

# 이미지 기반 파티클 충돌처리 시스템

이용선<sup>O</sup>

이상기

이택희<sup>\*</sup>

한국산업기술대학교

(heebung, lunta94, watersp)@kpu.ac.kr

## Image-based particle collision system

Lee Yongsun<sup>O</sup>

Lee Sang-gi

Lee Taekhee<sup>\*</sup>

KOREA POLYTECHNIC UNIVERSITY

### 요 약

본 논문에서는 많은 수의 파티클 간의 실시간 충돌 알고리즘을 제안한다. 제안된 알고리즘은 3차원 파티클 위치 정보를 2차원 이미지에 투영하고 이를 기반으로 충돌 처리를 수행하는 단계와 실제 렌더링을 수행하는 두 단계를 거친다. 첫 단계에선 파티클 간의 충돌처리를 수행하며 위치를 업데이트한다. 이때 사용되는 데이터는 파티클의 3차원 월드 공간 위치를 저장한 2차원 형태의 텍스처이며 이를 충돌 감지 텍스처라 칭한다. 각 파티클의 충돌 처리를 위해서 해당 파티클의 투영 결과를 기반으로 텍스처 좌표를 계산하고 이 좌표를 기준으로 충돌 감지 텍스처를 샘플링하여 파티클의 3차원 위치 값을 읽어온다. 충돌 감지 텍스처는 파티클의 3차원 정보를 투영하여 얻는 2차원 투영 좌표를 기반으로 업데이트되기 때문에 앞서 얻은 텍스처 좌표의 주변을 주어진 범위 내에서 샘플링하면 주변에 존재하는 파티클들의 위치 정보를 얻을 수 있다. 이 정보를 이용해 현재 파티클과의 충돌처리를 수행하고 처리 후 변화되는 3차원 좌표 형태의 값은 컬러 코드로 변환하여 충돌 감지 텍스처에 업데이트한다. 두 번째 단계는 첫 번째 단계에서 사용된 버퍼를 활용하는 렌더링 패스이다. 해당 버퍼는 충돌 처리가 끝난 3차원 위치 정보를 가지고 있으며 이를 이용하면 기하 셰이더를 통해 빌보드 형태의 쿼드를 생성할 수 있다. 이 쿼드를 파티클 형태로 렌더링하여 최종 결과를 만들어 낸다. 제안된 방법은 파티클당 충돌처리를  $O(n)$  (화면에 존재하는 파티클 개수 \* 샘플링하는 텍셀 개수) 복잡도로 끝마칠 수 있다. 결과적으로 실험을 통해서 평균 71ms 이내에 125,000개의 파티클 충돌 처리를 진행할 수 있었다.

### Abstract

In this paper, a large number of particles proposes a real-time collision algorithm. The proposed algorithm goes through two steps of projecting three-dimensional particle location information into a two-dimensional image and performing collision processing based on it and performing actual rendering. In the first stage, collision processing between particles is performed and the position is updated. The data used at this time is a two-dimensional texture that stores the particle's three-dimensional world space position, and this is called a collision detection texture. For collision processing of each particle, texture coordinates are calculated based on the projection result of the particle, and collision detection texture is sampled based on these coordinates to read the particle 3D position value. Since the collision detection texture is updated based on the 2D projection coordinates obtained by projecting the 3D information of the particle, sampling the periphery of the previously obtained texture coordinates within a given range can obtain the location information of the particles present in the periphery. Using this information, collision processing with the current particle is performed, and the values of 3D coordinates that change after processing are converted into color codes and updated in the collision detection texture. The second stage is a rendering pass that utilizes the buffer used in the first stage. The buffer has 3D location information that has been processed for collision, and if you use it, you can create a billboard-shaped quad through a geometry shader. This quad is rendered in particle form to produce the final result. The proposed method can end collision processing per particle with  $O(n)$  (number of particles in the screen \* number of texels to sample) complexity. As a result, we could process 125,000 particle collisions within an average of 71ms.

**키워드:** 파티클, 충돌처리, 실시간 렌더링, 이미지 기반

**Keywords:** Particle, Collision Processing, Realtime-Rendering, Image-based

## 1. 서론

게임과 같은 실시간 렌더링 환경에서는 높은 품질의 렌더링 된

장면을 보여줘야 하므로 정해진 시간(60hz or 144hz) 안에 렌더링을 끝내기 위해 오래 걸리는 계산을 간략화 하거나 생략한다. 이렇게 생략되는 부분에는 파티클 충돌이 포함된 경우가 존재한다.

시간이 지남에 따라 그래픽 프로세서의 성능이 점점 더 좋아져 짧은 시간 내에 더 많은 계산을 처리할 수 있게 되었다. 그러나 향상된 그래픽 프로세서를 사용해도 물리처리와 같이 계산이 필요한 객체가 많아질수록 계산량이 기하급수적으로 증가하는 문제가 발생하여 실시간으로 처리하기에는 부족하다. 이러한 문제를 해결하기 위해 octal tree, binary space partition, kd tree[1]와 같은 알고리즘을 이용해 3차원의 공간을 분할하여 한번에 처리해야 할 객체의 수를 줄여 계산량을 줄일 수 있는 방식이 제안되었고 사용되어 왔다.

움직임이 정확하게 계산되지 않아도 게임 플레이에 큰 영향을 주지 않는 파티클 이펙트 렌더링의 경우 Screen Space Depth Buffer Based Collision [2,3]과 같은 연구를 통해 정확도가 떨어지는 대신 더 빠르게 계산할 수 있는 방법이 제안되었다. 그러나 해당 방식은 파티클과 지오메트리 오브젝트에 대한 충돌 검사이다. 여전히 파티클 간의 충돌 처리처럼 한 장면에서 계산해야 할 객체가 많은 경우 성능에 대한 부하가 높아 객체의 개수를 줄이거나 제한된 경우에만 충돌계산을 하여 계산량을 줄이는 방법이 사용된다. 또한, 시각적으로 큰 문제가 없는 경우에는 충돌 처리를 진행하지 않는 경우도 존재한다.

본 논문에서는 GPU 기반 많은 수의 파티클간의 실시간 충돌 처리 알고리즘을 제안한다. 해당 연구 프로젝트는 OpenGL로 진행하였다. 파티클의 충돌과 업데이트를 담당하는 단계와 실질적으로 렌더링을 하는 단계를 거친다.

첫 단계에선 파티클간의 충돌처리, 파티클 움직임 업데이트 및 업데이트된 파티클 위치 정보의 2차원 투영과정이 이루어진다. 이때 충돌처리에 사용되는 데이터는 파티클의 3차원 월드 공간의 위치를 컬러 코드로 저장한 2차원 형태의 텍스처이며 이를 충돌 감지 텍스처라 칭한다.

해당 단계의 첫 프레임에서는 렌더링 결과를 저장할 충돌 감지 텍스처가 어떠한 정보도 가지고 있지 않다. 따라서 최초로 수행되는 충돌 처리 단계는 충돌 감지 텍스처의 초기 상태를 만들어내는 과정이 된다.

충돌 감지 텍스처 생성 이후 프레임에서는 파티클의 충돌 처리를 위해 입력된 정점의 2차원 화면 좌표계상의 투영 좌표를 계산한다. 이 좌표는 일반적인 투영행렬이 적용된 -1부터 1 사이의 값을 가진다. 이 값을 이용하여 충돌 처리 텍스처 샘플링 및 업데이트를 위한 텍셀 좌표를 계산한다. 이 과정은 x, y좌표에 1을 더하고 0.5로 나누어 0부터 1 사이의 정규화된 값으로 변환하여 이루어진다.

충돌 감지 텍스처의 텍셀에 저장되어있는 데이터에 접근하기 위해서 정규화 된 값을 기반으로 정수형의 2차원 인덱스를 계산한다. 이 과정은 충돌 감지 텍스처의 가로, 세로 해상도에 앞서 계산된 텍셀 좌표를 곱하여 계산된다. 이렇게 계산된 2차원 인덱스를 Base Texel 인덱스라 칭한다. Base Texel을 이용해

충돌 감지 텍스처에 저장된 3차원 위치 정보를 읽어온다.

충돌 처리를 위해 필요한 주변 파티클의 3차원 위치를 얻기 위해선 위에서 계산된 Base Texel을 중심으로 반복문을 사용하여 주변 텍셀의 값을 샘플링한다. 샘플링된 텍셀은 3차원 위치 정보를 컬러 코드 형태로 가지는데, 이를 기반으로 주변에 존재하는 다른 파티클과의 충돌처리를 수행한다. 충돌 처리는 구를 기반으로 하는 단순화된 충돌처리를 수행한다.

첫 단계의 기하 셰이더에서는 Transform Feedback 기능을 이용해 정점 셰이더에서 업데이트된 파티클의 3차원 위치, 방향, 속도, 충돌시간을 Transform Feedback Buffer Object(TBO)에 업데이트한다. 프래그먼트 셰이더에서 파티클의 3차원 좌표 형태의 값을 컬러 코드로 변환하여 충돌 탐지 텍스처를 업데이트한다.

두번째 단계는 첫번째 단계에서 업데이트된 TBO를 입력 데이터로 사용한다. 기하 셰이더를 통해 입력된 정점을 빌보드 형태의 쿼드로 변형하여 파티클을 그려주는 단계를 거친다.

제안된 방법의 충돌 처리는 정점 셰이더에서 이루어진다. 정점 셰이더는 파티클의 개수만큼 실행되며 충돌 처리는 미리 설정한 정수 값의 충돌 감지 범위만큼 이중 반복문을 통해 충돌을 계산한다. 그러나 충돌 감지 범위는 파티클에 개수에 따른 변화가 없기 때문에 파티클당 충돌처리를  $O(n)$  (화면에 존재하는 파티클 개수 \* 샘플링하는 텍셀 개수) 복잡도로 끝마칠 수 있다.

실험을 통해서 평균 71ms 이내에 125,000개의 파티클 충돌 처리를 진행할 수 있었다.

본 논문의 구성은 다음과 같다. 제2절에서는 관련연구에 대해 설명한다. 제3절에서는 충돌 처리 시스템에 대해 설명하고, 제4절에서는 제안한 알고리즘을 이용한 파티클 개수와 파티클 밀도 차이에 따른 성능 측정 데이터를 보이며, 제5절에서는 결론 및 문제점과 향후 연구방향을 설명한다.

## 2. 관련연구

제한된 그래픽 프로세서의 성능으로 인해 실시간 그래픽스 분야에서 계산량을 줄이기 위한 연구는 활발하게 진행되어 왔다. 먼저, 화면에 보이는 오브젝트만 렌더링 하는 기법으로 뷰잉 프러스텀(Viewing Frustum) 외부에 있는 오브젝트를 그리지 않는 프러스텀 컬링[4]과 다른 오브젝트에 가려져 카메라에 보이지 않게 된 오브젝트의 렌더링을 버리는 오클루전 컬링[5]등이 연구되었다. 하지만 오브젝트의 개수가 많아질 경우 [4], [5] 알고리즘을 실행하는 것 자체가 계산량이 많아 문제가 되었기 때문에 한 번에 계산할 객체들의 개수를 줄이기 위해 공간 분할 알고리즘들이 등장하기 시작했다.

Quad tree에서 삼차원 확장된 octal tree, 재귀적으로 유클리

드 공간을 초평면 상의 블록 집합으로 분할하는 binary space partition(BSP Tree), kd tree [1], obb tree [6] 등이 연구되었고 이러한 공간분할 방법들의 성능 비교에 대한 연구 또한 같이 진행되었다. [7]

그러나 위의 연구들은 게임에서의 파티클 시스템 시뮬레이션에 사용하기에는 파티클 시스템이 순간적으로 생성하는 많은 파티클의 수를 실시간으로 계산하기 어려움이 있었고 이것은 성능을 우선으로 하는 게임에서는 사용할 수 없었다.

이에 정확한 충돌계산보다 성능 우선순위가 높은 경우 사용될 수 있는 Screen Space Depth Buffer Based Collision [2,3] 연구가 진행되었다. 해당 연구는 상용적으로 사용되는 언리얼 엔진 4에서도 GPU 파티클과 지오메트리 오브젝트를 충돌하는데 사용할 정도로 효율적인 모습을 보인다. [8] 그러나 해당 연구는 정확도가 기존의 공간분할 충돌방식보다는 떨어지며, 파티클 간의 충돌을 처리하지 못한다는 단점이 있다. [9]

본 논문에서 제안하는 시스템은 많은 수의 파티클 간의 실시간 충돌 알고리즘을 제안한다. 파티클의 3차원 위치 정보를 가지고 있는 충돌 탐지 텍스처를 이용하여 파티클간 충돌을 처리하는 방식을 제안한다.

다음 절에서 어떠한 방법을 통해 충돌 탐지 텍스처를 생성하고 이것을 이용하여 어떻게 충돌을 처리하는지 설명한다.

### 3. 이미지 기반 충돌 시스템 구성

본 논문에서 제안하는 이미지 기반 파티클 충돌처리 시스템은 2단계 알고리즘으로 구성되어 있다. 알고리즘 설명 이전에 본 논문에 사용된 OpenGL 기능인 Transform Feedback을 설명하고 파티클 충돌처리 시스템에 대해 설명한다.

#### 3.1 Transform Feedback

본 논문에서는 CPU의 간섭을 줄이고 GPU에서 연산을 처리하기 위해서 OpenGL의 기능인 Transform Feedback을 사용한다.

Transform Feedback은 그래픽스 파이프라인 Vertex Processing 단계에서 처리되는 기능이다. Vertex Processing 단계는 정점 셰이더(Vertex Shader) 및 기하 셰이더 단계를 포함하는데, 정점 셰이더 또는 기하 셰이더 단계에서 도출된 결과값을 TBO에 반영하여 업데이트할 수 있다. 이 전용 버퍼는 GPU 메모리에 할당되므로 CPU-GPU간 데이터 이동이 없어 매우 빠르게 GPU메모리 내용을 업데이트할 수 있다.

업데이트 된 버퍼는 다른 Draw 단계에서 정점 데이터 입력으로 사용할 수 있다. 이를 위해 두개의 TBO를 생성하며 각 TBO는 업데이트를 위한 Buffer와 현재 사용하기 위한 Buffer로 각자의 역할을 가진다. 이로 인해 읽기 및 쓰기 과정에서 나타날 수

있는 충돌을 피할 수 있으며 매 프레임마다 서로 역할을 바꾸어 최신 정보를 기반으로 계산이 이루어 질 수 있도록 한다.

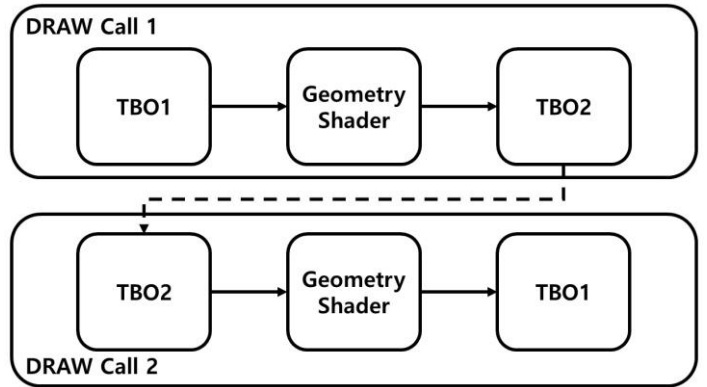


Figure 1 Transform Feedback Buffer Object Update

Transform Feedback은 병목 현상을 일으킬 수 있는 CPU-GPU간 메모리 이동을 없앨 수 있기 때문에 좌표가 매 프레임 변경되는 충돌 처리 과정에 유용하게 사용될 수 있다.

#### 3.2 파티클 충돌 및 업데이트 단계

본 논문에서 소개하는 파티클 간 충돌처리 알고리즘은 파티클의 충돌과 업데이트를 담당하는 단계와 실질적으로 렌더링을 하는 단계를 거친다. 각 단계는 하나의 그래픽스 파이프 라인 동작을 기준으로 나누어 졌기 때문에 단계별로 한번의 Draw Call을 수행한다.

첫 번째 단계에선 충돌처리, 파티클 움직임 업데이트 및 업데이트된 파티클 위치의 2차원 투영과정이 이루어진다.

첫 프레임에는 렌더링 결과를 저장할 충돌 탐지 텍스처가 어떠한 정보도 가지고 있지 않다. 따라서 최초로 수행되는 충돌 처리 과정은 충돌 감지 텍스처의 초기 상태를 만들어 내는 과정이 된다. 이후 진행되는 프레임에선 업데이트된 충돌 탐지 텍스처를 기반으로 충돌 처리가 이루어진다. Figure 2는 충돌 감지 텍스처가 생성된 이후의 충돌처리 단계를 나타낸다

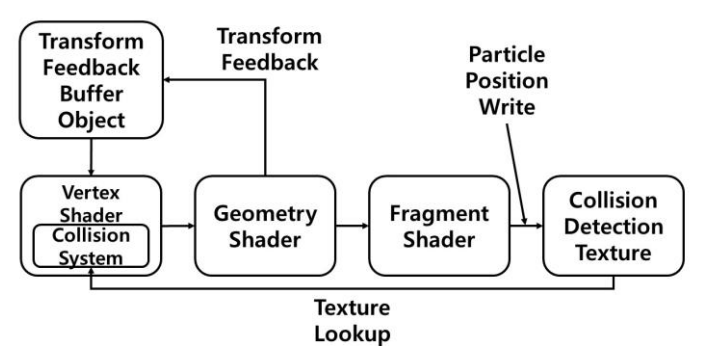


Figure 2 Collision Pass

충돌 감지 텍스처를 사용한 파티클의 충돌 처리를 위해서 현

재 그래픽스 파이프라인의 입력된 정점의 2차원 화면 좌표계상의 투영 좌표를 계산한다. 이 좌표는 일반적인 투영행렬이 적용된 -1부터 1사이의 값을 가진다. 이 값을 이용하여 충돌 처리 텍스처 샘플링 및 업데이트를 위한 텍셀 좌표를 계산한다. 이 과정은  $x, y$ 좌표에 1을 더하고 0.5로 나누어 0부터 1사이의 정규화된 값으로 변환하여 이루어진다.

충돌 감지 텍스처는 이미지 표현을 위해 사용하지 않기 때문에 일반적으로 텍스처 샘플링시 수행되는 보간이 일어나서는 안되며 정확한 정수 형태의 2차원 인덱스를 기반으로 값을 참조한다. 이에 충돌 감지 텍스처의 텍셀에 저장되어 있는 데이터에 접근하기 위해서 정규화된 값을 기반으로 정수형의 2차원 인덱스를 계산한다. 이 과정은 충돌 감지 텍스처의 가로, 세로 해상도에 앞서 계산된 텍셀 좌표를 곱하여 계산된다. 이렇게 계산된 2차원 인덱스를 Base Texel 인덱스라 칭한다. 충돌 처리를 위해 필요한 주변 파티클의 3차원 정보 값을 얻기 위해서 충돌 감지 텍스처의 텍셀을 Base Texel을 중심으로 반복문을 사용하여 주변 텍셀들의 값을 샘플링한다.

Base Texel 값을 사용하여 충돌 감지 텍스처에 저장된 현재 파티클의 3차원 위치 정보를 읽어 오기 위해 GLSL에서 제공하는 `texelFetch(Sampler, Texel, Lod)` 함수를 사용한다. 해당 함수는 정수 형태의 2차원 인덱스를 통해 텍스처의 텍셀에 저장되어 있는 데이터에 접근할 수 있는 기능을 가진 함수이다. 여기서 Sampler는 텍스처를 의미하고 Texel은 2차원 인덱스를 의미하며 Lod는 텍스처에 저장되어 있는 Mipmap 레벨을 의미한다.

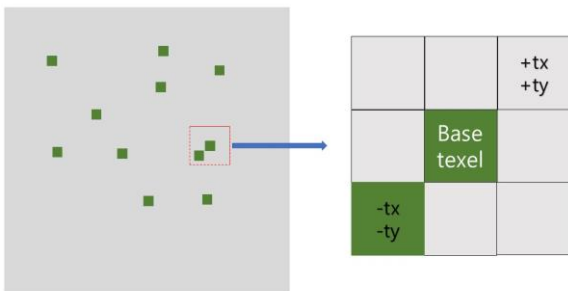


Figure 3 Collision Detection Range

Base Texel 주변 범위의 텍셀을 순환하기 위해 이중 반복문을 사용한다. 탐색 범위를 정하기 위해 사전에 설정해둔 충돌 감지 범위 값을 `texel_collide_range`로 칭한다.

의사코드의 9, 10번 라인에서 반복문의 조건 변수를  $tx$ 와  $ty$ 로 칭하였다. 두 조건변수는 `-texel_collide_range`의 값을 초기 값으로 가진다. 두 조건변수는 초기값으로부터 값을 1씩 증가시켜 `texel_collide_range`까지의 범위를 순환한다.

Figure 3는 `texel_collide_range`를 1로 설정했을 때의 충돌 감지 범위를 보여주는 그림이다.

11번 라인에서 기존의 2차원 인덱스인 Base Texel에 조건 변수  $tx, ty$ 를 더하여 새로운 2차원 인덱스 Texel을 계산한다. 이때 라인 12, 13에서 볼 수 있듯이  $tx$ 와  $ty$ 의 값이 0을 가지는 경우 `continue` 함수를 통해 if문 아래의 충돌 처리를 하지 않고 반복문을 순환한다.

계산된 Texel이 가지는 3차원 위치 정보를 얻기 위해 라인 14번과 같이 `texelFetch` 함수를 통해 주변부 파티클의 3차원 좌표를 계산한다. 계산된 3차원 위치 좌표와 현재 파이프라인에 입력된 정점의 3차원 위치 좌표의 직선거리를 GLSL의 `length(vector)` 함수를 통해 계산한다. `length` 함수는 인자로 넘어온 벡터의 요소를 각각 제곱하여 더한 뒤 루트를 씌워 벡터의 길이를 얻는 함수이다. 이때 얻어지는 직선거리를 `len`이라 칭한다.

의사코드의 6번 라인에서 설정한 `particle_min_length`는 한번에 여러 파티클과의 충돌을 방지하고 가장 가까운 파티클과의 충돌처리를 위해 최소거리를 저장하는데 사용된다.

만약 계산된 `len`의 값이 의사코드 7번에서 사전에 설정한 충돌 기준 값보다 작고 `particle_min_length`의 값보다 작은 경우 충돌로 판정한다. 충돌로 판정된 경우 `particle_min_length`의 값을 현재 계산된 `len`의 값으로 업데이트한다.

의사코드 22번에서 볼 수 있듯이 파티클의 충돌 이후 방향 계산을 위해 충돌한 파티클의 3차원 위치 정보가 필요하다. 18번 라인과 같이 충돌로 판정된 경우 충돌된 파티클의 3차원 위치 정보를 저장한다.

반복문의 종료 이후 20번 라인과 같이 충돌 판정된 파티클의 처리를 진행한다. 속도를 20% 감소시키고, 방향은 충돌한 두 파티클의 3차원 위치 정보의 뺄셈을 통해 얻어지는 벡터를 정규화한 값을 사용한다. 변경된 3차원 위치 정보는 계산된 속도와 방향을 곱하여 이전의 3차원 위치 정보에 더하여 업데이트한다. 파티클의 시각적 표시 및 지속적인 충돌 방지를 위해 충돌시간을 업데이트한다.

충돌하지 않은 파티클들은 중력의 영향을 받아 업데이트 되어 위치 정보가 변경된다. 이 과정을 의사코드로 표현하면 다음과 같다.

#### ParticleCollisionPass

```

1  loop particle in scene.particle
2      vec4 gl_Position = u_ProjView * vec4(a_Position,1)
3      vec2 TexCoord = (gl_Position.xy + vec2(1)) * 0.5
4      ivec2 texel_base = ivec2(TexCoord.x * u_ClientWidth, TexCoord.y *
u_ClientHeight)
5      vec4 curr_pos = texelFetch(u_CDTxture, texel_base, 0)
6      float particle_min_length = 100000
7      float g_collision_range = 0.01

```

```

8   if a_CollideTime < 0
9       loop ty in -texel_collide_range <= ty <= texel_collide_range
10      loop tx in -texel_collide_range <= tx <= texel_collide_range
11          ivec2 texel = texel_base + ivec2(tx, ty)
12          if tx == 0 && ty == 0
13              continue
14          vec4 particle_pos = texelFetch(u_CDTexture, texel, 0)
15          float len = length(curr_pos.xyz - particle_pos.xyz)
16          if g_collision_range > len && len < particle_min_length
17              particle_min_length = len
18              particle_collide_pos = particle_pos
19              particle_collision = true
20  if particle_collision == true
21      f_Speed = a_Speed * 0.2
22      v_Dir = normalize(curr_pos.xyz - particle_collide_pos.xyz)
23      f_CollideTime = g_collision_time
24      v_Position = a_Position + (v_Dir * f_Speed)
25  else
26      vec3 temp = (a_Dir * a_Speed) + (g_gravity * g_tick)
27      v_Dir = normalize(temp)
28      f_Speed = length(temp)
29      f_CollideTime = a_CollideTime - g_tick
30      v_Position = a_Position + (v_Dir * f_Speed) * g_tick

```

이후 기하 셰이더에서 파티클의 3차원 위치, 방향, 속도 및 충돌시간을 TBO에 업데이트하게 된다. 프래그먼트 셰이더를 통해 파티클의 3차원 위치 정보는 쿼리코드로 변환되어 충돌 감지 텍스처에 업데이트 된다.

### 3.3 파티클 렌더링 단계

렌더링 단계는 Figure 4와 같은 구조로 이루어져 있다.

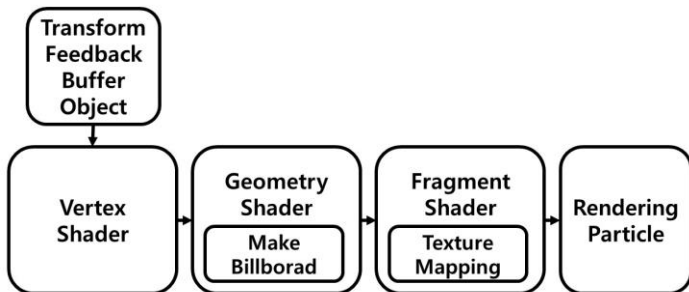


Figure 4 Rendering Pass

렌더링 단계는 충돌 처리 단계를 통해 업데이트된 TBO를 렌더링 단계의 정점 셰이더의 입력 데이터로 활용한다.

기하 셰이더에서는 입력된 하나의 정점을 빌보드 형태로 변형시키는 작업을 진행한다. 입력 받은 정점을 기준으로 카메라를 바라보는 4개의 정점을 생성하여 아래의 Figure5와 같은 형태로 렌더링 된다.

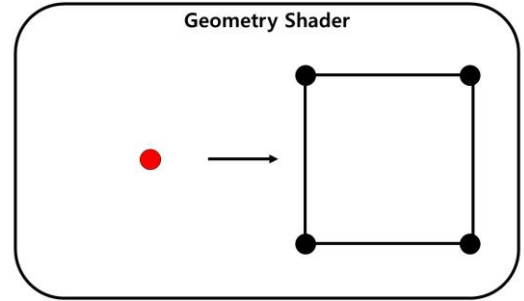


Figure 5 Billboard

## 4. 성능 측정 결과

본 논문에서 제안한 파티클 충돌 시스템은 OpenGL을 사용하여 NVIDIA GeForce GTX 1060 그래픽 카드가 설치된 PC에서 구현하였다. 그래픽스 파이프라인을 프로그래밍 하기 위해서 GLSL Shader Language를 이용하였다. 측정을 위해 생성되는 충돌 탐지 텍스처의 해상도는 500x500의 해상도를 설정하였다.

본 논문에서 제안한 파티클 충돌시스템의 성능 측정을 위해 두 가지의 모델 데이터를 사용하였다. 모델 데이터는 다음과 같은 구조를 가지고 있다. Bunny 모델은 30,338개의 폴리곤, 정점이 15,258개이며, Dragon 모델은 21,782개의 폴리곤과 10,783개의 정점을 가지고 있다.

최소 5만개에서 20만개의 파티클까지 4단계의 측정을 하였다. 또한, 파티클의 밀도에 따른 충돌 결과를 측정하기 위해 두 가지의 밀도 타입을 두어 측정하였다. 밀도가 높게 분포된 타입을 A 타입으로 칭하고 A 타입보다는 밀도가 낮게 분포된 타입을 B 타입으로 칭하였다.

Table 1 Experimental results(ms) according to the number of particles

Mesh	Density Type	Particle Number			
		50,000	100,000	150,000	200,000
Bunny	A	19	40	75	96
	B	26	54	97	129
Dragon	A	20	51	75	100
	B	28	59	114	160



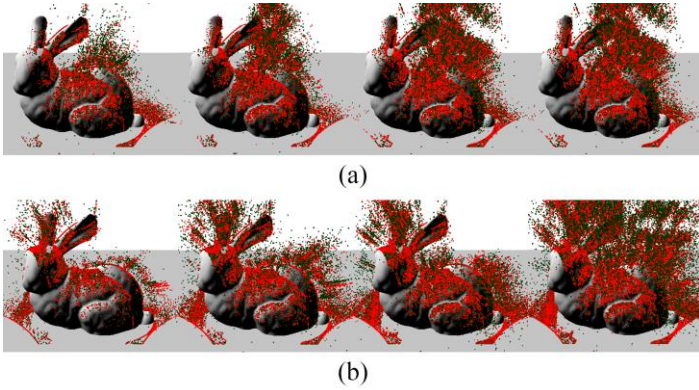


Figure 6 Bunny Mesh Collision Result

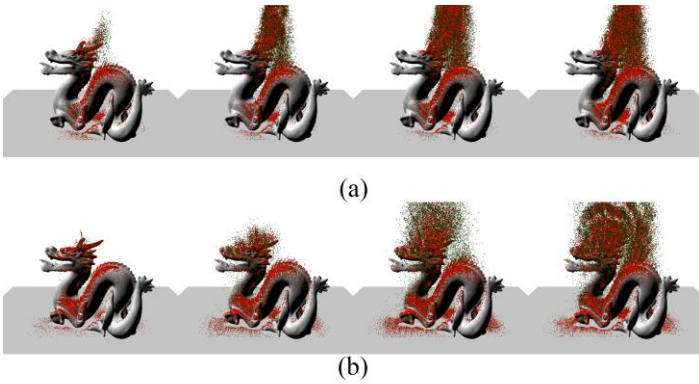


Figure 7 Dragon Mesh Collision Result

Table 1의 결과를 확인하면 파티클의 개수에 따라 선형적으로 ms가 증가하는 것을 확인할 수 있다.

Figure 6(a)는 Table 1에서 얻은 Bunny 모델의 밀도 A타입의 결과를 캡처한 것이며 Figure 6(b)는 밀도 B타입의 렌더링 결과를 캡처한 그림이다.

Figure 7(a)는 Dragon 모델의 밀도 A타입의 결과를 나타낸 사진이며 Figure 7(b)는 밀도 B타입의 렌더링 결과를 나타낸다.

그러나 두 모델 모두 A타입에서 조금 더 효율적인 모습을 관찰할 수 있다. 이러한 현상이 나타나는 이유는 연구 프로젝트의 파티클 충돌 설정과 관련이 있다.

Figure 6, 7의 사진들은 전반적으로 충돌된 파티클들이 산처럼 쌓이는 현상이 나타난다. 이 현상은 현재 실험을 위한 프로젝트에서는 파티클의 라이프타임을 적용하지 않았기 때문에 파티클이 시간이 지나도 사라지지 않는 것과 충돌된 파티클은 지속적으로 속도가 20%씩 감소하게 설정되어 결과적으로 속도가 0이 되어 제자리에 멈추게 된다.

이 두 요인의 작용으로 인해 파티클의 속도가 0이 되어도 지속적인 충돌이 일어나며 점차 쌓여가는 모습이 관찰된다. 이 현상으로 인해 파티클이 더 넓은 장소로 떨어지는 밀도 B타입의 밀도 A타입의 성능보다 낮은 결과를 보인다.

본 논문에서 제안된 방식을 통해 NVIDIA GeForce GTX 1060 그래픽 프로세서를 사용하는 컴퓨터에서 테스트한 결과 평균적으로 12만5천개의 파티클을 처리하는데 71ms가 소요되었다.

## 5. 결론 및 문제점과 향후 연구.

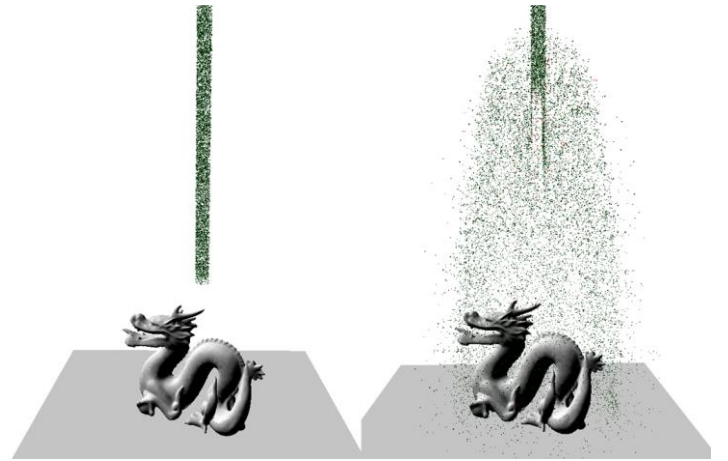


Figure 8 Collision Pass Off/On

Figure 8의 왼쪽 사진은 본 논문의 첫번째 단계인 충돌 처리 및 업데이트 단계를 실행하지 않고 렌더링 단계만 실행한 상태의 그림이다. Figure 8의 오른쪽 사진은 충돌 및 업데이트를 실행한 결과를 나타낸다. 해당 사진은 밀도 높게 뭉쳐서 생성된 파티클이 본 논문에서 제안된 알고리즘을 통해 충돌로 인해 충돌하며 내려오는 모습을 보여준다.

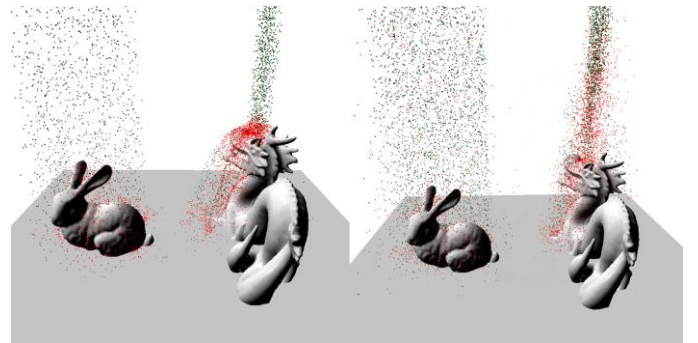


Figure 9 Particle Collision Result

본 논문에서 제안한 이미지 기반 파티클 충돌처리 시스템을 구현한 후 테스트한 결과 일부 경우는 정상적으로 작동하였으나 몇 가지 문제점이 발생하였다.

첫 번째로 카메라와 파티클의 거리에 따라 충돌 결과가 달라지는 문제가 발견되었다.

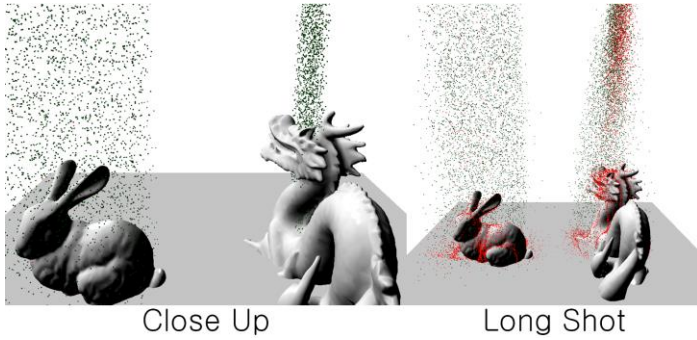


Figure 10 Camera Position Test

Figure 10에서 카메라가 Close Up 상태일 경우 충돌 감지 텍스처에 갱신되는 이웃한 파티클간의 텍셀 거리가 늘어나는 현상이 이유이다. 제한된 알고리즘은 정해진 수만큼 이중 반복문을 통해 주변 텍셀에 저장된 다른 파티클들의 위치 정보를 샘플링한다. 만약 파티클간의 텍셀거리가 늘어나게 되면 충돌 감지 범위를 벗어나게 되어 충돌 처리에서 제외될 수 있다. Long Shot의 경우 Close Up 상태와 반대로 파티클간의 텍셀거리가 줄어들어 충돌 감지 범위 내에 많은 양의 파티클이 샘플링되어 충돌이 일어날 수 있다.

두 번째로 그래픽스 파이프라인의 정점 셰이더에서 다른 파티클들의 이전 프레임 정보를 가져와 충돌 처리를 하는 특성에 따른 문제가 발생하였다. 이는 현재 그래픽스 파이프라인에 입력된 파티클이 충돌인 경우 충돌된 파티클과의 계산을 통해 방향을 변경하지만 충돌된 상대 파티클의 방향은 변경할 수 없기 때문에 서로 밀어내는 충돌을 할 수 없다.

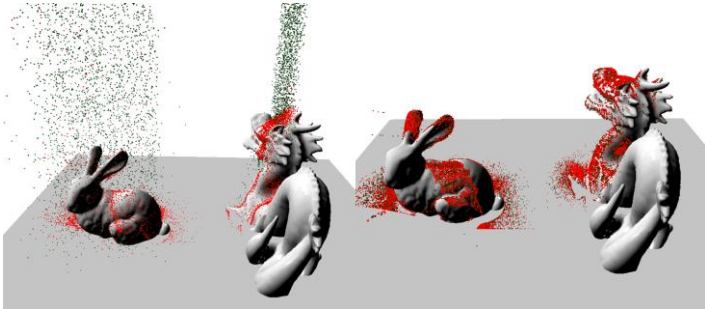


Figure 11 Tested the particles after reducing the particle speed.

Figure 11와 같이 인위적으로 파티클의 속도를 줄여서 파티클이 멀리 튕기지 않도록 설정하였다. Figure 11의 왼쪽 사진에서는 Figure 9에서 보이는 결과와 다르게 파티클이 오브젝트의 표면을 따라 흐른다. 오른쪽 사진에서는 모든 파티클이 떨어진 후 다른 파티클과 겹쳐진 파티클들이 지속적으로 충돌하여 오브젝트 표면에 남아있는 것을 관찰했다.

이 문제가 심화될 경우 아래의 Figure 12과 같은 문제가 발생할 수 있다.

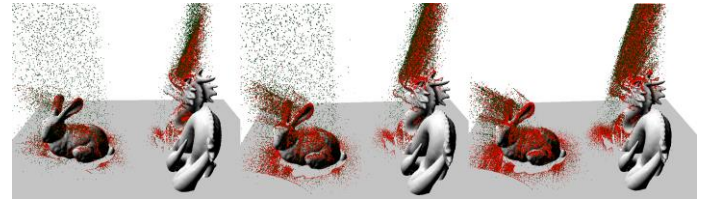


Figure 12 The frequency of collisions per second increased 10 times.

Figure 12에는 파티클의 충돌 빈도를 10배 증가시킨 결과가 표현된 그림이다.

파티클이 오브젝트 표면에서 지속적으로 충돌하여 속도가 감소하여 제자리에 멈추게 된다. 이후 떨어지는 파티클이 멈춘 파티클 근처에서 지속적으로 충돌하게 되어 파티클이 점점 쌓여 뿔처럼 자라게 된다.

세번째 문제로는 다른 파티클이나 오브젝트에 가려진 파티클이 충돌하지 않는 문제가 있다.

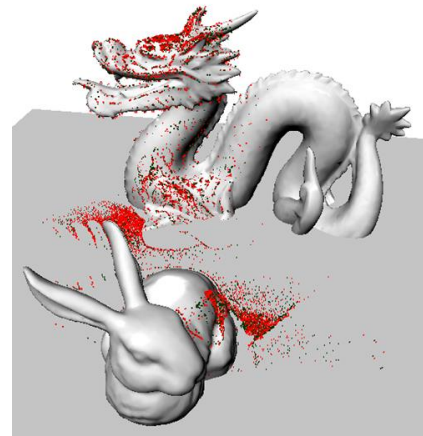
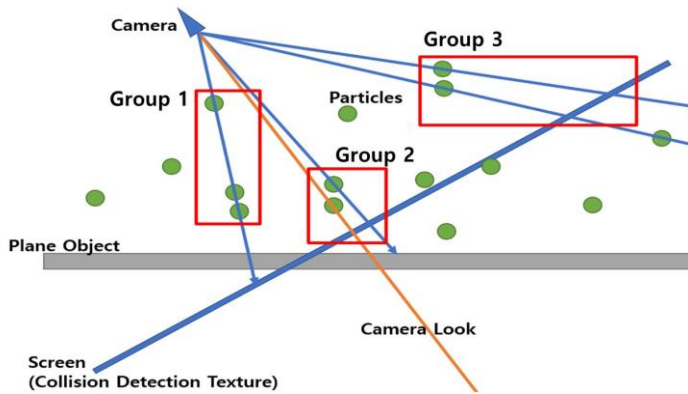


Figure 13 Move the camera

Figure 13에서는 Figure 12에서 뿔처럼 자란 파티클들이 카메라의 초점이 바뀌므로 인해 중력의 영향을 받아 다시 떨어진다. 지속적으로 충돌하고 있는 파티클들이 오브젝트에 가리는 경우 마찬가지로 충돌을 멈추고 떨어지기 시작한다. 다른 파티클에 가려지는 경우 Figure 12과 같이 정지되어있는 파티클 뒤로 지나가는 파티클들은 충돌이 일어나지 않는다.

마지막으로 네번째 문제로는 3차원 파티클의 위치와 충돌 감지 텍스처에 투영되는 텍셀의 위치의 차이에서 생기는 문제점이다.



**Figure 14 Image-based Collision Processing Error**

Figure 14는 충돌 감지 텍스처에 투영되는 파티클들의 위치를 나타내고 있고, 그에 따른 문제점에 대한 내용을 표현한 그림이다.

Group1의 파티클들은 3차원 월드 공간에서는 두개의 파티클은 충돌할 정도의 거리로 가까이 있지만 하나의 파티클은 비교적 먼 거리에 있는 것을 볼 수 있다. 정상적인 충돌 시스템의 경우 두개의 파티클들의 충돌을 기대할 수 있다. 그러나 파티클들이 충돌 감지 텍스처에 투영되는 경우 카메라에 가장 가까운 파티클의 위치가 충돌 감지 텍스처에 기록되기 때문에 뒤에 있는 충돌 가능한 파티클들의 충돌이 일어나지 않는다.

Group2와 Group3의 파티클들은 3차원 월드 공간에서 충돌할 정도로 가까운 거리에 있는 것을 확인할 수 있다. Group2의 파티클들의 경우 카메라의 중심과 가까운 곳에 위치하여 2차원 투영시에도 두 텍셀의 거리가 가까운 위치에 투영된다. 이에 비해 Group3의 파티클들은 카메라의 중심에서 먼 위치에 있어 2차원 투영시 두 텍셀의 거리가 멀어 충돌이 일어나지 않을 수 있다.

또한, Plane Object 아래에서 충돌 감지 텍스처에 기록된 파티클들은 실제로 충돌 연산이 진행되지만 Plane Object에 의해 보이지 않게 된다.

앞서 설명한 문제들은 충돌범위를 늘리거나 파티클 속도를 증가시키고 충돌빈도를 줄이는 방법으로 증상을 완화할 수는 있지만 근본적인 해결법은 될 수 없다. 따라서 이러한 문제점을 근본적으로 해결할 수 있는 방법을 향후 연구할 예정이다.

## 감사의 글

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

## References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld and Mark

Overmars, *Computational Geometry Algorithms and Applications 3rd ed.*, Springer-Verlag Berlin Heidelberg, p318, pp 261-262, pp 99-105, 2008.

[2] Edd Biddulph. *Screenspace particle physics*, 2013.

[3] Gareth Thomas. *Advanced visual effects with directx 11 : Compute-nased gpu particle system.*, 2014.

[4] Ulf Assarsson, Tomas Möller, *Optimized View Fustum Culling Algorithms*, 1999.

[5] Satyan Coorg, Seth Teller, *Real-Time Occlusion Culling for Models with Large Occluders*, 1997.

[6] S.Gottschalk, M. C. Lin, D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*, 1996.

[7] Richard J. Anderson, *Tree Data Structiures for N-Body Simulation*, 1997.

[8] Unreal Engine, *GPU Particles with Scene Depth Collision* [[http://api.unrealengine.com/KOR/Resources/ContentExamples/EffectsGallery/1\\_E/index.html](http://api.unrealengine.com/KOR/Resources/ContentExamples/EffectsGallery/1_E/index.html)]

[9] Stevan Miloradovic, Nils Kavemark, *Particle System A Comparison Between Octree-based and Screen Space Particle Collision.*, 2018.