한국컴퓨터그래픽스학회
Korea Computer Graphics Society Vol. 26, No. 3, P. 123~131

실시간 렌더링 환경에서의 3D 텍스처를 활용한 GPU 기반 동적 포인트 라이트 파티클 구현

김병진O 이택희*
한국산업기술대학교
(holdim_provae, watersp)@kpu.ac.kr

GPU-based dynamic point light particles rendering using 3D textures for real-time rendering

Kim Byeong JinO Lee Taek Hee*
Korea Polytechnic University

요 약

본 연구는 10만 개 이상의 움직이는 파티클 각각이 발광원으로서 존재할 때 라이팅을 위한 실시간 렌더링 알고리즘을 제안한다. 각 라이트의 영향 범위를 동적으로 파악하기 위해 2개의 3D 텍스처를 사용하며 첫 번째 텍스처는 라이트 색상 두 번째 텍스처는 라이트 방향 정보를 가진다. 각 프레임마다 두 단계를 거친다. 첫 단계는 Compute shader 기반으로 3D 텍스처 초기화 및 렌더링에 필요한 파티클 정보를 갱신하는 단계이다. 이때 파티클 위치를 3D 텍스처의 샘플링 좌표로 변환 후 이 좌표를 기반으로 첫 번째 3D 텍스처엔 해당 복셀에 대해 영향을 미치는 파티클 라이트들의 색상 총합을, 그리고 두 번째 3D 텍스처에 해당 복셀에서 파티클 라이트들로 향하는 방향벡터들의 총합을 갱신한다. 두 번째 단계는 일반 렌더링 파이프라인을 기반으로 동작한다. 먼저 렌더링 될 폴리곤 위치를 기반으로 첫 번째 단계에서 갱신된 3D 텍스처의 정확한 샘플링 좌표를 계산한다. 샘플링 좌표는 3D 텍스쳐의 크기와 게임 월드의 크기가 1:1로 대응하므로 픽셀의 월드좌표를 그대로 샘플링 좌표로 사용한다. 샘플링한 픽셀의 색상과 라이트의 방향벡터를 기반으로 라이팅 처리를 수행한다. 3D 텍스처가 실제 게임 월드와 1:1로 대응하며 최소 단위를 1m로 가정하는데 1m보다 작은 영역의 경우 해상도 제한에 의한 계단 현상 등의 문제가 발생한다. 이러한 문제를 개선하기 위한 텍스처 샘플링 시 보간 및 슈퍼 샘플링을 수행한다. 한 프레임을 렌더링하는데 소요된 시간을 측정한 결과 파티클이 라이트의 개수가 262144개일 때 Forward Lighting 파이프라인에서 146ms, deferred Lighting 파이프라인에서 46ms 가 소요되었으며, 파티클 라이트의 개수가 1024576개일 때 Forward Lighting 파이프라인에서 214ms, Deferred Lighting 파이프라인에서 104ms 가 소요되었다.

Abstract

This study proposes a real-time rendering algorithm for lighting when each of more than 100,000 moving particles exists as a light source. Two 3D textures are used to dynamically determine the range of influence of each light, and the first 3D texture has light color and the second 3D texture has light direction information. Each frame goes through two steps. The first step is to update the particle information required for 3D texture initialization and rendering based on the Compute shader. Convert the particle position to the sampling coordinates of the 3D texture, and based on this coordinate, update the colour sum of the particle lights affecting the corresponding voxels for the first 3D texture and the sum of the directional vectors from the corresponding voxels to the particle lights for the second 3D texture. The second stage operates on a general rendering pipeline. Based on the polygon world position to be rendered first, the exact sampling coordinates of the 3D texture updated in the first step are calculated. Since the sample coordinates correspond 1:1 to the size of the 3D texture and the size of the game world, use the world coordinates of the pixel as the sampling coordinates. Lighting process is carried out based on the color of the sampled pixel and the direction vector of the light.

*corresponding author: Lee Taek Hee/Korea Polytechnic University(watersp@kpu.ac.kr)

Received : 2020.06.21./ Review completed : 1st 2020.06.22. / Accepted : 2020.06.25.
DOI : 10.15701/kcgs.2020.26.3.123
ISSN : 1975-7883(Print)/2383-529X(Online)

The 3D texture corresponds 1:1 to the actual game world and assumes a minimum unit of 1m, but in areas smaller than 1m, problems such as stairs caused by resolution restrictions occur. Interpolation and super sampling are performed during texture sampling to improve these problems. Measurements of the time taken to render a frame showed that 146 ms was spent on the forward lighting pipeline, 46 ms on the defered lighting pipeline when the number of particles was 262144, and 214 ms on the forward lighting pipeline and 104 ms on the deferred lighting pipeline when the number of particle lights was 1,024766.

키워드: 파티클, 라이팅, 3D 텍스처, 컴퓨트 셰이더
Keywords: Particle, Lighting, 3D Texture, Compute Shader

1. 서론

게임은 플레이어와 게임 월드와의 상호작용이 주가 되는 멀티미디어 콘텐츠이다. 그리고 게임 월드의 환경 및 플레이어와 게임 월드 간의 상호작용을 가장 효과적으로 표현하기 위한 수단으로 파티클을 활용한 시각 효과를 주로 사용한다. 특히 플레이어 캐릭터의 상태 및 특수 기술, 그리고 플레이어가 게임 월드에 대해 필수적인 상호작용 요소에 대한 강조의 표시로 스스로 빛을 내며 발광하는 빌보드 파티클을 사용하는 경우가 많다. 그러나 파티클 빌보드 하나하나를 포인트 라이트 형태로 다루게 될 경우 한 화면을 그리는 것에 대한 라이트 연산 부하가 과도하게 들어가 60fps(frames per second)를 유지하는 것이 불가능하다. 따라서 기존 방식은 파티클 하나하나를 포인트 라이트로 다루는 대신 투영된 결과를 기반으로 Bloom 등의 효과를 주어 발광체인 것처럼 모사하는 방법을 사용한다.

상용 게임엔진의 파티클 시스템에도 각각의 파티클에 대한 동적 파티클 라이트를 지원하고 있으나 실시간 렌더링을 위해선 수십 개 이내로 동적 파티클 라이트 개수를 줄여야 한다. 이에 대한 보완책으로 파티클에 블룸 효과를 주거나 파티클 군집의 중심에 단일 포인트 라이트를 배치하여 라이트 효과를 근사하는 방식이 주로 사용되고 있다.

본 논문에서는 실시간 렌더링 환경에서 10k 개 이상의 파티클 각각이 동적 라이트로서 동작할 때 렌더링 과정에서 파티클 라이트의 개수와 관계없이 안정적인 프레임 레이트를 유지할 수 있는 알고리즘을 제안한다. 이를 위해 라이팅 연산에 필요한 정보를 가질 수 있는 3D 텍스처를 2 개를 유지한다. 각 3D 텍스처는 전체 월드 크기에 맞춰 생성되며 1 복셀의각축별 크기는 1m 로 한다. 이 두 개의 3D 텍스처는 각각 해당 복셀에 대해 영향을 주는 라이트의 색상, 그리고 정반사 계산을 위한 라이트의 방향벡터를 저장하는 용도로 사용한다. 매 프레임마다 각각의 복셀을 게임 월드의 위치와 대응한 후 라이팅 연산 결과를 미리 계산한다.

난반사 연산의 경우 라이트의 색상과 라이트의 감쇠 계수 및 라이트와 복셀에 대응하는 위치와의 거리를 사용하여 해당 복셀에 대한 라이트 색상을 저장한다. 정반사 연산의 경우 카메라의 시선 벡터를 필요로 하므로 각각의 복셀에 대해 복셀의 좌표로부터 해당 복셀에 영향을 주는 라이트의 위치로 향하는 방향벡터의 합을 구한 후 또 다른 3D 텍스처에 저장한다. 방향벡터의 합을 구할 때 각각의 라이트의 방향벡터에 라이트의 밝기 값을 곱한 결과를 합 연산에 반영한다.

실제 렌더링 단계에선 화면에 그리고자 하는 픽셀에 대한 라이팅 연산을 반영할 때 해당 픽셀의 월드 위치를 3D 텍스처의 UVW 좌표로 활용하여 미리 계산된 라이트 색상과 라이트의 방향벡터 합을 샘플링한다. 샘플링 된 두가지 정보를 기반으로 Phong 라이팅 모델에 적용하여 최종 결과를 얻는다.

2. 관련 연구

한 게임 월드 내에 라이트 개수가 많아지면 많아질수록 한 화면을 렌더링하는 데 소요하는 시간이 길어져 높은 프레임 레이트를 유지하기 힘들어진다. 상용 게임과 같이 화면을 실시간으로 렌더링하는 응용 프로그램의 경우 실감 나거나 화려한 시각효과를 그림과 동시에 높은 프레임 레이트를 유지하기 위해 라이트와 같이 많은 계산을 해야 하는 시각 효과의 경우 미리 계산된 결과를 활용하거나 또는 Phong 모델과 같은 일반적으로 사용되는 라이팅 연산 모델을 사용하지 않고 라이팅 연산을 모사하는 방법을 사용한다.

Ola Olsson 은 화면 공간 (screen space)을 격자로 나누고 각 격자에 대해 영향을 끼치는 라이트만 계산하는 컬링 과정을 거쳐 광범위한 라이트에 대한 연산량을 줄였으나 2D 공간상에 여러 라이트가 겹치는 상황에서 깊이가 구분되지 않아 2D 화면상 픽셀이 실제로 영향받는 라이트보다 더 많은 수의 라이트에 의한 영향을 받는 것으로 취급되는 문제가 있었다. [2][3] 이에 대한 후속 연구로 Clustered Deferred and Forward Shading 기법이 제안되었으며 단순히 화면공간 상에서 격자를 나누는 것이 아니라 3D cluster Grid 를 이용하여 각각의 3D 클러스터 격자에 대해 교차하는 라이트들의 리스트를 저장하고 투영공간에서 각각 클러스터 영역에 해당하는 픽셀에 대해 리스트에 저장된 라이트에 대해서만 연산을 수행하여 전체 라이팅 연산량을 줄이는 방법을 제안하였다.

2017 년 Yuriy O'Donnell 은 CPU 에서 라이트를 절두체 컬링 한 후 남아있는 라이트들을 라이트의 위치와 카메라 평면 간의 거리별로 정렬한 후 라이트를 화면 공간에서 타일에 넣은 다음 스크린 스페이스의 타일마다 라이트-트리를 생성한 후, GPU 에서 라이팅 패스를 수행하는 도중 현재 렌더링하고자 하는 픽셀을 덮고 있는 라이트-트리를 식별한 후 해당 라이트-트리에 포함되는 라이트에 대한 조명 연산을 수행하는 방식을 제시하였다.[5]

Cem Yuksel 은 폭발과 같이 수많은 파티클이 자체적으로 포인트 라이트로서 화면에 존재할 때 포인트 라이트 전체를 연산하지 않고 라이트가 존재하는 공간에 대해, 마치 텍스처의 밉맵과 같이 여러 단계의 해상도를 가진 라이트 그리드로 이루어진 계층구조를 생성한 후 그리드 당 하나의 포인트 라이트를 가지도록 하여 라이팅 계산에 반영해야 하는 포인트 라이트의 개수를 줄이고 낮은 해상도 단계의 라이트 그리드의 포인트 라이트의 위치는 높은 해상도 단계의 라이트 그리드의 포인트 라이트의 위치 값을 기반으로 보정하는 방식으로 계산 후 각 단계의 라이팅 결과를 블랜딩하는 방식을 제시하였다.[6]

Cem Yuksel 은 2019 년에 라이트 그리드 계층 구조 알고리즘을 확장하여 가상의 수많은 포인트 라이트를 생성한 후 이들을 사용한 라이트 그리드 계층 구조를 만들어 전역 조명을 렌더링하는 방법을 제시하였다.[7]

본 논문에서는 다수의 포인트 라이트를 다루는 다른 관련 연구에서 최적화를 위해 사용한 라이트 컬링 또는 공간 분할을 기반으로 한 라이트 최적화 기법을 사용하지 않고 게임 월드에 대응하는 3D 텍스처를 만든 후 컴퓨트 셰이더에서 3D 라이트 텍스처의 각 복셀에 대한 라이트 색상 반영과 라이트 방향 벡터를 미리 반영한 후 라이팅 단계에서 그리고자 하는 픽셀에 대해 영향을 주는 라이트 들을 순회 및 탐색할 필요 없이 픽셀의 월드 좌표를 그대로 3D 텍스처의 샘플링 좌표로 사용한 후 라이트의 정보를 읽어와 라이팅 하는 방법을 적용하여 성능을 측정 및 분석한 후, 향후 연구에서 관련 연구의 최적화 기법을 적용하여 비주얼 개선과 퍼포먼스 향상시키는 것을 목표로 한다.

3. 환경 구성

3.1 3D Light Texture

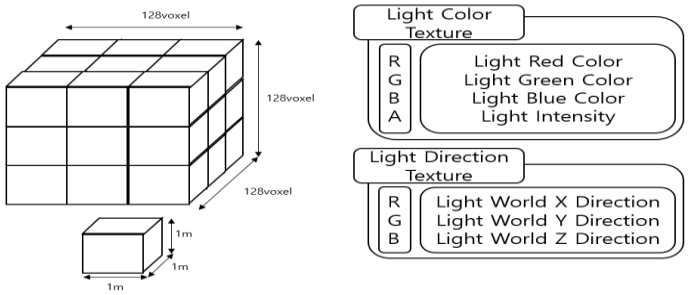

Figure 1: Structure of 3D Light Color Texture and direction Texture

본 논문에서는 게임 환경에서의 라이트의 광도 및 색상정보를 저장하기 위해 128*128*128 크기의 3D 텍스처 두 개를 사용한다. 여기서 복셀의 크기는 figure 1 의 왼쪽 이미지와 같이 게임 월드 1m 와 1:1 로 대응하였다. 첫 번째 3D 텍스처는 라이트 색상 텍스처로써 라이트의 색상과 광도를 저장하기 위한 용도로 사용하며 RGBA 채널을 사용한다. RGB 채널은 해당 파티클 라이트의 색상, A 채널은 파티클 라이트의 광도를 저장한다.

두 번째 3D 텍스처는 라이트 방향 텍스처로써 3D 환경의 해당 좌표에 영향을 주고 있는 라이트의 방향을 저장하는 용도로 사용하며 RGB 채널을 사용한다. Light 방향 텍스처는 라이트가 표면에 비출 때 나타나는 정반사 광의 색상을 계산하는 데 사용한다.

3.2 Particle Structure

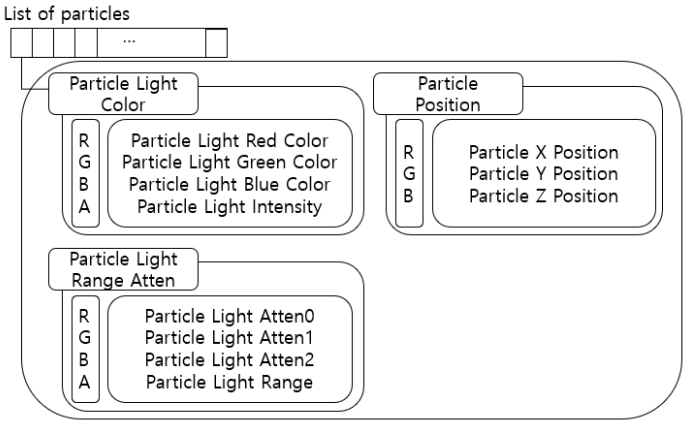

Figure 2: Structure of each Particle information for Lighting

본 논문에서의 파티클은 Opengl 그래픽 API 에서 제공하는 Shader Storage Buffer Object 로 관리된다. 매 3D 텍스처의 라이트 정보를 갱신하는데 사용하는 파티클의 구조는 Figure 2 와 같은 구조를 사용한다. 파티클의 라이트가 월드에 영향을 미치는 위치를 나타내기 위한 Particle Position 변수, 파티클이 지닌 라이트의 색상을 표현하기 위한 Particle Light Color 변수, 그리고 파티클의 라이트가 영향을 미치는 범위를 표현하기 위한 Particle Light Range Atten 변수를 사용한다.

4. Update Particle Light Texture

시간의 흐름에 따른 파티클의 위치 변화와 그에 따른 3D 라이트 텍스처 초기화와 갱신과정은 컴퓨트 셰이더를 이용한다.

매 프레임마다 파티클 시뮬레이션을 진행한 직후 각각 파티클의 위치가 변동됨에 따라 월드의 각각의 지점이 받는 라이트의 색상 및 광도가 변동됨에 따라 라이트 색상 텍스처와 라이트 방향 텍스처를 갱신해야 한다. 월드의 라이트 정보를 갱신하기 전 두 개의 라이트 텍스처의 모든 복셀의 색상 값을 0 으로 초기화 한다.

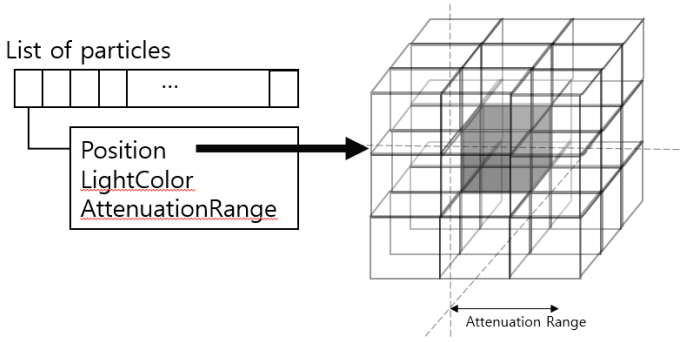

Figure 3: Access Light Texture Voxel by each particle's position data

그다음 파티클의 정보를 담고 있는 배열을 차례대로 순회하며 각각 파티클의 위치와 색상을 읽는다. 그 다음 이 단계에서 읽어낸 파티클의 위치를 3D 라이트 텍스처의 UV 좌표로써 사용한다. 그 다음 파티클의 위치와의 거리 값이 현재 참조하고 있는 파티클의 Particle Light Range Atten 변수의 A 채널 값인 Particle Light Range 변수보다 작은 모든 복셀을 순회하며 3D 라이트 텍스처와 라이트 방향 텍스처의 색상정보를 갱신한다. 파티클의 위치와 라이트 텍스처의 복셀간 거리 DistValue 변수는 GLSL 함수 distance(Particle Position, Voxel Position)을 사용한다.

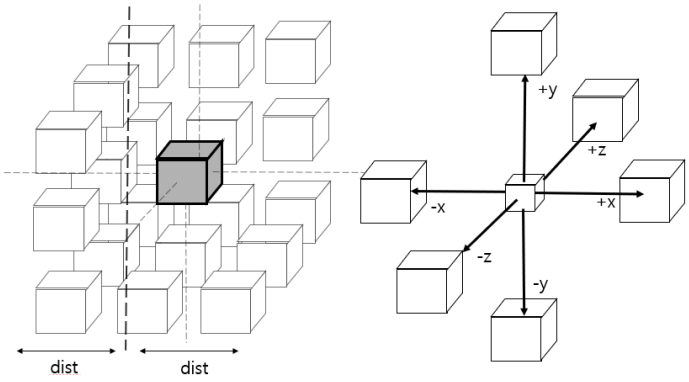

Figure 4: Access Range for 3D Light Texture

Figure 4 와 같이 파티클의 위치로부터 ±x 축 ±y 축 ±z 축 방향으로 – DistValue 부터 +DistValue 만큼 떨어져 있는 모든 복셀이 순회에 포함되며 최종적으로 각 파티클당 총 (2*DistValue)^3 개의 복셀을 갱신하게 된다.

다음은 각각의 복셀에 대한 라이트의 방향과 색상을 갱신한다. 파티클의 위치와 갱신하는 복셀의 거리에 따른 밝기의 감쇠를 반영하기 위해 파티클의 Particle Light Range Atten 변수의 RGB 실수 값을 감쇠 계수로 사용한다.

복셀의 위치와 파티클의 중점 간 거리에 따른 감쇠 공식은 다음 식을 이용한다.

$$Distvalue = distance(Particle\ Position, Pixel\ Position)$$

$$Attenuation = \frac{1}{ParticleLightRangeAtten.x + ParticleLightRangeAtten.y * Distvalue + ParticleLightRangeAtten.z * Distvalue^2}$$

위 식을 이용하여 계산한 감쇠 계수를 이용한 최종 색상은 Attenuation * ParticleLightColor 이며 이 결과를 현재 순회하고 있는 라이트 색상 텍스처의 복셀에 저장한다. 정반사 계산에 사용할 라이트의 방향 값의 경우 파티클의 위치에서 현재 복셀의 위치의 차이 값을 정규화한 float3 크기의 벡터를 이용한다. 이때 라이트의 밝기에 따른 가중치를 반영하기 위해 계산한 라이트 방향 값에 파티클 라이트 컬러의 A 채널 값을 곱한 후 저장한다. 이 과정을 의사 코드로 표현하면 다음과 같다.

Particle World Intensity Update

```
loop particle in scene.particle
  ivec3 texcoord = ivec3(particle.position)
 int dist = int( particle.AttenRange.w)+1
 loop i in -dist <= i <= dist
  loop j in -dist <= j <= dist
   loop j in -dist <= k <= dist
    ivec3 pixel_coordinate = texcoord + ivec3(i,j,k)
    float distanceValue
          = distance(particle.position, pixel_coordinate)
    float attenuation
           = 1.0 / particle.AttenRange.x + particle.AttenRange.y * distanceValue +
           particle.AttenRange.z * distanceValue^2
    vec4 result_color = attenuation * vec4( particle.LightColor)
    vec4 result_direction
           = normalize(particle.position - pixel_coordinate) * result_color.a
    vec4 stored_color = imageLoad(LightColorTexture)
    vec4 stored_direction
            = imageLoad(Light direction Texture)
    imageStore(LightColorTexture,pixel_coordinate, stored_color + result_color )
   ImageStore(Light direction Texture,pixel_coordinate, stored_direction +
        result_direction )
```

5. Lighting method

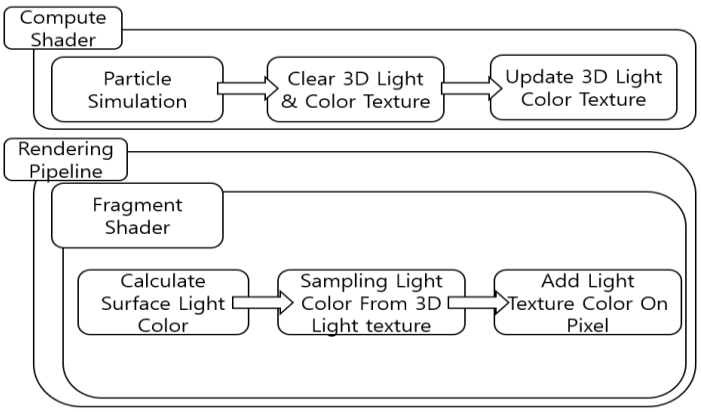

Figure 5: Rendering Pipeline

컴퓨트 셰이더에서 3D Light 색상 텍스처 와 3D Light 방향 텍스처를 갱신한 후 이 결과를 렌더링 파이프라인의 프래그먼트 셰이더에서 사용한다. 게임 월드와 3D Light 텍스처의 크기가 1:1 로 대응하기 때문에 프래그먼트 셰이더에서 렌더링하는 픽셀의 월드좌표를 그대로 3D 텍스처의 UVW 샘플링 좌표로 사용할 수 있다. 다만 렌더링 파이프라인에서 샘플링 함수를 사용할 때 UVW 좌표 값은 0 부터 1 사이의 정규화 된 값을 사용한다. 따라서 현재 픽셀의 월드 좌표를 3D 라이트 텍스처의 크기로 나누어 0 부터 1 사이의 값으로 정규화 된 값을 UVW 샘플링 좌표 값으로 사용한다.

파티클 라이트가 월드에 반영이 될 때 광도와 색상은 컴퓨트 셰이더를 기반으로 갱신된다. 이 때문에 렌더링 파이프라인에서 파티클 라이트를 반영하기 위해선 파티클 라이트를 제외한 다른 라이트 연산을 끝낸 후 반영해야 한다. 다른 라이트 연산이 끝난 최종 색상에 3D 라이트 색상 텍스처와 3D 라이트 방향 텍스처의 샘플링 색상 값을 이용해 연산한 파티클 라이트의 색상을 가산하여 파티클 라이트 반영을 수행한다.

본 논문에서는 Phong 라이팅 모델을 사용하였다. 파티클 라이트의 난반사를 반영하기 위해 3D 라이트 색상 텍스처의 RGBA 채널을 샘플링한 후 다른 라이트 연산이 끝난 색상 결과에 샘플링 결과를 가산한다.

Phong 모델에서 정반사 계수를 구하기 위해서는 표면에서 카메라를 향하는 뷰 벡터와 표면에서 라이트로 향하는 라이트 방향벡터가 필요하다. 파티클 라이트에서의 라이트 방향은 3D 라이트 방향 텍스처의 RGB 채널 색상값을 샘플링한 값을 사용한다. 이때 3D 파티클 라이트 텍스처 갱신 단계에서 파티클의 위치로 향하는 방향벡터에 파티클 라이트의 광도를 곱하는 방법으로 가중치를 표현하였기 때문에 3D 라이트 방향 텍스처에는 정규화 되어있지 않은 값이 저장되어 있다. 따라서 3D 라이트 방향 텍스처를 샘플링한 후 이 값을 -1 과 1 사이의 값으로 만든 다음 정반사 모델의 라이트 벡터로 사용한다.

6. 3D Texture Filtering

6.1 3D texture interpolation method

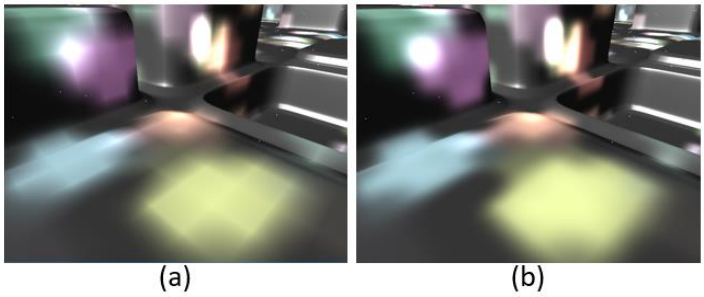

Figure 6: (a) Lighting Result bilinear interpolation (b) Lighting result trilinear interpolation

3D 라이트 텍스처의 경우 게임 월드와 정확하게 1:1 로 대응됨과 동시에 해상도가 고정되어 있어 카메라가 월드의 좁은 영역을 가까이서 관측할수록 라이팅 결과에서 사각형 형태의 아티팩트과 관측됨을 확인할 수 있다.

이러한 아티팩트를 개선하기 위해선 단일 복셀을 사용하는 것이 아니라 주변 복셀의 색상 값까지 고려한 보간방법을 사용해야 한다.

3D 라이트 텍스처 에서 색상값을 샘플링하는 과정에서 Figure 6 의 (a) 이미지와 같이 선형 보간을 사용하여 1x1x1 크기의 복셀의 양 끝 8 개의 버텍스가 지닌 색상 값과 3D 텍스처의 단일 복셀 안에서의 샘플링 위치와 복셀의 버텍스간의 상대적인 위치까지 고려하여 샘플링해 더 부드러운 라이팅 결과를 도출할 수 있으나 여전히 복셀과 복셀간의 선명한 아티팩트가 드러나는 것을 확인할 수 있다.

이를 완화하기 위해 Figure 6 의 (b)와 같이 보간 가중치를 반영하기 위한 버텍스의 개수를 4x4x4 로 확장한 삼선형 보간을 사용하여 결과를 개선할 수 있으나 여전히 각이져있는 비주얼을 확인할 수 있다.

6.2 3D texture oversampling method

오버 샘플링은 텍스처의 값을 샘플링하는 과정에 있어 하나의 복셀 좌표에 따른 하나의 색상값 뿐만 아니라 그 주변에 해당하는 다른 복셀 좌표들의 색상값 들을 샘플링한 다음 이들의 평균을 최종 샘플링 결과로 활용하는 기법이다.

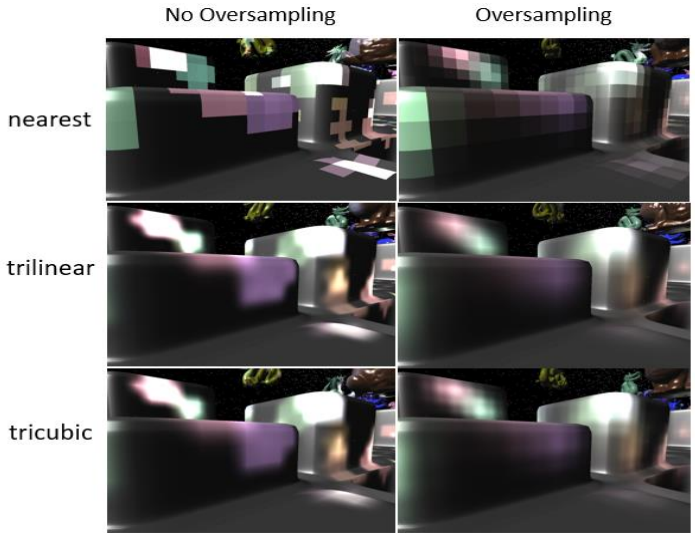

Figure 7: Rendering Result with Difference interpolation Method and Compare result of Oversampling

Figure 7 은 각각의 보간 방식 그리고 각각의 보간 방식에 오버샘플링이 적용된 경우와 적용되지 않은 경우의 렌더링 결과를 나타낸다. 최근접 보간의 경우 오버 샘플링 적용 여부와 관계없이 사각형의 계단 현상(aliasing)과 아티팩트가 나타나는 것을 확인할 수 있다. 다만 오버샘플링이 적용됨으로써 오브젝트의 표면에 반영된 라이트의 색상이 오버샘플링이 적용되지 않은 결과보다 더 부드럽게 반영된 것을 확인할 수 있다. Tricubic 보간은 스플라인 곡선을 사용하여 보간하는 방법이며 본 논문에서 사용된 Tricubic 보간 에서의 스플라인 곡선의 보간은 Catmull-rom 보간법을 사용한다. 삼선형 보간에 비해 더 많은 연산이 필요하지만, 더 부드러운 결과를 도출할 수 있다.

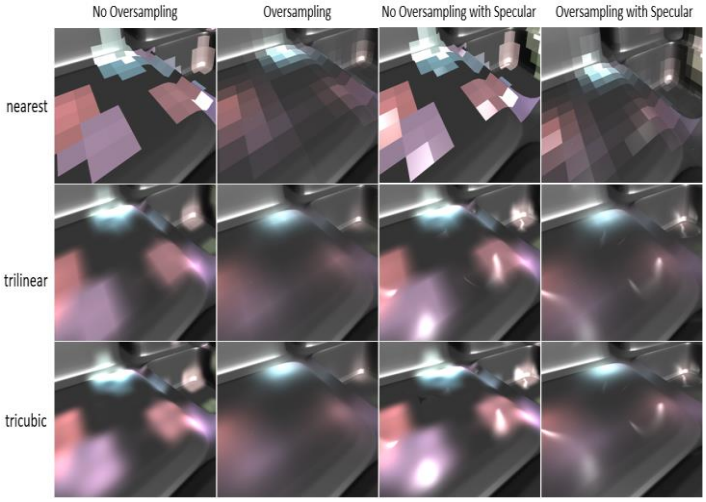

Figure 8: Diffuse and Specular Lighting Result with Difference interpolation Method and Compare result of Oversampling

Figure 8 왼쪽 두 개 열에서 확인할 수 있듯, 난반사 연산의 경우 삼선형 보간법과 Tricubic 샘플링 간 차이가 크게 나지 않는다. 반면 정반사 라이팅의 경우 삼선형 보간과 Tricubic 보간 간 정반사 광의 모양의 차이가 있음을 확인할 수 있다.

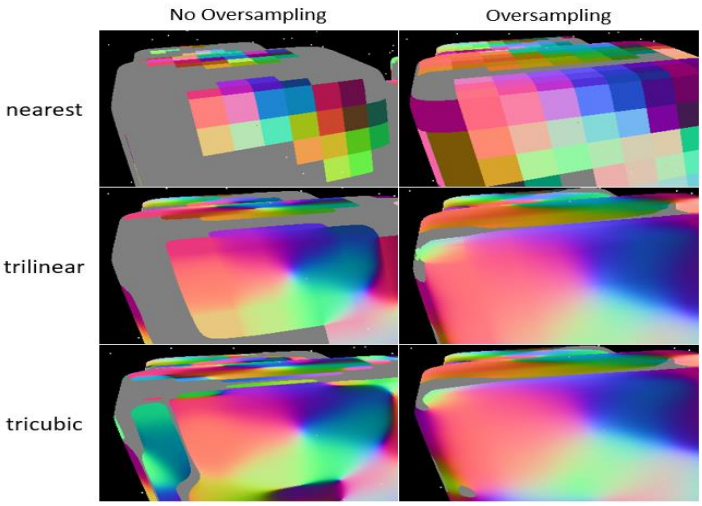

Figure 9: Light direction Texture Sampling Result with Difference interpolation method and Oversampling

Figure 9 은 각각의 보간 방식과 오버샘플링 적용 여부에 따른 라이트 방향 텍스처의 색상 샘플링 결과를 그대로 출력 색상으로 하여 렌더링하였을 때의 결과다 오버샘플링을 적용하였을 경우 삼선형 보간법과 Tricubic 보간법 간 샘플링 결과의 큰 차이가 보이지 않으나 오버샘플링이 적용 되어있지 않은 경우 파티클 라이트가 적용되지 않는 부분과 적용되는 부분 사이의 경계 부분에서 큰 차이가 있음을 확인할 수 있다.

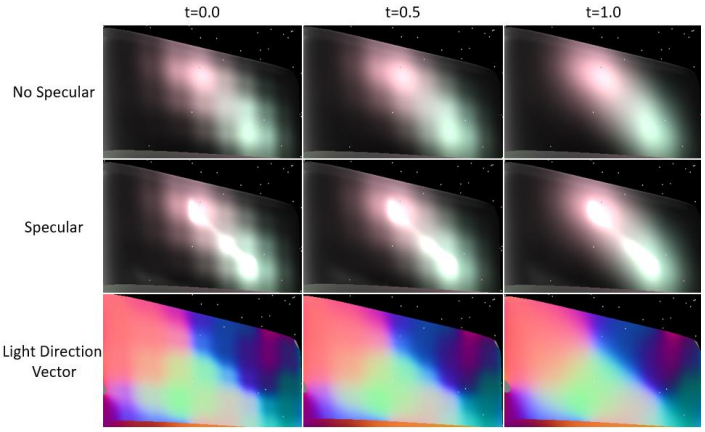

Figure 10: Rendering Result with Difference interpolation Method and Compare result of Oversampling

Tricubic 보간을 수행할 때 스플라인 곡선에 사용되는 tension 값에 따라 스플라인 곡선의 보간 형태가 달라지는데, 오버샘플링이 비활성화된 경우 tension 값이 0 일때 가장 부드러운 곡선이 생성되나 오버샘플링 이 수행될 경우 Figure 10 의 오른쪽 이미지와 같이 tension 값이 1.0 일때 가장 부드러운 형태가 나타난다.

Number of Particle	16384	65536	262144	1024576
Forward Lighting	142ms	146ms	162ms	214ms
Deferred Lighting	32ms	46ms	48ms	104ms

Figure 11: Comparing Performance on Difference of Number of Particles and Rendering Pass

7. 결론 및 향후 연구

본 논문에서 제안한 파티클 라이팅 시스템이 구동된 환경은 OPENGL 4.3 버전 그래픽 API 를 사용하였으며 그래픽카드는 NVIDIA GTX 980 모델을 사용하였다.

성능 측정에 있어 기준이 되는 게임 월드는 128m*128m*128m 크기를 상정하였으며 라이트 색상 텍스처 와 라이트 방향 텍스처 의 크기 또한 128*128*128 를 사용하였다. 그리고 동적인 게임 월드를 표현하기 위해 매 프레임마다 y 축을 기준으로 무작위 방향과 속도를 가지고 회전하는 Bunny 오브젝트와 Dragon 오브젝트를 각각 50 개 무작위 위치에 배치하였다.

게임 월드에서의 오브젝트를 나타내기 위해 사용한 Bunny Model 은 91’014 개의 정점을, Dragon Model 은 65’346 개의 정점을 가지고 있다. 성능 측정은 포워드 렌더링 패스와 디퍼드 렌더링 패스에서 파티클 개수에 따른 한프레임을 그리기 위해 소요되는 시간을 측정하였다. Figure 11 은 파티클의 개수가 각각 16’384 , 655’36 , 262’144 , 1’024’576 개일 때 각각 라이트 패스에 따른 소요 시간을 나타내고 있다. 포워드 렌더링 패스보다 디퍼드 렌더링 패스에서 더 적은 시간이 소요되며 동시에 파티클 개수의 증가에 따른 소요 시간의 증가 폭이 더 적음을 확인할 수 있다.

현재 게임 월드의 1m 와 텍스처의 크기가 1:1 대응하는 상황이기에 중거리에 있는 오브젝트 표면에 파티클 라이트가 반영될 경우 그럴듯한 라이팅 결과를 도출할 수 있으나, 게임 환경에서 카메라가 좁은 환경을 비추고 있을수록 일반적인 라이팅 결과와 비교하였을 때 화면에 반영되는 파티클 라이트의 결과가 조악하게 표현된다. 또한 그림자를 반영하는 연산 과정이 적용되어 있지 않기 때문에 포인트 라이트의 영향권 내에 차폐물이 있더라도 그림자가 생성되지 않는 문제가 있다.

Figure 12: Specular artifact on surface

그리고 파티클의 빛에 의한 정반사를 3D 라이트 방향 텍스처에서 샘플링한 값을 라이트 반사모델의 라이트 방향벡터로 사용하는데 이때 3D 라이트 방향 텍스처 에는 해당 복셀에서 해당 복셀에 영향을 주는 파티클로 향하는 방향벡터의 합산 값을 정규화된 값을 가지고 있기 때문에 Figure 12 와 같이 파티클이 산발적으로 위치한 경우 정반사의 아티팩트가 발생할 수 있다.

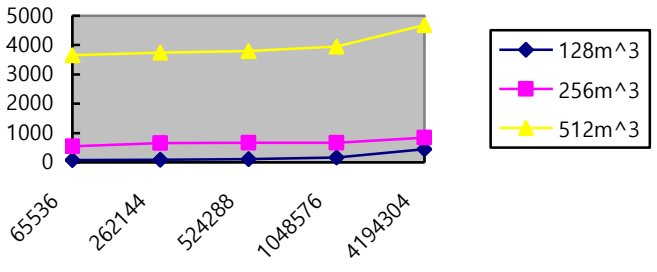

Table 1: Compare Performance Between Scene 3D Texture size and Number of Particles

Number of Particle	65536	262144	524288	1048576	4194304
128m^3	79ms	89ms	111ms	164ms	449ms
256m^3	545ms	653ms	662ms	670ms	846ms
512m^3	3659ms	3737ms	3798ms	3951ms	4685ms

Table 1 은 파티클 라이트의 감쇠 거리가 0.5m 일 때 게임 월드와 파티클 3D 텍스처의 크기와 파티클의 개수에 따라 프레임 하나를 렌더링하기 위해 소요되는 시간을 측정한 것이다. 같은

파티클 3D 텍스처의 크기와 조건에 대해 파티클의 개수에 따른 프레임 소요 시간의 상승 폭이 크지 않으나 파티클 3D 텍스처의 크기가 커지면 공간 복잡도 또한 제곱으로 늘어나 렌더링 소요 시간이 급격하게 늘어난다.

따라서 3D 텍스처의 크기가 큰 환경에 대응하기 위해서 카메라의 뷰 절두체에 포함되지 않는 파티클을 컬링 하거나 텍스쳐의 라이트 정보를 갱신할 때 뷰 절두체에 포함되지 않는 복셀의 경우 라이트 갱신을 생략하여 연산양을 줄여야 할 필요성이 있다.

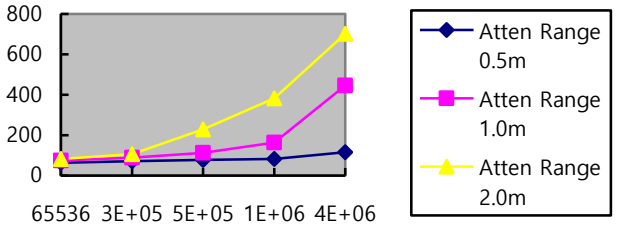

Table 2: Compare Performance Between Particle Light Attenuation Distance and Number of Particles

Number of Particle	65536	262144	524288	1048576	4194304
Atten 0.5m	64ms	71ms	77ms	83ms	116ms
Atten 1m	75ms	89ms	112ms	164ms	446ms
Atten 2m	83ms	106ms	228ms	383ms	701ms

각각의 파티클의 라이트가 영향을 주는 영역이 클 수록 3D 라이트 색상 텍스처와 3D 라이트 방향 텍스처에 파티클 라이트의 정보를 갱신하기 위해 순회해야 하는 복셀의 개수가 늘어나기 때문에 각각의 파티클 라이트가 영향을 주는 감쇠 영역이 크면 클수록 공간 복잡도가 높아져 높은 프레임 레이트를 유지하기 힘들다.

Table 2 는 128m x 128m 게임 환경일 때 3D 텍스처의 정보를 샘플링하는 과정에서 보간 과 oversampling 을 하였을 때 파티클의 개수와 파티클 라이트의 감쇠 거리에 따라서 한 프레임을 렌더링하는데 필요한 평균 시간을 계산한 것이다. 감쇠 거리가 짧을수록 파티클 개수에 따른 소요 시간의 증가폭이 작으나 감쇠 거리가 길수록 파티클 개수에 따른 소요 시간의 증가폭이 커져 대량의 파티클 라이트를 반영하는데 무리가 따른다.

향후 파티클에 대한 절두체 컬링과 라이트 그리드 계층구조를 만드는 방식으로 실제로 라이팅 계산에 필요한 최소의 파티클을 구한 후 해당 파티클에 대한 라이트 관련 연산을 월드 공간이 아닌 스크린 공간에서 처리하여 상수 시간 알고리즘을 구현하는 것을 목표로 한다.

감사의 글

이 논문은 2018 년도 한국산업기술대학교 학술연구진흥사업 및 2019 년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2017R1C1B200730)

References

[1]F. Lekien and J. Marsden , "Tricubic interpolation in three dimensions", *international journal for numerical methods in engineering,* 2005

[2]Ola Olsson, Markus Billeter & Ulf Assarsson., "Clustered Deferred and Forward Shading", *HPG 2012,* 2012

[3]Ola Olsson , "Tiled and Clustered Forward Shading". , *GPU Pro 4,* 2013

[4]Ola Olsson and Ulf Assarsson ,"Tiled Shading - Preprint", Chalmers University of Technology, *JGT 2011,* 2011

[5] Yuriy O'Donnell , "Tiled Light Trees", *I3D '17: Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games,* 2017

[6]Cem Yuksel, "Lighting grid hierarchy for self-illuminating explosions" , *ACM Transactions on Graphics*, july 2017

[7]Cem Yuksel "Real-Time Rendering with Lighting Grid Hierarchy", *Proceedings of the ACM on Computer Graphics and Interactive Techniques,* june 2019,

[8] Wolfgang Engel. "The light pre-pass renderer: Renderer design for efficient support of multiple lights". *SIGGRAPH Course: Advances in realtime rendering in 3D graphics and games,* 2009.

[9] Scott Kircher and Alan Lawrance. "Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In Sandbox" *'09:Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games, pages 39–45*, New York, NY, USA, 2009.

[10] Andrew Lauritzen. "Deferred rendering for current and future rendering pipelines". *SIGGRAPH Course: Beyond Programmable Shading,* 2010

[11] Matt Swoboda. "Deferred lighting and post processing on playstation 3". *Game Developer Conference*, 2009.

[12] Abdul Bezrati (Insomniac Games), "REAL-TIME LIGHTING VIA LIGHT LINKED LIST", *SIGGRAPH Course : Advances in Real-Time Rendering in Games,* 2014

[13] Emil Persson (Avalanche), Ola Olsson (Chalmers University of Technology), "PRACTICAL CLUSTERED DEFERRED AND FORWARD SHADING", *SIGGRAPH Course : Advances in Real-Time Rendering in Games, 2013*

〈저자소개〉

김 병 진

- 2013-2018 한국산업기술대학교 게임공학부 학사
- 2019-현재 한국산업기술대학교 지식기반기술 · 에너지대학원 미디어융합디자인공학과 석사과정
- 관심분야: Real-time Rendering
- https://orcid.org/0000-0002-2083-8383

이 택 희

- 2001년 서울대학교 컴퓨터공학부 학사
- 2009년 서울대학교 컴퓨터공학부 박사
- 2009-2012년 ㈜삼성전자 무선사업부
- 2017년-현재 한국산업기술대학교 게임공학부 조교수
- 관심분야: 컴퓨터 그래픽스
- https://orcid.org/0000-0003-2985-3229