Internship Project: Particle System

Subin Han

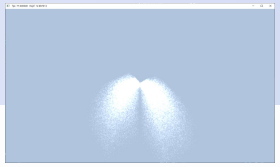
Particle System

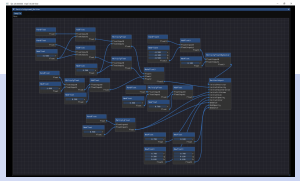
- 선정 동기
  - 내 의지와 무관하게 선택된 문제를 해결하는 것이 가장 도전적
- 소개
  - 유저가 원하는 효과를 만들기 위해 유연한 Control을 제공하는 파티클 시스템
  - 필자는 예술가가 아니므로, 심미적인 효과를 구현하는 것보다는 내부의 기술적인 구현에 집중

History

- 1~4주차: 기반 시스템 구축
  - 기본적인 파티클 관리 및 렌더링 파이프라인
  - UI
  - 동적인 파티클 제어
  - 불 렌더링 등 실질적인 파티클 효과 시연
- 5~8주차: 보다 특수한 기술 구현 및 최적화
  - 리본 렌더링
  - 리본 텍스처 매핑
  - 최적화
  - Bounding k-gon

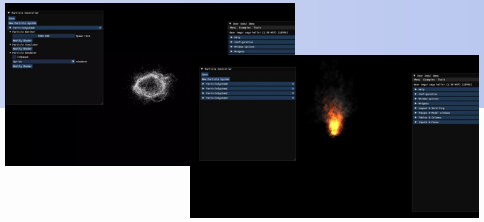
Basic Particle System





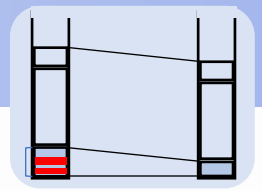
UI & Particle Control

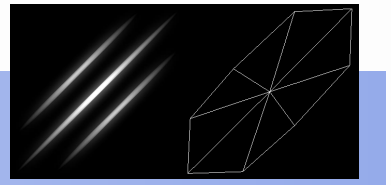
Forces & Fire Rendering



Ribbon Rendering

Optimization

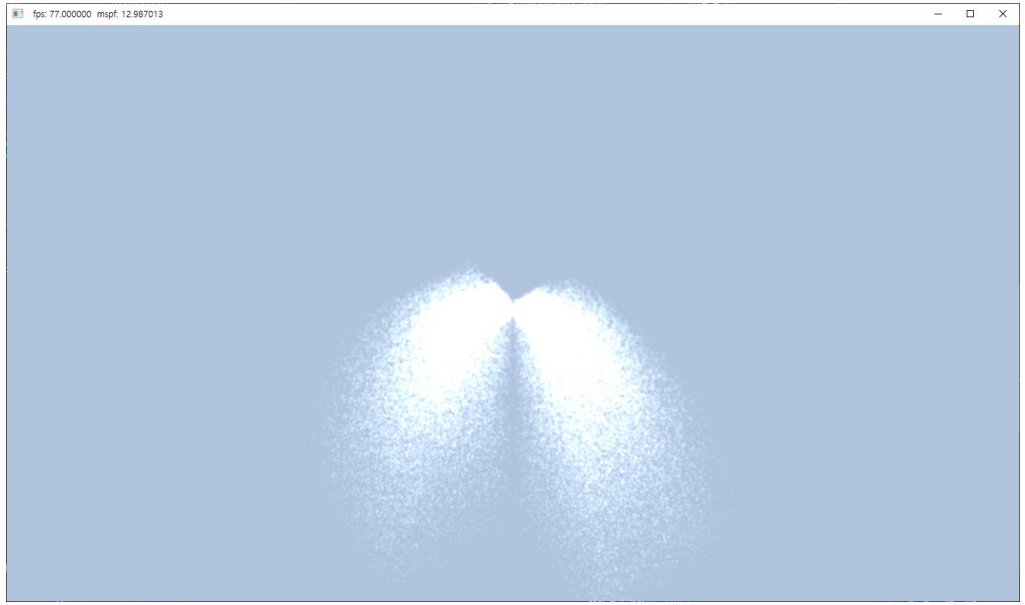


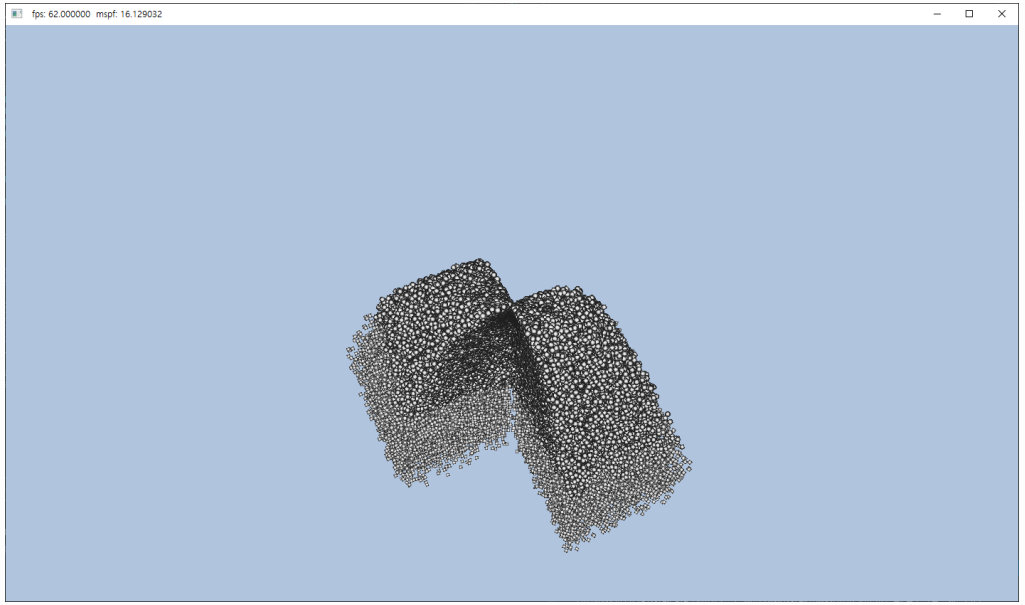


Bounding k-gon

Week 1~2: Basic Particle System

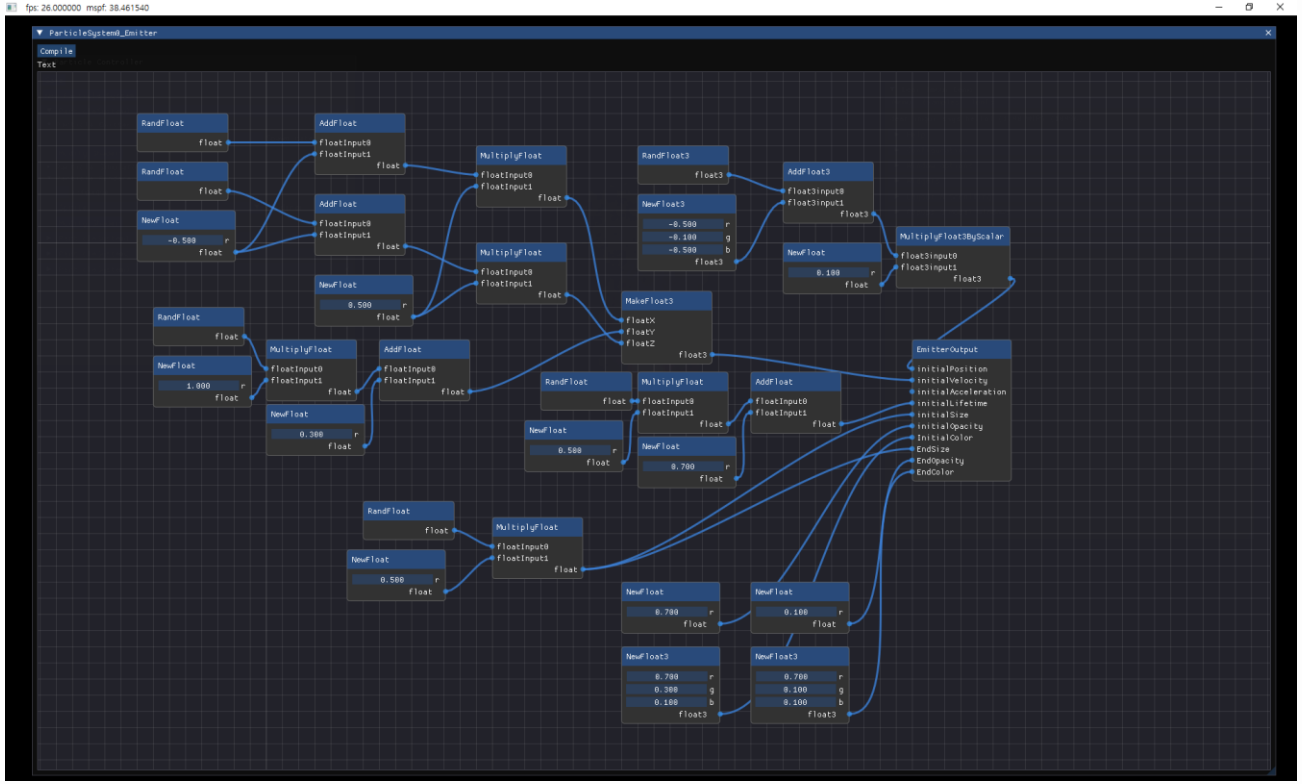
- 주제 선정 및 D3D12 개발 환경 구성
- 기본적인 파티클 관리, 렌더링 파이프라인 시스템 구축





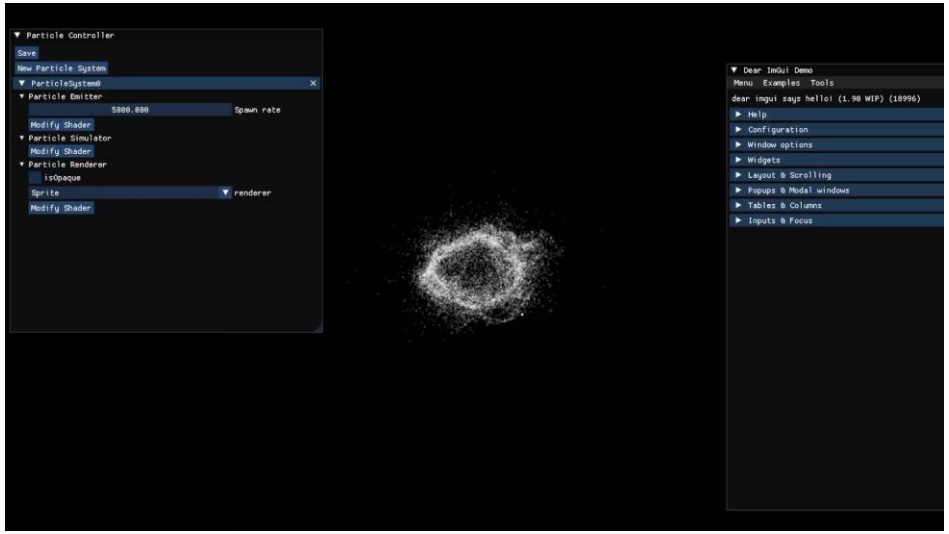
Week 3: UI & Particle Control

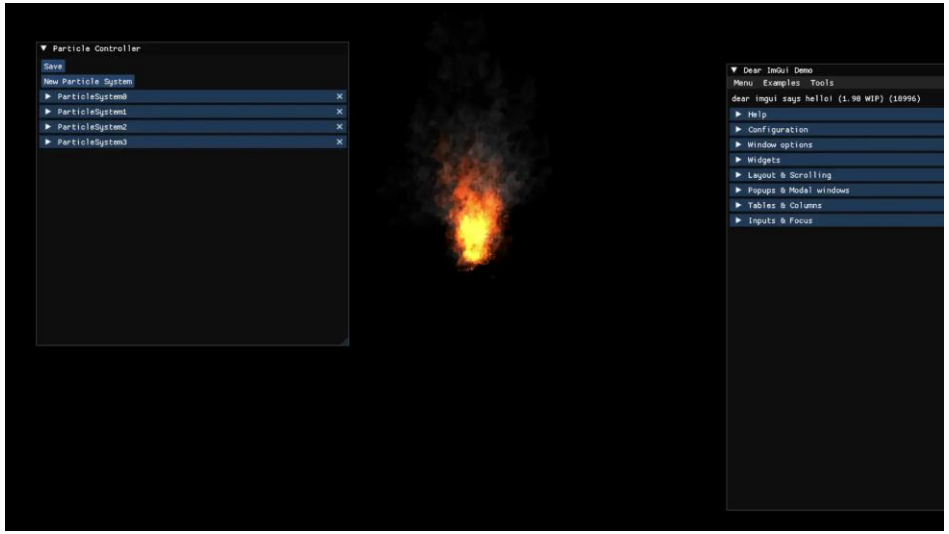
- imgui를 이용한 UI 도입
- 그래프 기반 에디터로 Emission, Simulation, Rendering 쉐이더를 동적으로 수정할 수 있음



Week 4: Forces & Fire Rendering

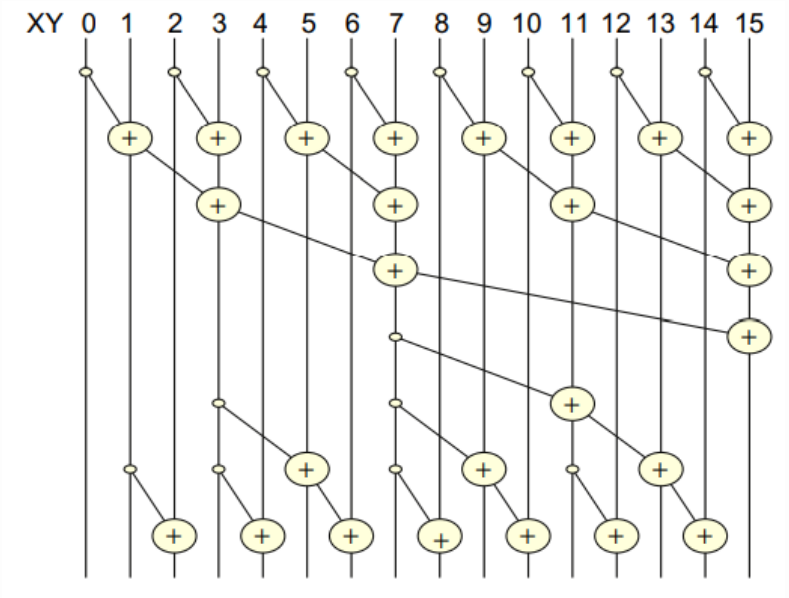
- Drag, Vortex, Point Attraction, Curl Noise 등의 힘을 추가해 기본적인 시뮬레이션이 가능하도록 확장
- 이를 이용한 불 렌더링 시연

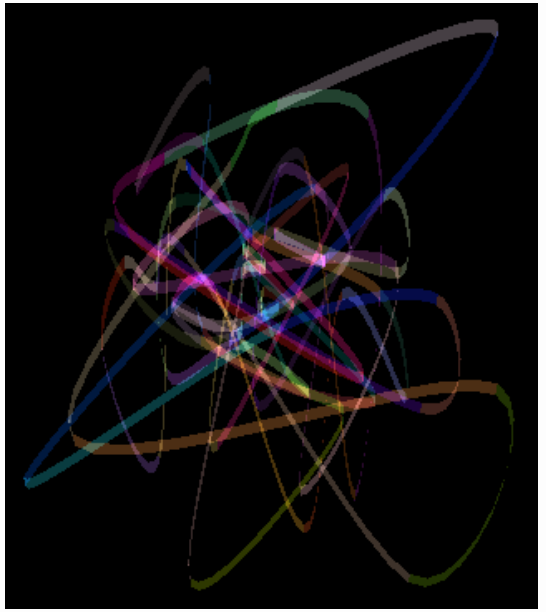


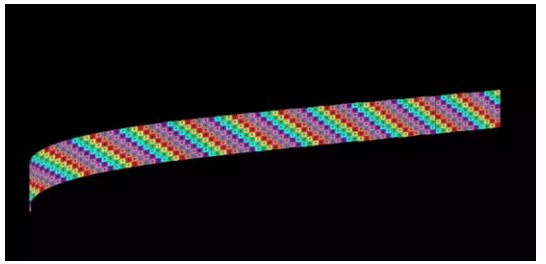


Week 5~6: Ribbon Rendering & Refactoring

- Ribbon Rendering 구현
  - 다양한 Texture Mapping 기법에 대해서 공부
    - Segment based
    - Stretched
    - Distance based
- 코드 리팩토링
- 성능 측정 환경 구성

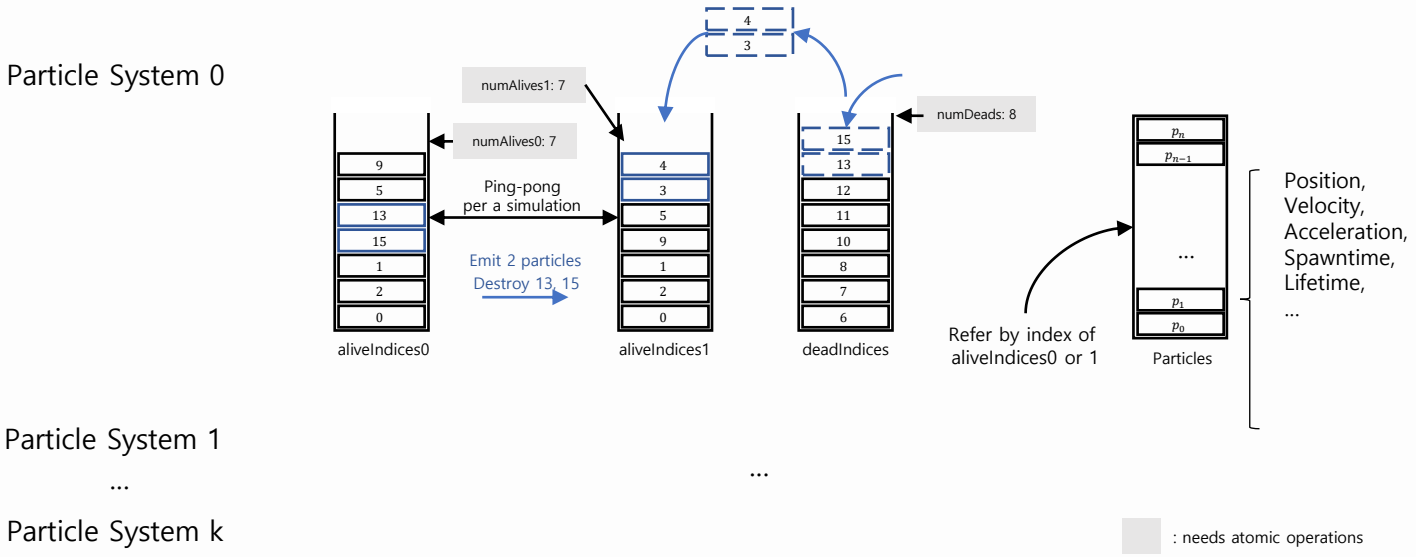






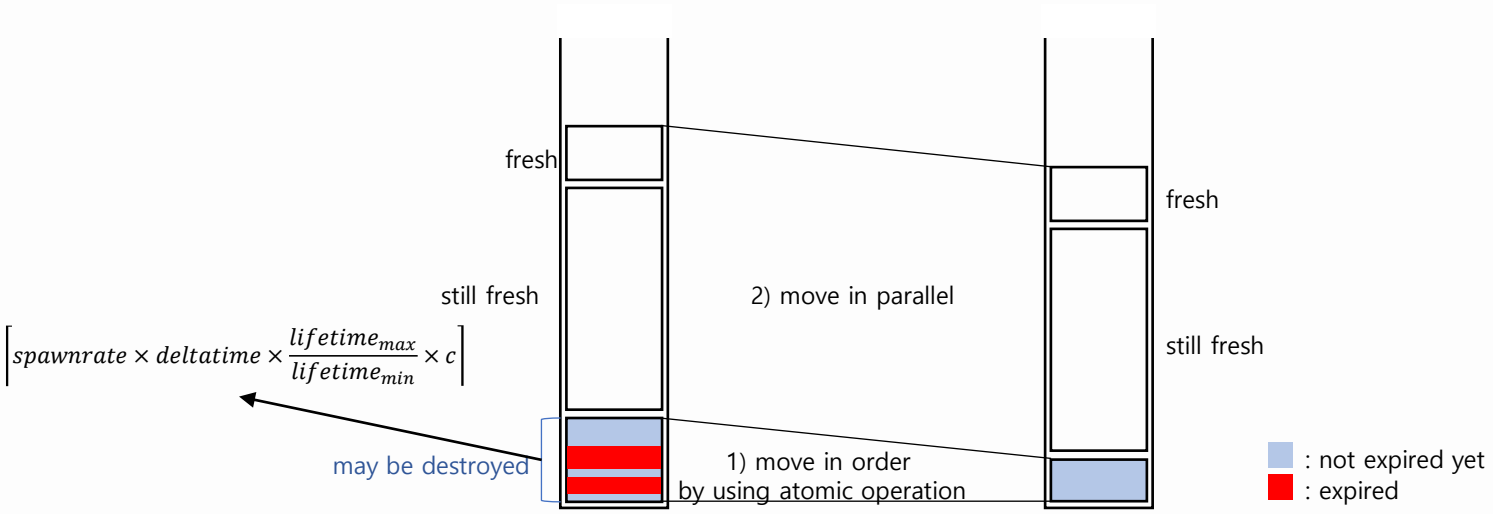
Optimization: Particle Management

- 기존의 방식은 파티클 버퍼가 존재하고, 이를 참조하는 인덱스 버퍼들을 이용해서 파티클을 관리하는 방식이었음.
  - 캐싱에 불리하다는 단점



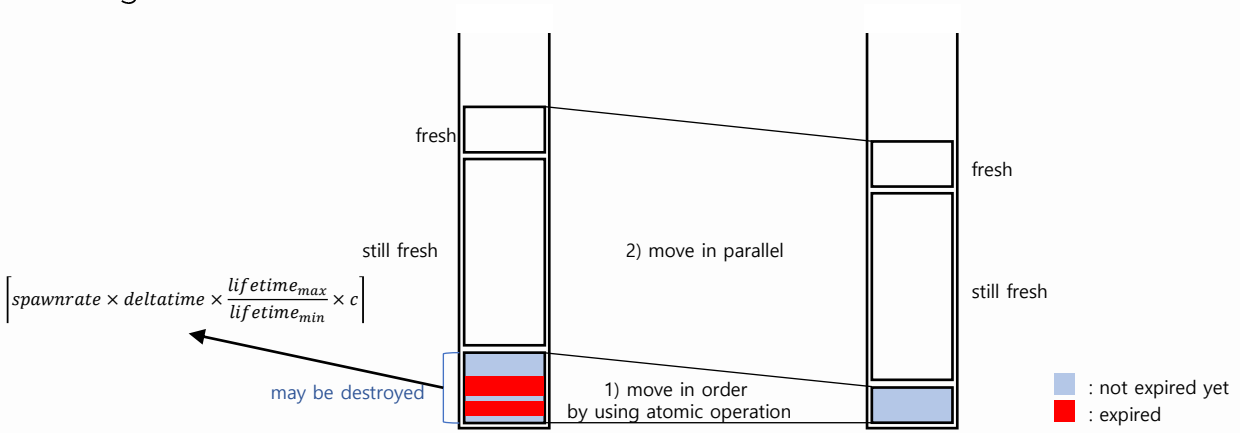
Optimization: Particle Management

- 파티클들의 파괴는 어느 정도 예측할 수 있다.
  - 먼저 태어난 파티클들은 먼저 죽는 경향이 있다.
  - 만약 모든 파티클들이 고정된 lifetime을 지닌다면, 순서대로 죽는다.
  - Lifetime이 random하더라도, random 값의 범위를 이용해서 "죽을 가능성이 있는" 파티클들의 수를 결정할 수 있다.



Optimization: Particle Management

- 추가적인 이점: 파티클들의 상대적인 순서가 (거의) 보존되므로 정렬이 필요없다.
  - fixed lifetime이라면 완벽하게 순서를 보존할 수 있음
  - Ribbon Rendering 성능 증가하여 10만 개 이상의 파티클도 Ribbon Rendering이 가능은 하게 되었음! (기존에는 200ms+)
  - 다만 Sprite Rendering의 경우 별다른 성능적 이점은 없었음



# of Particles	Emission	Destroy	Move Alives	Post-destroy	Simulation	Pre-prefix Sum	Prefix Sum	Computing Indirect Commands	Indirect Drawing	Total
100,000	0.013ms	0.025ms	0.768ms	0.000ms	0.550ms	0.127ms	0.865ms	0.001ms	35.458ms	38.287ms

Optimization: Data Packing

```
struct Particle
{
    float3 Position;
    float InitialSize;

    float3 Velocity;
    float InitialLifetime;

    float3 Acceleration;
    float InitialOpacity;

    float3 InitialColor;
    float RemainLifetime;

    float3 EndColor;
    float EndSize;

    float EndOpacity;
    float SpawnTime;
    uint SpawnOrderInFrame;
    float DistanceFromPrevious;

    float DistanceFromStart;
    float3 Pad;
};
```



```
struct Particle
{
    uint PositionXY;
    uint PositionZVelocityX;
    uint VelocityYZ;
    uint InitialLifetimeAndRemainLifetime;
    //----//
    uint AccelerationXY;
    uint AccelerationZAndSpawnTime;
    uint InitialSizeAndEndSize;
    uint DistanceFromPreviousAndDistanceFromStart;
    //----//
    uint InitialColor;
    uint EndColor;
};
```

- 112byte -> 40byte
- 패킹 코드 예

```
uint packFloat2ToUint(float a, float b)
{
    uint a16 = f32tof16(a);
    uint b16 = f32tof16(b);
    return (a16 << 16) | b16;
}

void unpackUintToFloat2(uint input, out float a, out float b)
{
    a = f16tof32(input >> 16);
    b = f16tof32(input & 0x0000FFFF);
}
```

Optimization: Data Packing

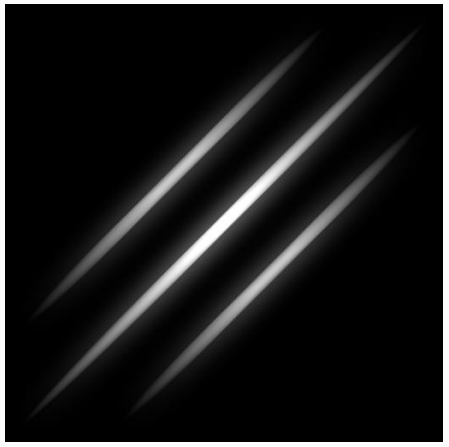
- 데이터 크기에 비례해서 성능이 향상됨을 확인할 수 있었음
- 특이사항:
  - Aligned 여부는 큰 영향을 미치지 않음
  - 64 byte일 때 데이터 크기가 더 작은 경우보다 Simulation 성능이 좋았음

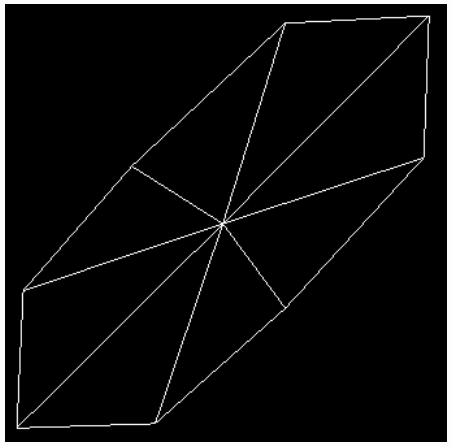
# of Particles	Particle Data Size	MoveAlives	Simulation	
1,000,000	112byte*	7.388ms	1.782ms	Particle 당 48-byte access
	80byte*	4.613ms	1.584ms	
	72byte	3.996ms	1.902ms	
	64byte*	2.972ms	0.891ms	Particle 당 24-byte access
	48byte*	2.159ms	0.994ms	
	44byte	2.040ms	1.040ms	
	40byte	1.738ms	0.893ms	

Optimization: Bounding K-gon

- Reference
  - Aggarwal, A., Chang, J.S. & Yap, C.K. Minimum area circumscribing Polygons. The Visual Computer 1, 112–117 (1985).
  - 주어진 Convex hull을 덮으면서 넓이가 가장 작은 최적의 k-gon을 찾는 방법

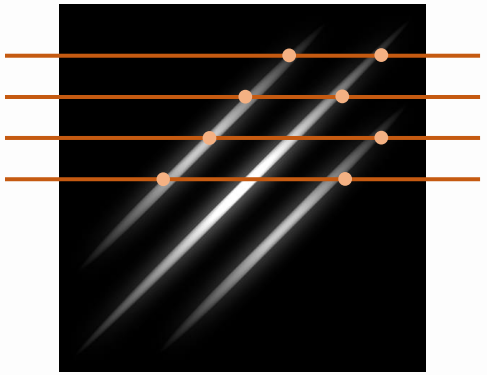
- Quad가 아닌 K개의 꼭짓점을 갖는 다각형으로 파티클을 렌더링하자!
  - 대부분의 픽셀이 기각되는 경우에 한해 overdraw를 방지할 수 있다.



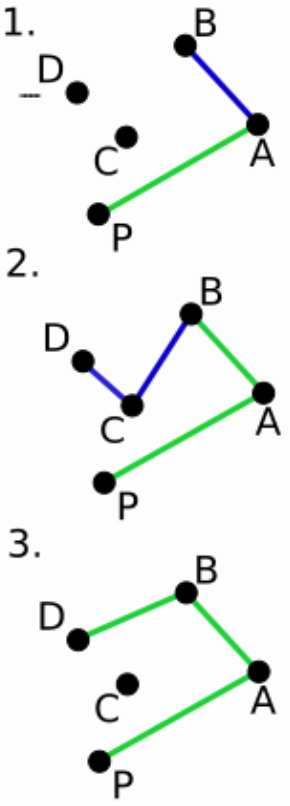


Convex Hull

- 각 행을 기준으로 alpha가 있는 양쪽 끝의 점을 선별
  - $N \times N$ 텍스처에서 최대 $2N$ 개의 점
  - $O(n^2)$
- Graham Scan으로 convex hull 생성
  - $O(n \log n)$



Point Selection



Graham Scan

Key Concepts

- Lemma 1. (DePano)
  - For k > 4, there exists a globally minimum k-gon Q that has at least k- 1 edges flush with P. Furthermore, all the edges of this k-gon are balanced.

- flushed?
  - 한 edge를 완전히 포함하고 양쪽으로 늘어난 경우
- balanced?
  - 한 vertex에서 접하고 접하는 부분이 변의 중점인 경우

- 따라서 $[i, j]$는 flushed chain, $[j, i]$는 balanced chain으로 가정하고, 모든 (i, j) 쌍에 대해서 탐색하면 된다.

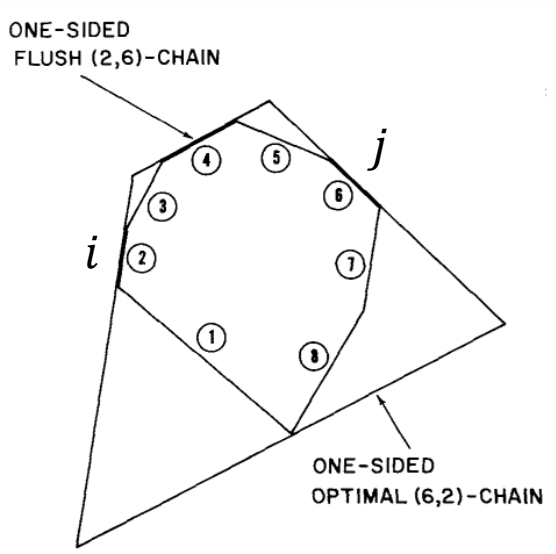


Fig. 1. Minimum circumscribing polygon

Key Concepts

- Lemma 2. (Interspersing Lemma)
  - Let $i < j < j'^{(mod\ P)}, C \in W_h(i,j), and\ C' \in W_h(i,j`). If C = C_h(i,j) and\ if\ C''$ $C_h(i,j')\ then\ C'\ intersperses\ C.$

- Lemma 3.
  - $If\ C_h^*(i,j)\ is\ optimal\ for\ W_h^*(i,j'), C_h^*(i,j')\ is\ optimal\ for\ W_h^*(i,j),\ and\ i <$ $j < j'(mod\ P)\ then\ C_h^*(i,j')\ intersperses\ C_h^*(i,j).$

- (i,j)쌍에 대해서 최적인 m이 찾아졌다면, (i,j') 쌍에 대해서는 $[i,j']$가 아닌 $[m,j']$쌍에 대해서만 탐색해도 됨!
  - 이를 재귀적으로 이용해서 log 시간으로 단축할 수 있음
  - i를 전부 순회: $O(n)$
  - $j_1 = i+1, j_2 = i-1$로 시작해 반절로 나누어가며 순회: $O(n \log n)$

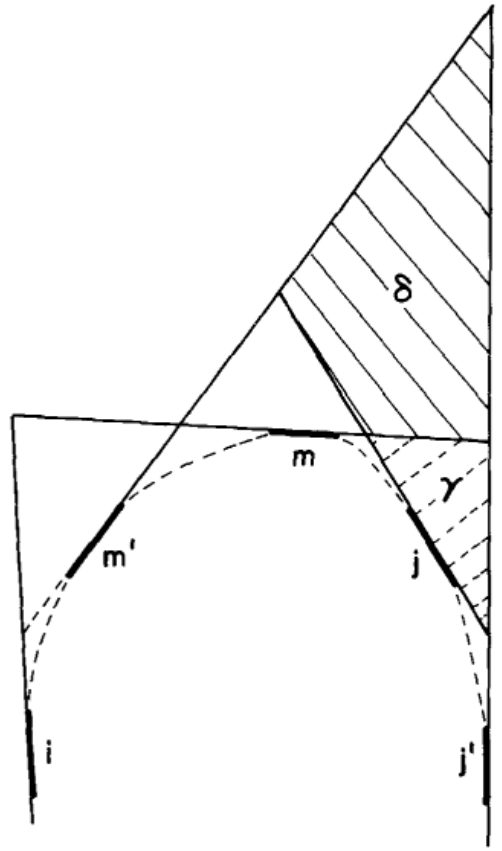


Fig. 2. Non-interspersing polygons

Key Concepts

- Balanced chain 탐색
  - 직선 $\vec{a} + t\vec{n}$을 k를 기준으로 점대칭 후 $\vec{b} + s\vec{m}$과의 교점을 이용해 balanced chain을 찾을 수 있음
  - 해당 교선이 convex hull을 뚫고 지나가는 경우에 대한 예외처리가 필요함!
    - k 양쪽의 edge와의 외적 결과의 부호를 이용해 이를 결정할 수 있음
    - 만약 convex hull을 뚫고 지나간다면, 해당하는 방향 쪽의 edge를 flush한 경우가 최적임.

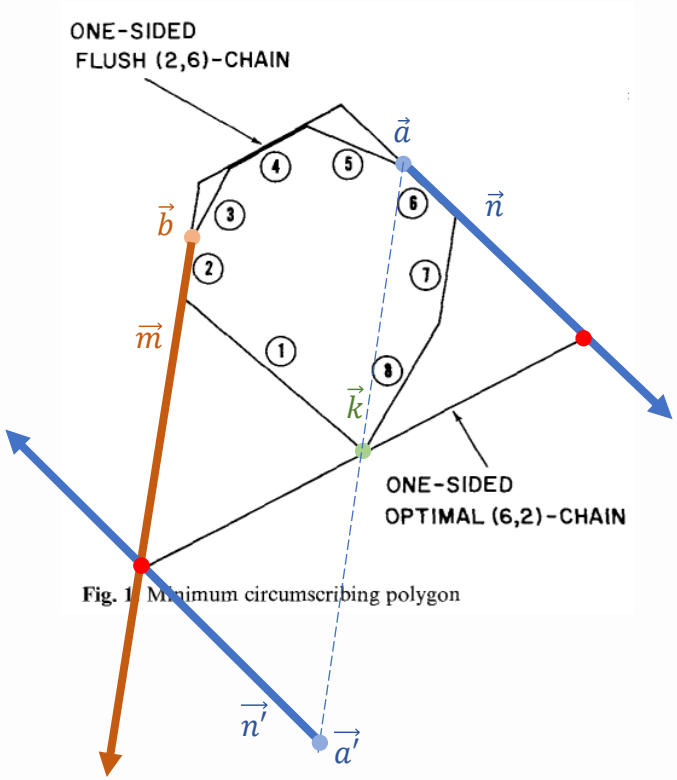
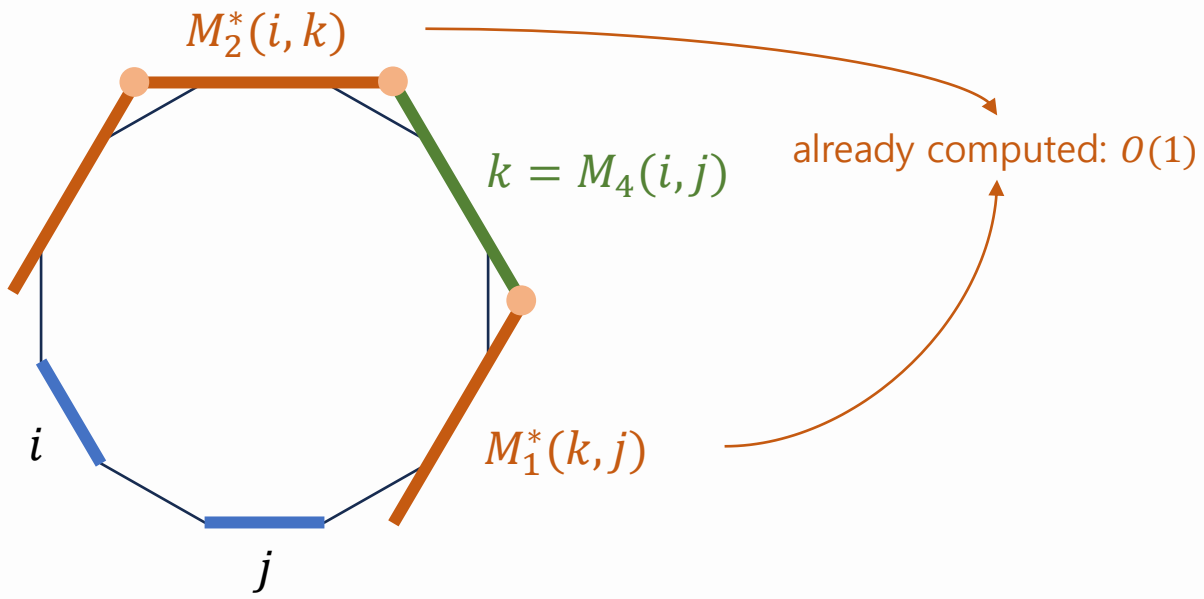


Fig. 1. Minimum circumscribing polygon

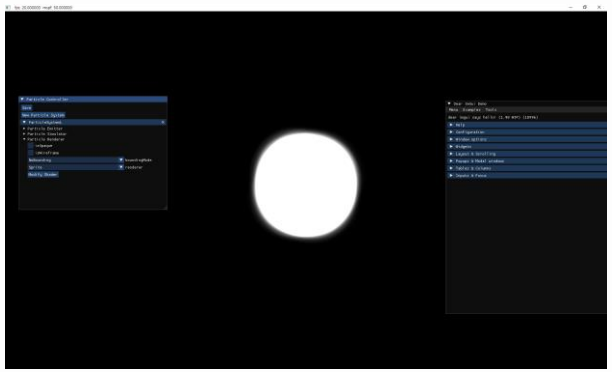
Key Concepts

- Flushed chain 탐색 & 넓이 계산
  - i와 j는 flushed edge이며 사이에 h 만큼의 flushed edge가 존재한다고 가정하고, h를 1부터 $k-3$까지 늘려가며 탐색
  - $M_h^*(i,j)$는 i와 j 사이에 h개의 flushed edge가 있고 이것이 최적일 때, **가운데**에 있는 flushed edge.
  - 이전에 계산되었던 $M_h^*(i,j)$를 이용해서 최적의 넓이를 바로 계산할 수 있음!
  - $O(n^2 k \log n)$

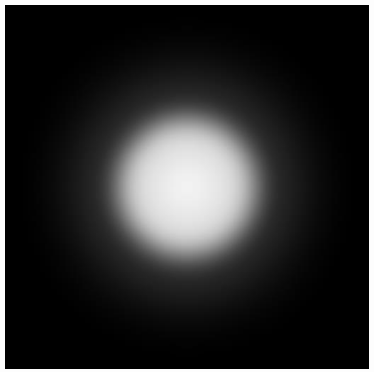


Optimization: Bounding K-gon

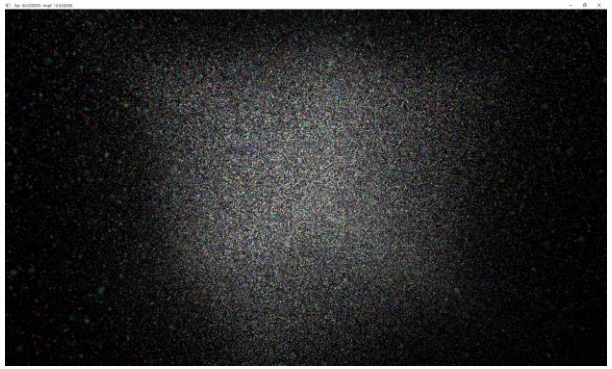
- Quad로 렌더링 시 반절 이상의 픽셀이 기각되는 텍스처를 선정
- 2가지 양극단의 상황을 가정하여 성능을 측정하였음:
  - 상황1: 파티클 하나가 화면의 픽셀을 많이 차지하는 경우 (파티클 3,500개)
  - 상황2: 파티클 하나가 화면의 픽셀을 적게 차지하는 경우 (파티클 1,000,000개)



상황1



Texture



상황2

Optimization: Bounding K-gon

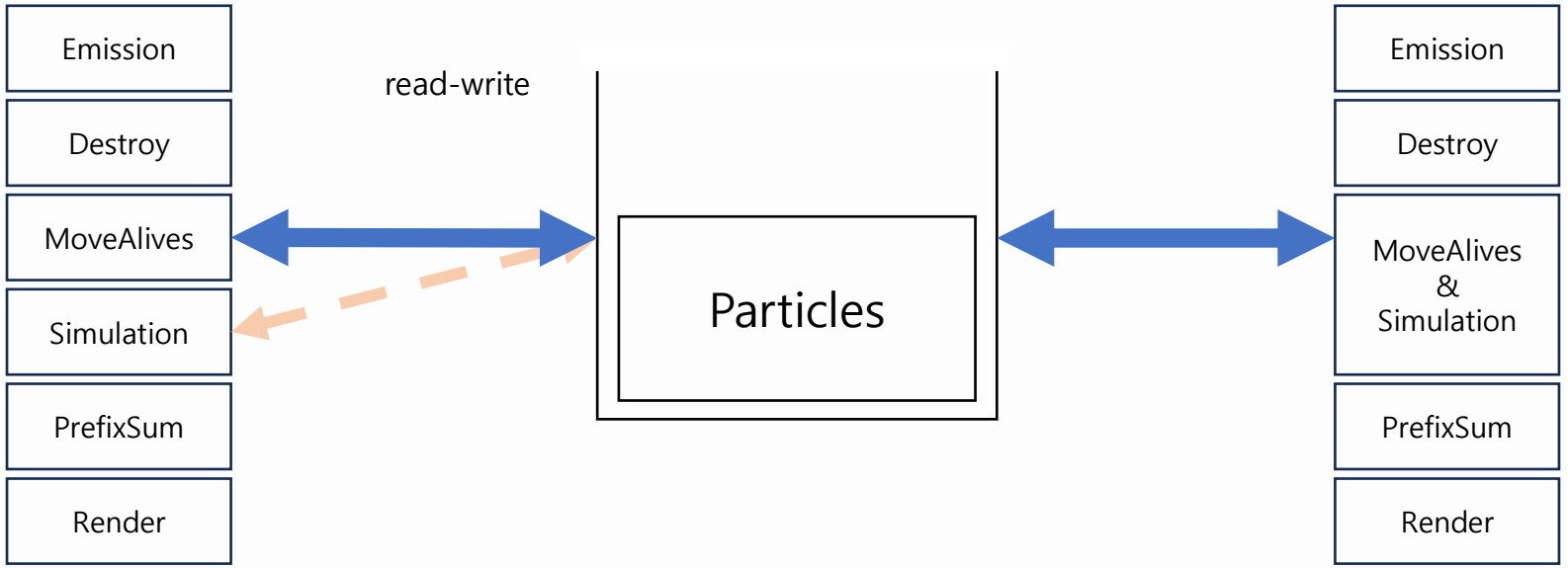
- 2가지 양극단의 상황을 가정하여 성능을 측정하였음:
  - 상황1: 파티클 하나가 화면의 픽셀을 많이 차지하는 경우 (파티클 3,500개)
  - 상황2: 파티클 하나가 화면의 픽셀을 적게 차지하는 경우 (파티클 1,000,000개)

Case	# of Particles	Bounding	Indirect Drawing
1	3,500	X	20.67ms
		O	16.54ms
2	1,000,000	X	7.41ms
		O	9.78ms

- 결론
  - 렌더링 시 실질적으로 기각되는 픽셀이 많을 때 유용할 수 있으나, quad가 아닌 n-gon을 생성하는 데 드는 비용을 적절히 고려해야 함.
  - 이러한 bounding이 필요 없도록 텍스처 및 파티클 효과와 관련한 아티스트의 지원을 받는 것이 가장 좋은 방법이며, 이는 최후의 수단으로 고려되어야 할 것임!

Optimization: Reducing Memory Access

- ping-pong 버퍼에 파티클을 옮기면서, 동시에 simulation의 결과도 계산할 수 있음
  - 굳이 둘의 pass를 나누어서 서로 다른 시점에 메모리를 접근하는 것보다, 한 번 접근할 때 레지스터 수준에서 연산을 수행하는 것이 훨씬 이득



Optimization: Reducing Memory Access

- ping-pong 버퍼에 파티클을 옮기면서, 동시에 simulation의 결과도 계산할 수 있음
  - 굳이 둘의 pass를 나누어서 서로 다른 시점에 메모리를 접근하는 것보다, 한 번 접근할 때 레지스터 수준에서 연산을 수행하는 것이 훨씬 이득

# of Particles	Emission	Destroy	MoveAlives	Post-destroy	Simulation	Computing Indirect Commands	Indirect Drawing	Total
1,000,000	0.015ms	0.010ms	1.738ms	0.001ms	0.89ms	0.001ms	6.797ms	9.456ms
	0.018ms	0.012ms	1.734ms	0.001ms	-	0.001ms	6.674ms	8.440ms

- 해석 시 주의 사항:
  - 위 경우 Simulation에서 단순한 연산(Emission 시 정의된 속도와 가속도를 기반으로 위치를 조정)만 하고 있으므로 Simulation에서 드는 비용이 실질적으로 메모리를 읽고 쓰는 연산 이외에는 없음.
  - 시뮬레이션 연산에 대한 비용은 여전히 존재하며, 100만 개 기준 0.9ms의 상수 시간이 최적화 되었다고 보는 것이 타당함!
    - 예를 들어 curl noise force를 적용할 경우 100만개 기준 2ms가 추가적으로 소요됨.

Performance Comparison: Sprite

- FHD, Opaque, Sprite

# of Particles	Emission	Simulation	Post-simulation	Computing Indirect Commands	Indirect Drawing	Total	Description
1,000,000	0.074ms	11.903ms	0.001ms	0.517ms	13.620ms	26.115ms	초기
	0.032ms	7.685ms	0.001ms	0.002ms	8.165ms	15.885ms	Cache 고려 & Instancing

# of Particles	Emission	Destroy	MoveAlives	Post-destroy	Simulation	Computing Indirect Commands	Indirect Drawing	Total	Description
1,000,000	0.043ms	0.036ms	7.388ms	0.000ms	1.782ms	0.001ms	8.483ms	17.733ms	구조 변경
	0.018ms	0.013ms	4.000ms	0.001ms	1.902ms	0.001ms	8.191ms	14.122ms	일부 Data Packing
	0.021ms	0.031ms	4.002ms	0.001ms	1.902ms	0.001ms	6.891ms	12.855ms	렌더링 시 메모리 접근 최소화
	0.015ms	0.010ms	1.738ms	0.001ms	0.89ms	0.001ms	6.797ms	9.456ms	Data Packing
	0.018ms	0.012ms	1.734ms	0.001ms	-	0.001ms	6.674ms	8.440ms	Simulation Pass 통합

Performance Comparison: Ribbon

- FHD, Opaque, Ribbon, Distance based UV

# of Particles	Emission	Simulation	Post-simulation	Pre-sort	Sort	Pre-prefix Sum	Prefix Sum	Computing Indirect Commands	Indirect Drawing	Total	Description
10,000	0.074ms	0.005ms	0.001ms	0.042ms	0.451ms	0.039ms	0.874ms	0.032ms	3.981ms	5.497ms	초기
	0.032ms	0.126ms	0.001ms	4.707ms	8.149ms	0.374ms	0.521ms	0.002ms	3.070ms	16.958ms	Cache 고려 & Instancing
100,000						200ms+					

# of Particles	Emission	Destroy	Move Alives	Post-destroy	Simulation	Pre-prefix Sum	Prefix Sum	Computing Indirect Commands	Indirect Drawing	Total	Description
100,000	0.013ms	0.025ms	0.768ms	0.000ms	0.550ms	0.127ms	0.865ms	0.001ms	35.458ms	38.287ms	구조 변경
	0.007ms	0.004ms	0.180ms	0.001ms	0.112ms	0.087ms	0.732ms	0.002ms	25.150ms	26.276ms	Data Packing

Thank you

Subin Han