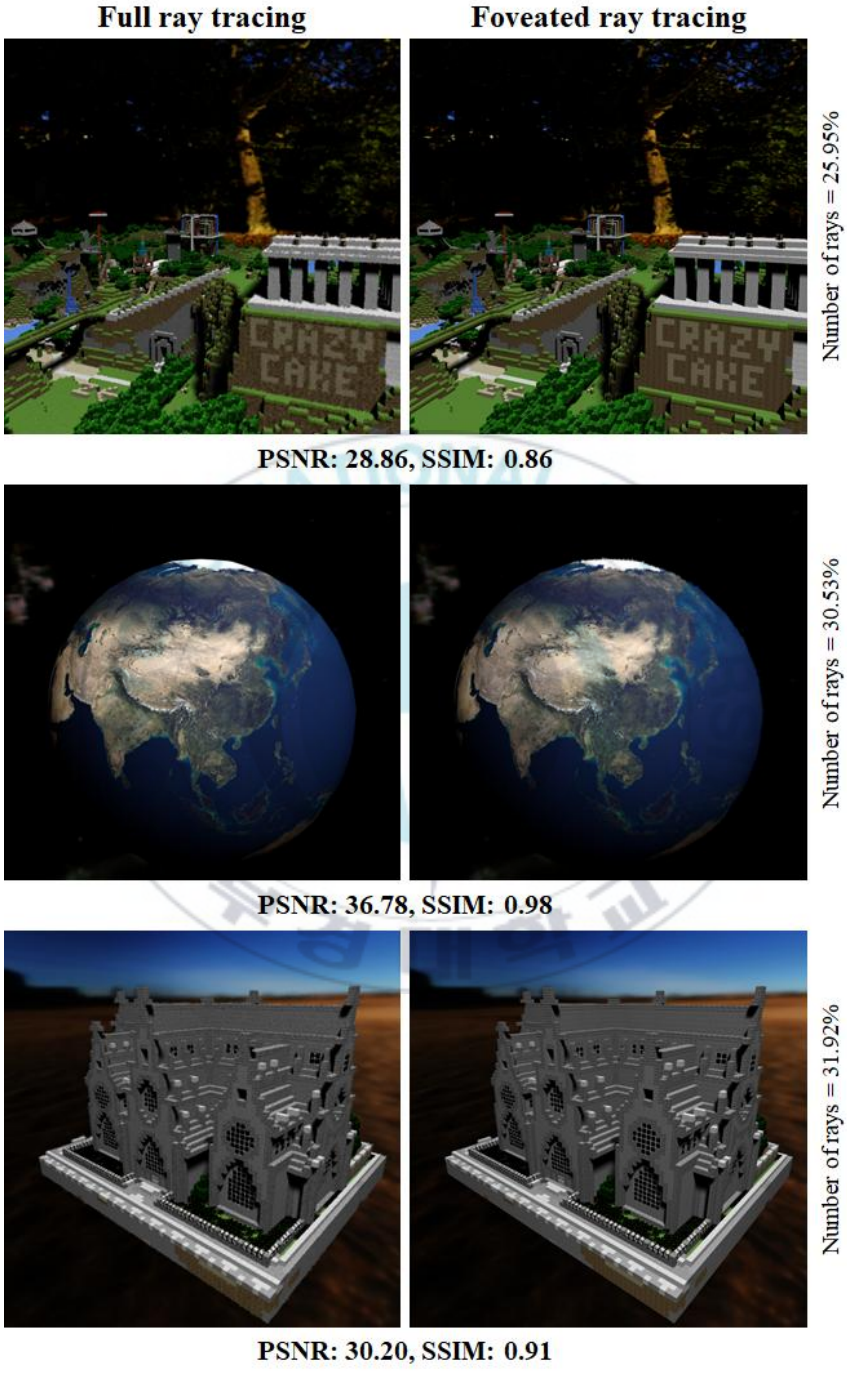

그림 37. 이미지 품질 평가 2(좌: 전체 해상도 광선 추적 후애, 우: 제안한 샘플링으로 인한 광선추적 결과)

2. 성능 평가

그림 38은 각 테스트 장면에 대해 전체 광선 추적을 수행한 것과 GPU 최적화를 수행하지 않은 버전, 그리고 제안한 파이프라인에서 실행된 결과의 성능을 나타낸다. 모든 픽셀에 대해 광선 추적을 수행하는 것에 비해 재정렬을 수행하지 않은 포비티드 렌더링과 제안한 방법 모두 성능향상을 보이나, 재정렬로 인해 더욱 광선 추적에 드는 계산 비용이 감소함을 알 수 있다.

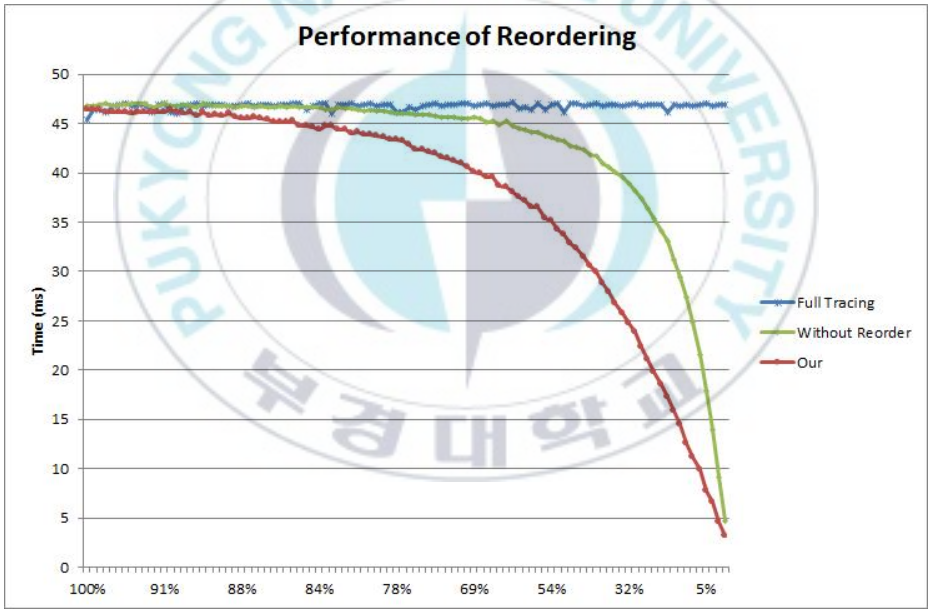

그림 38. 재정렬에 의한 성능평가 그래프

그림 39는 park의 이산 자연 이웃 보간과 제안한 방식의 보간 성능을 나타낸 결과이다. 제안한 방법은 보로노이 시드의 수가 증가할수록 보간을 위한 검색 윈도우의 사이즈가 줄어들기 때문에 동일한 해상도에서 더욱 빠른 성능을 보여준다.

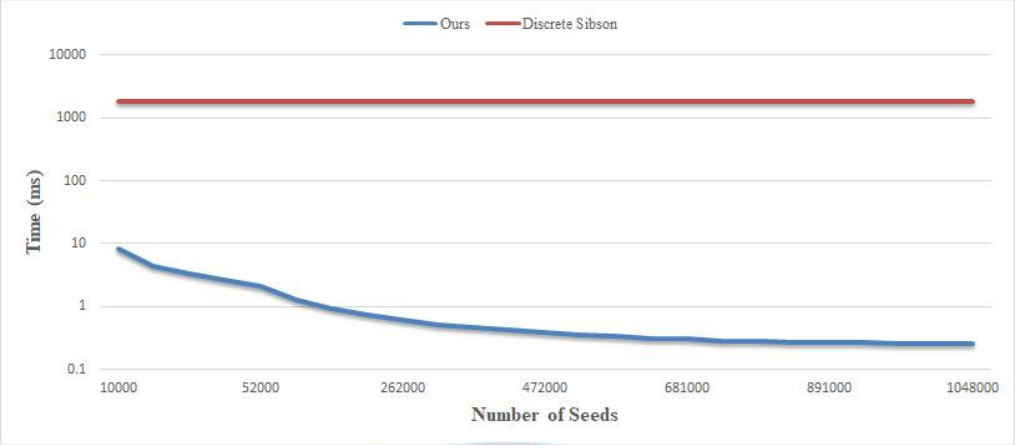

그림 39. 시드 수에 따른 보간 성능 비교 (Y축은 로그 시간)

3. 논의

앞 절의 성능 평가와 품질평가를 바탕으로 개선된 계산 성능과 지각적인 세부사항의 손실률이 크지 않다는 것을 확인하였다. 터널시야 현상은 주변시 영역에서의 과도한 블러효과로 야기되는 것이므로 주변시 영역에서 시각적 주의를 집중시키는 특징에 대해 샘플링 수행하는 방법을 통해 이 문제가 해결 될 것을 기대한다. 또한 제안한 방법은 다른 포비티드 렌더링 기술과 동일한 샘플의 수로도 재정렬의 방법을 통해 약 23%이상의 계산시간이 줄어듬을 확인하였다. 다음 그림 42에서 47까지는 각 장면에 대해 중심시 영역과 주변시 영역의 결과를 확대 비교한 결과이다.

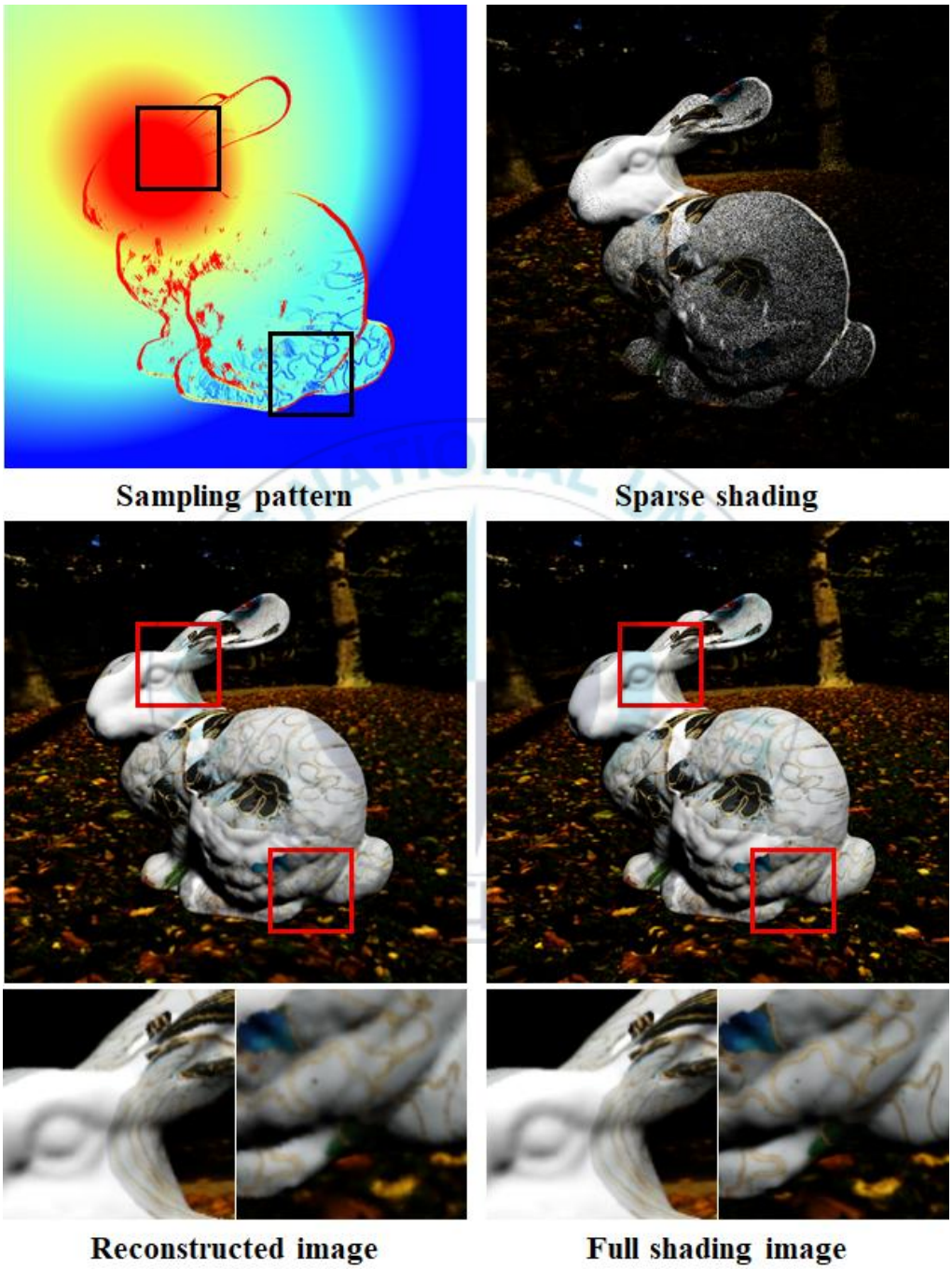

그림 40. 렌더링 결과 1. (sampling rate = 41.97%)

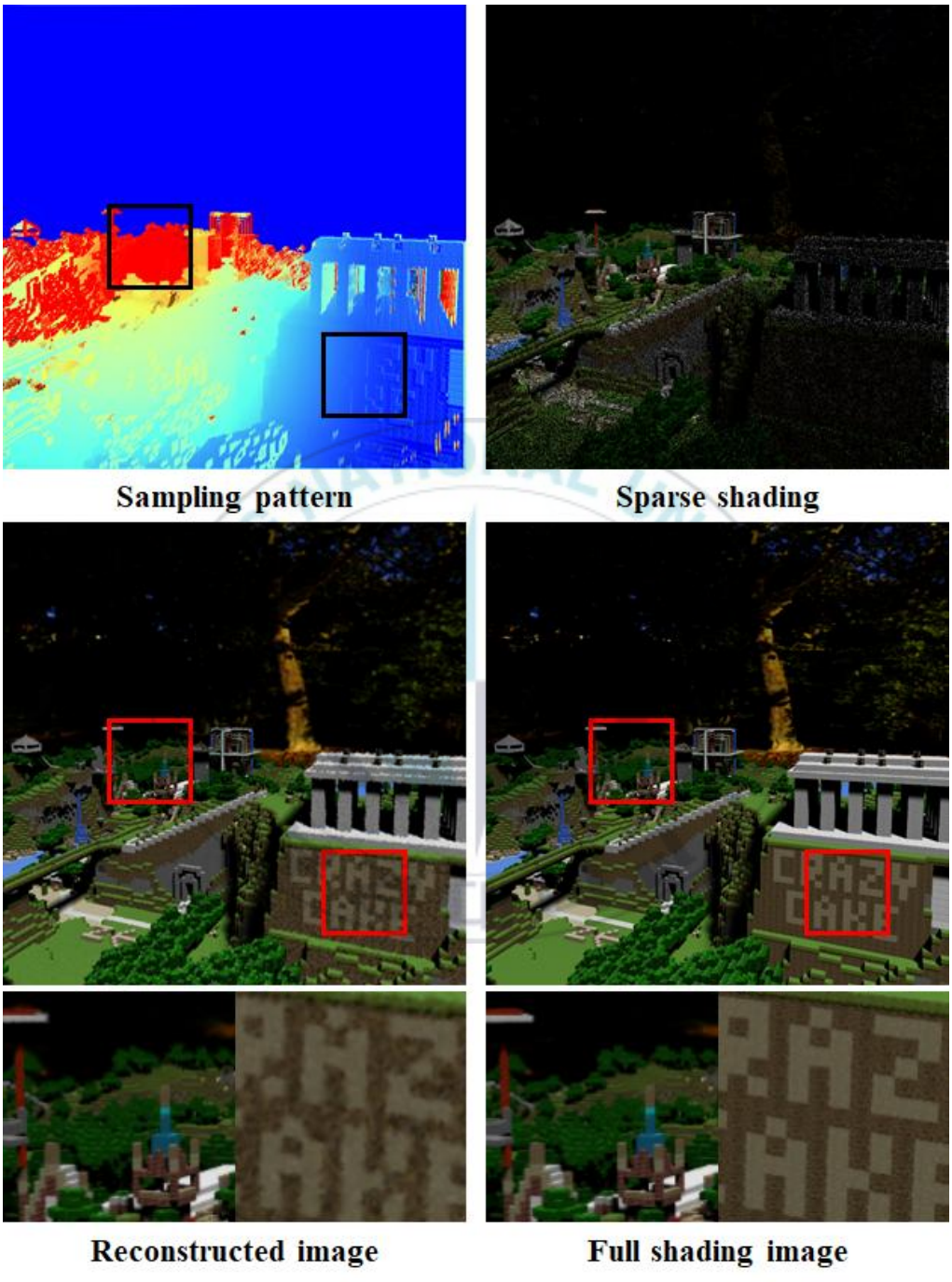

그림 41. 렌더링 결과 2. (sampling rate = 25.95%)

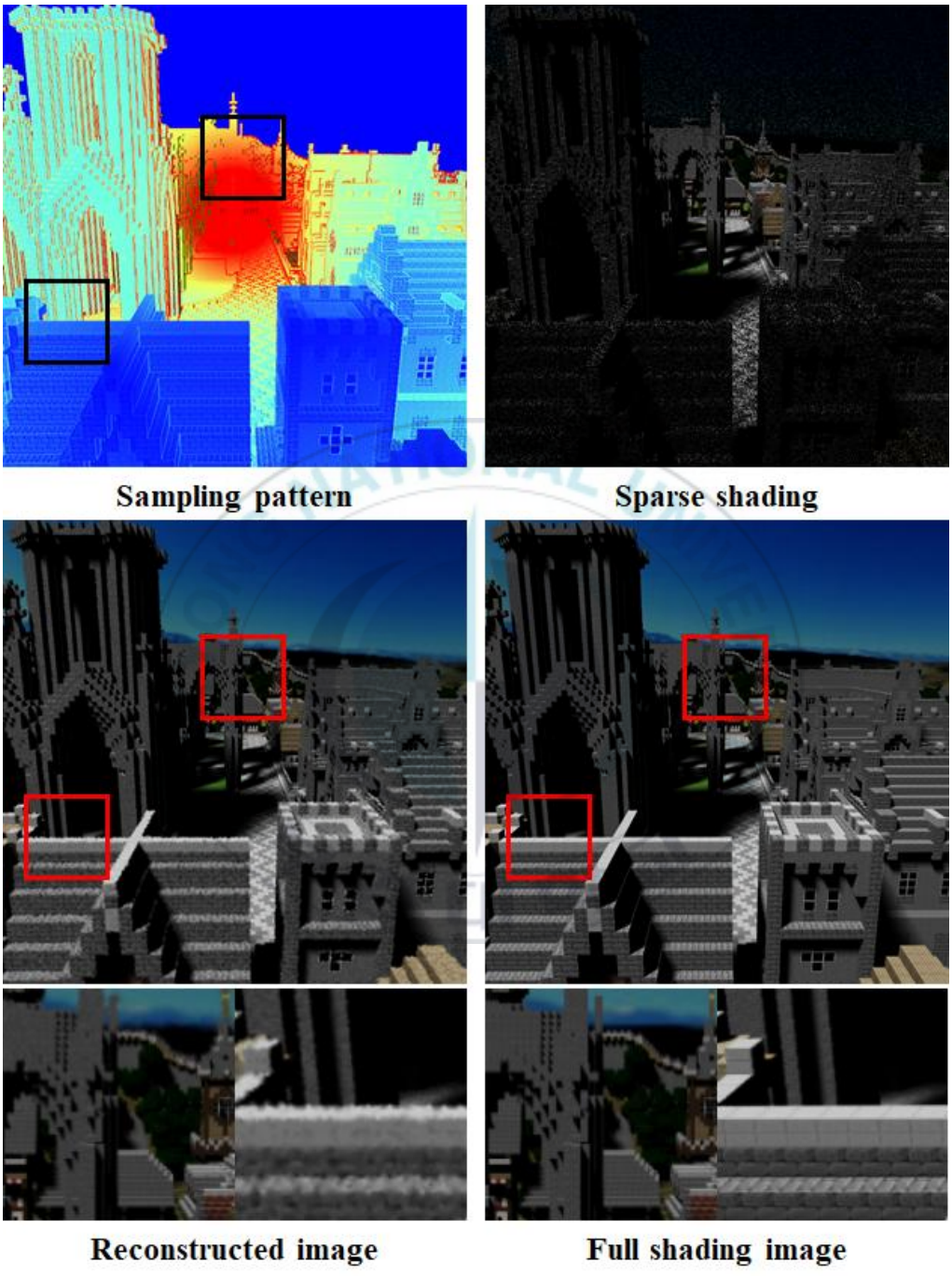

그림 42. 렌더링 결과 3. (sampling rate = 31.38%)

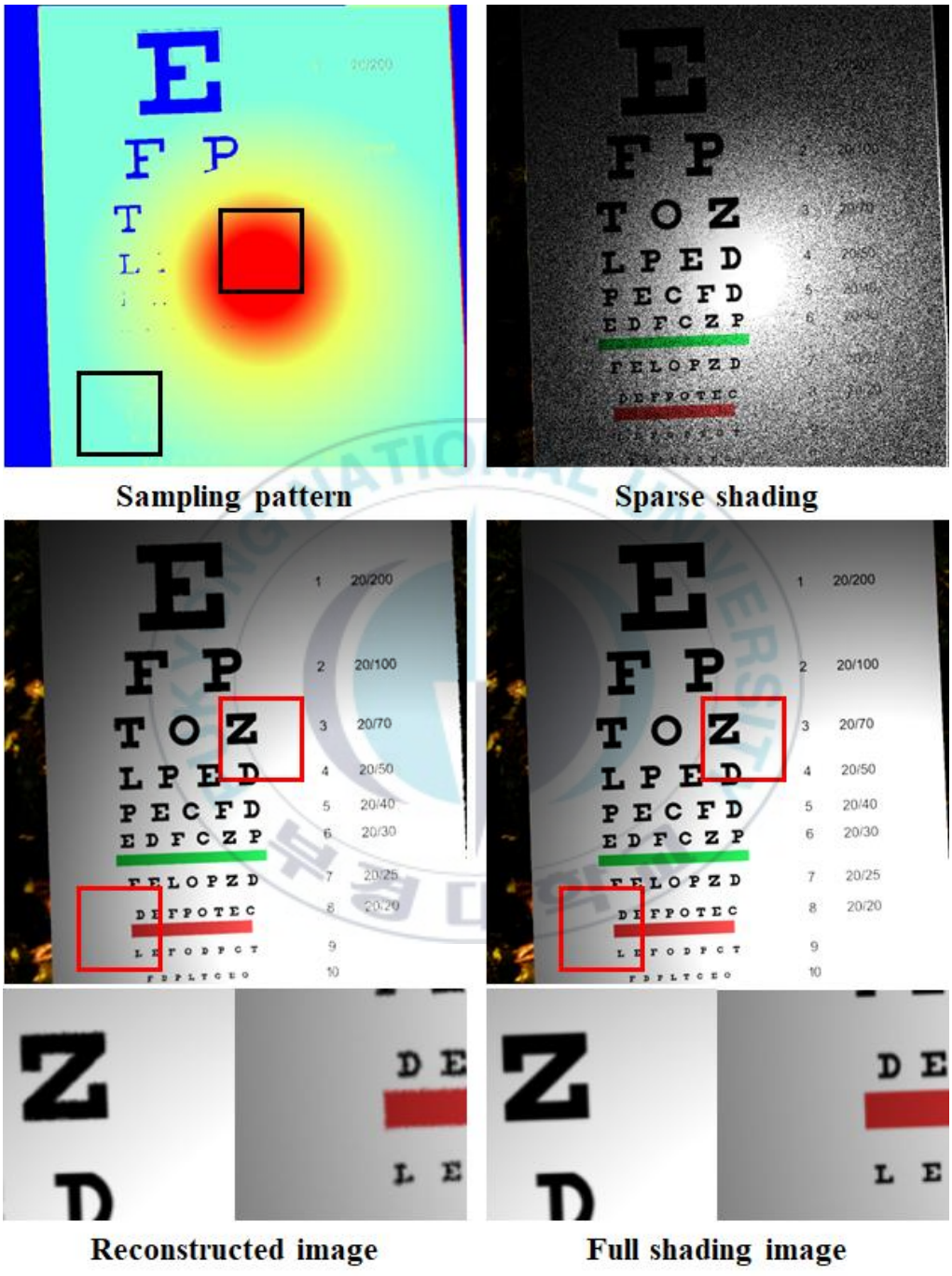

그림 43. 렌더링 결과 4. (sampling rate = 45.93%)

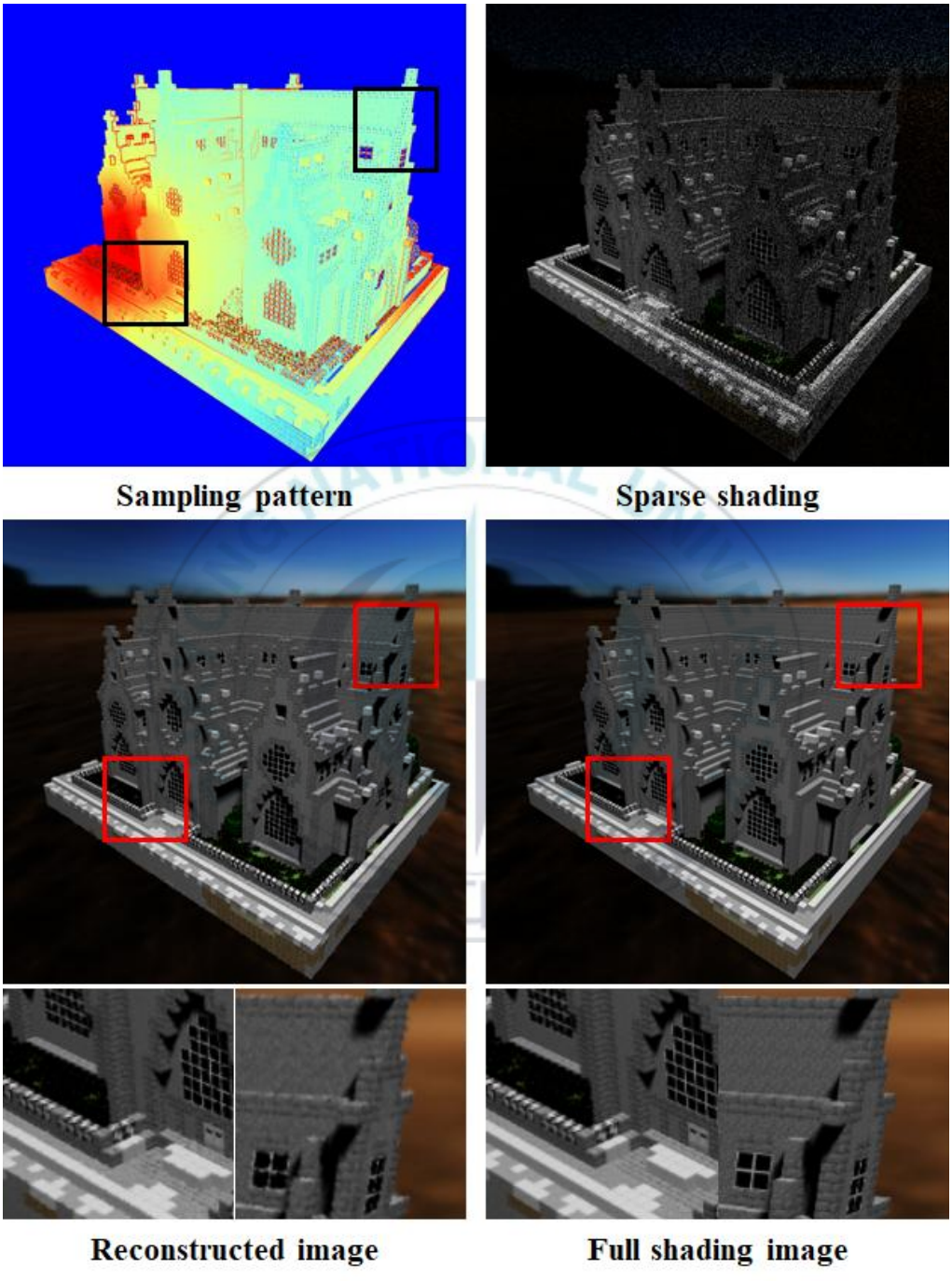

그림 44. 렌더링 결과 5. (sampling rate = 31.92%)

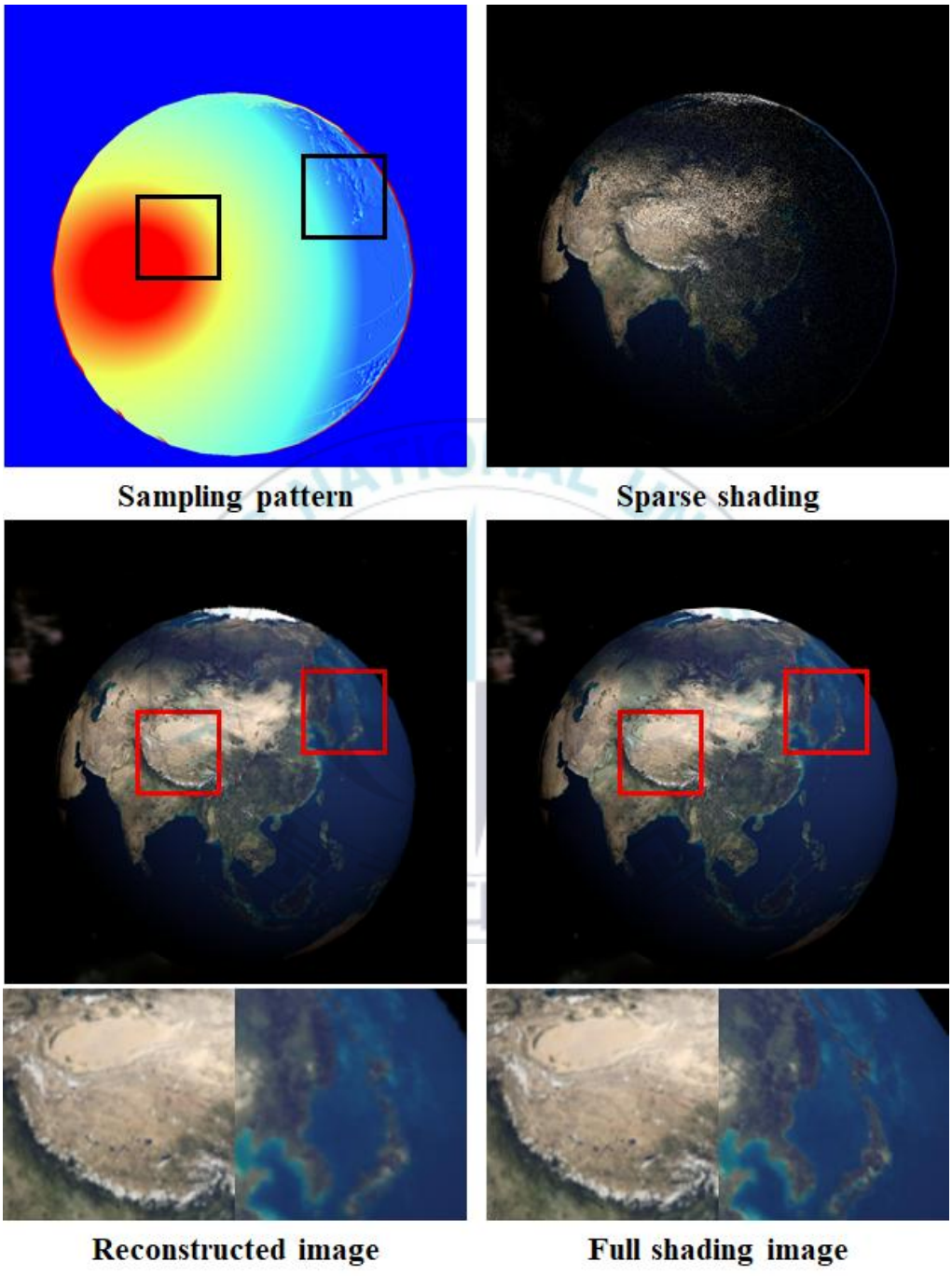

그림 45. 렌더링 결과 6. (sampling rate = 30.53%)

VI 결론

본 논문에서는 시선추적을 기반으로 포베이션 확률 및 시각적 인지, 재구성의 계산 시간을 고려한 포비티드 렌더링을 제안하였다. 제안한 방법은 인간의 시각 시스템을 모방하는 방식으로 동작하며, 인간의 망막 및 시각적인 주의, 그리고 재구성 단계에서 드는 계산 비용을 고려한 샘플링 맵을 설계하였다. 특히 GPU에서 효과적으로 동작할 수 있도록 설계된 샘플링 맵을 통하여 이전의 포비티드 샘플링으로 개선된 계산 비용을 추가적으로 줄일 수 있다. 이는 샘플링에 의해 수행되는 광선추적법을 GPU에서 효과적으로 스케줄링 되어 보다 효과적으로 동행할 수 있는 계산상의 이점을 제공한다. 또한 광선 추적법(ray tracing) 뿐만 아니라 템포럴 리프로젝션(temporal reprojection)을 복합적으로 사용하여 언더 샘플링으로 인한 희소한 픽셀 이미지를 보완하고 마지막으로 재구성 단계를 통해 최종적으로 빽빽한 이미지를 출력한다. 희소한 픽셀의 빈 부분을 보로노이 (voronoi) 다이어그램 형태로 채우고, 실시간 자연 이웃 보간법을 통해 보로노이 셀 간 경계를 부드럽게 처리하는 접근법의 성능 및 품질 평가를 통해 재구성 단계의 효과를 확인하였다. 결론적으로 제안한 방법은 인간의 시각 시스템과 유사하면서, GPU에서 효과적으로 동작하며, 매우 사실적인 고품질의 이미지를 실시간으로 획득할 수 있을 것으로 기대된다.

참고 문헌

[1] M. Weier, T. Roth, E. Kruijff, A. Hinkenjann, A. P. P□rard-Gayot, P. Slusallek, and Y. Li, "Foveated Real-Time Ray Tracing for Head-Mounted Displays," Computer Graphics Forum, Vol. 35, No. 7, pp. 289-298, 2016.

[2] O. Kwon, J. Yoon, K. Park, and Y. Kim, "Fast Ray Reordering and Approximate Sibson Interpolation for Foveated Rendering on GPU," Journal of Korea Multimedia Society, Vol. 22, No. 2, pp. 311-321, 2019.

[3] W. Hunt, "Virtual Reality: The Next Great Graphics Revolution," Keynote Talk HPG, Oculus Research, 2015.

[4] M. Koskela, T. Vitanen, P. Jaaskelainen, and J. Takala, "Foveated Path Tracing," International Symposium on Visual Computing, Springer, Cham, pp. 723-732, 2016.

[5] T. C. Ruch, and J. F. Fulton, Medical Physiology and Biophysics,AcademicMedicine,35.11,1067,1960.

[6] D. Purves, G.J. Augustine, D. Fitzpatrick, W.C. Hall, A. Lamantia, J.O. Mcnamara, et al., Neuroscienc e,SinauerAssociates,Sunderland,2004.

[7] M. F. Deering, "A Photon Accurate Model of the Human Ey e," ACM Transactions on Graphics, Vol. 24, No. 3, pp. 649-658, 2005.

[8] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N.

Benty,et al., “Towards Foveated Rendering for Gaze-Tracked Virtual Reality,” ACM Transactions on Graphics (TOG), Vol. 35.6, No. 179, 2016.

[9] M. Bauer, H. Cook, and B. Khailany, “CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization,” Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, No. 12, 2011.

[10] D. Nehab, P.V. Sander, J. Lawrence, N. Tatarchuk, and J.R. Isidoro, “Accelerating Real-Time Shading with Reverse Reprojection Caching,” Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware, pp. 25-35, 2007.

[11] C. Tyler, “Analysis of Human Receptor Density,” in Basic and Clinical Applications of Vision Science, Ed, V. Kluwer Academic Publishers, pp. 63-71, 1997.

[12] L. Gori, “A New Method for Measuring Color Acuity in Humans: A Pilot Study,” 2014

[13] J. T. Kajiya, “The Rendering Equation”, Proc. SIGGRAPH’ 86, Computer Graphics 1986.

[14] M. Fujita, and T. Harada, “Foveated Real-Time Ray Tracing for Virtual Reality Headset,” Siggraph Asia’ 14, 2014.

[15] M. Weier, M. Stengel, T. Roth, P. Didyk, E. Eisemann, M. Eisemann, et al. “Perception-drivenAcceleratedRenderin g,” Computer Graphics Forum, Vol. 36.2, pp. 611-643, 2017.

[16] H. Strasburger, I. Rentschler, and M. Juttner, “Peripheral Vision and Pattern Recognition: a review.” , Journal of Vision

[17] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder, “Foveated 3D Graphics,” ACM Transactions on Graphics (TOG), Vol. 31.6, No. 164, 2012.

[18] M. Bock, A.K. Tyagi, J. Kreft, and W. Alt, “Generalized Voronoi Tessellation as a Model of Two-dimensional Cell Tissu Dynamics,” Bulletin of Mathematical Biology, Vol. 72.7, pp. 1696-1731, 2010.

[19] G. Rong, and T. Tan, “Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transfor m,” Proceedings of the 2006 symposium on Interactive 3D Graphics and Games ACM, pp. 109-116, 2006.

[20] R. Sibson, “A Vector Identity for the Dirichlet Tessellation,” Mathematical Proceedings of the Cambridge Philosophical Society, vol. 87, no. 1, pp. 151-155, 1980.

[21] S. Park, L. Linsen, O. Kreylos, J. Owens, and B. Hamann, “Discrete Sibson Interpolation,” IEEE Transactions On Visualization And Computer Graphics, Vol. 12, No. 2, 2006.

[22] H. Dammertz, D. Sewtz, J. Hanika, and H.P.A Lensch, “ Edge-avoiding -Trous Wavelet Transform for Fast Global Illumination Filtering,” Proceedings of the Conference on High Performance Graphics, Eurographics Association , pp. 67-75, 2010.