

# 실시간 게임서버 최적화 전략

유영천

tw:@dgtman

<https://megayuchi.com>

# 들어가기 전에...

- 이 발표는 Voxel Horizon 프로젝트의 서버 아키텍처를 소개함.
- 최적화가 너무나 중요한 프로젝트이기 때문에 좋은 예가 될 것으로 판단했기 때문.
- 다른 방식의 서버 아키텍처를 비난할 의도 없음.

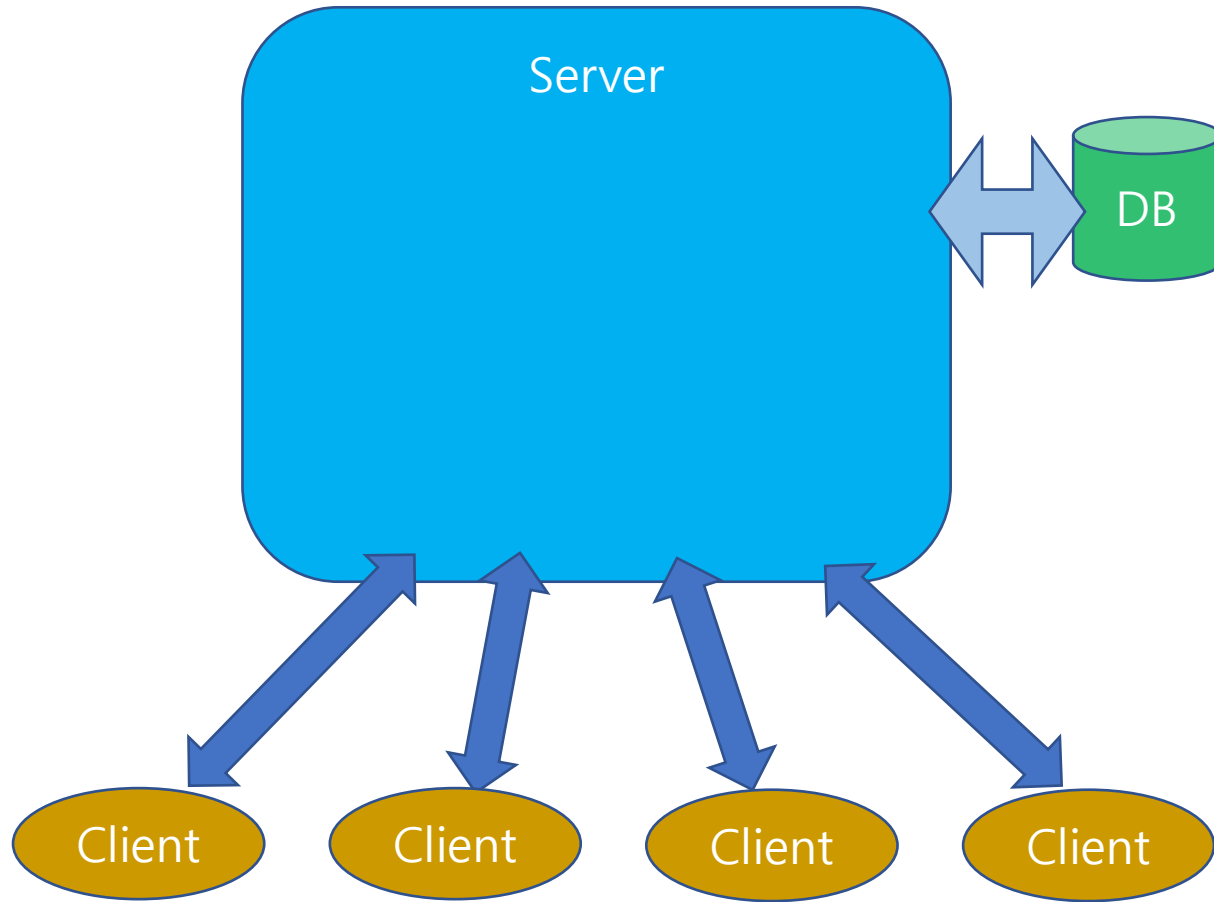
# 이런 게임을 만듭니다. (Voxel Horizon)



<https://youtu.be/V3f8eHy9rHI>

서버가 하는 일

# 수동적인 서버

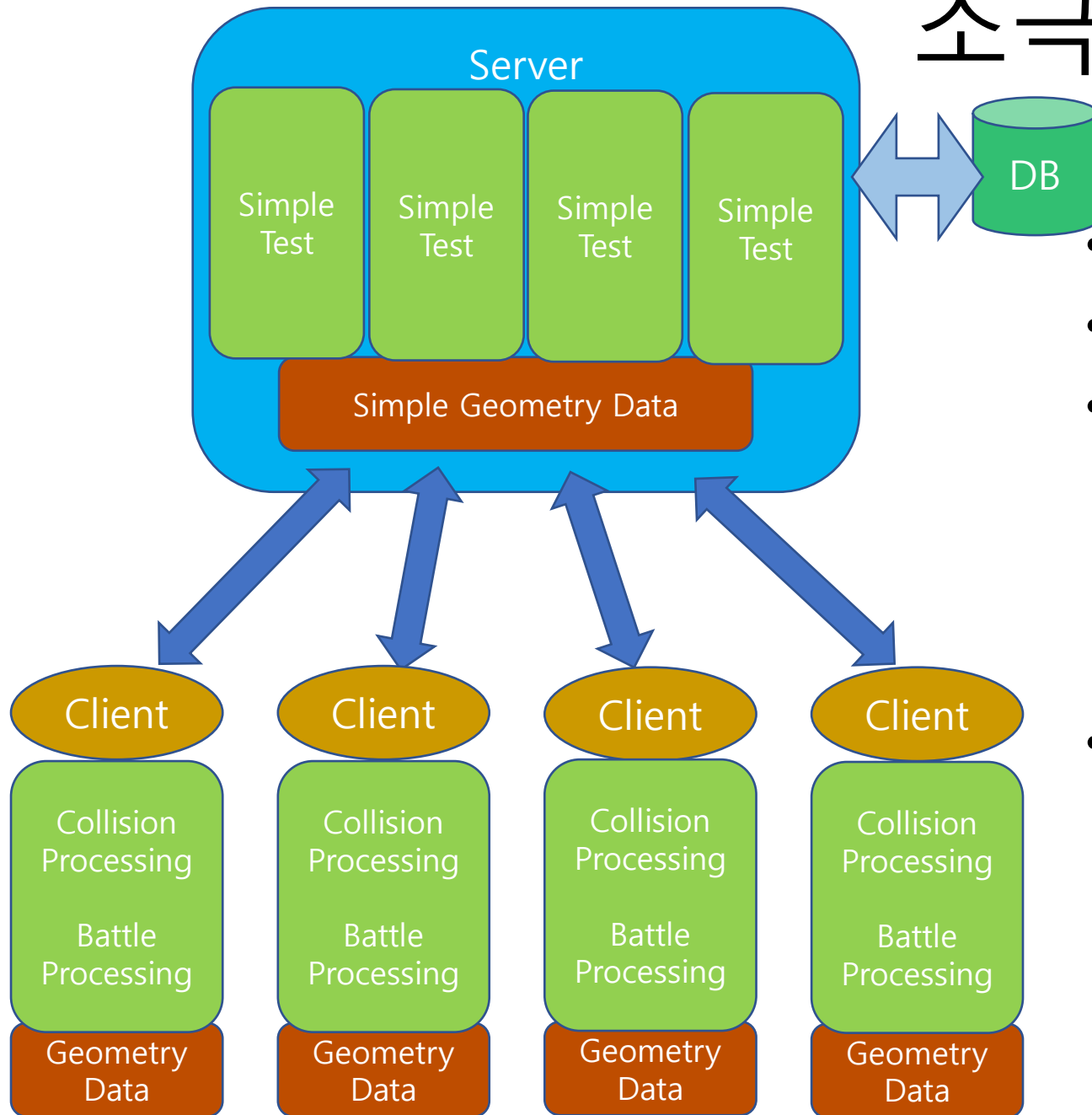


- 클라이언트가 요청한 서비스 제공
  - 클라이언트의 상태를 저장
  - 패킷 중계
- Stateless 서버
- 기본적으로 요청을 받아서 응답
- 웹서버등...
- 오늘 주제와 무관.

# 능동적인 서버 – 일반적인 게임서버

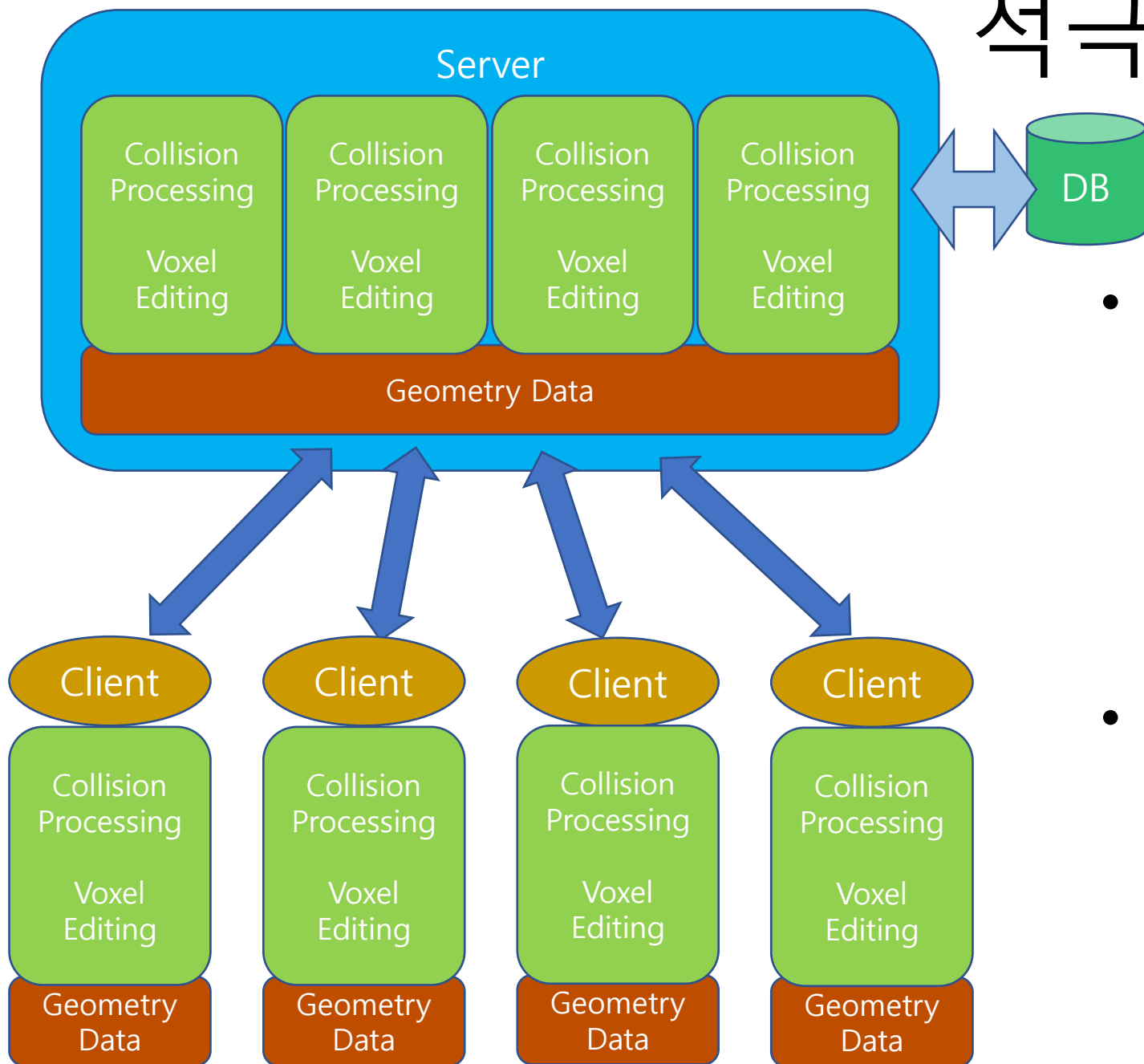
- 상태를 서버의 메모리에 가지고 있다.
- DB는 보조수단이다.
- 클라이언트와의 시간차 0을 목표로 한다(불가능하지만).
- 클라이언트의 요청이 없어도 서버에서 상태를 유지/변경 한다.
- 게임 루프 필요.

# 소극적으로 능동적인 서버



- State를 가지고 있고 게임루프도 있음.
- 서버에서 능동적으로 게임을 진행.
- 단 성능/기술적 문제로 많은 정보를 클라이언트에 의존.
  - (스피드해킹 등)해킹에 취약할 수 있다.
- 일본산 콘솔용 MO들 상당수가 여기에 해당.

# 적극적으로 능동적인 서버



- 클라이언트의 동작을 서버에서 그대로 시뮬레이션
  - 서버측에서의 이동(충돌)처리
  - 서버측에서의 정밀한 전투판정
  - 서버측에서의 복셀 데이터 편집
- 이론상 접속한 클라이언트 수만큼 부하 증가.

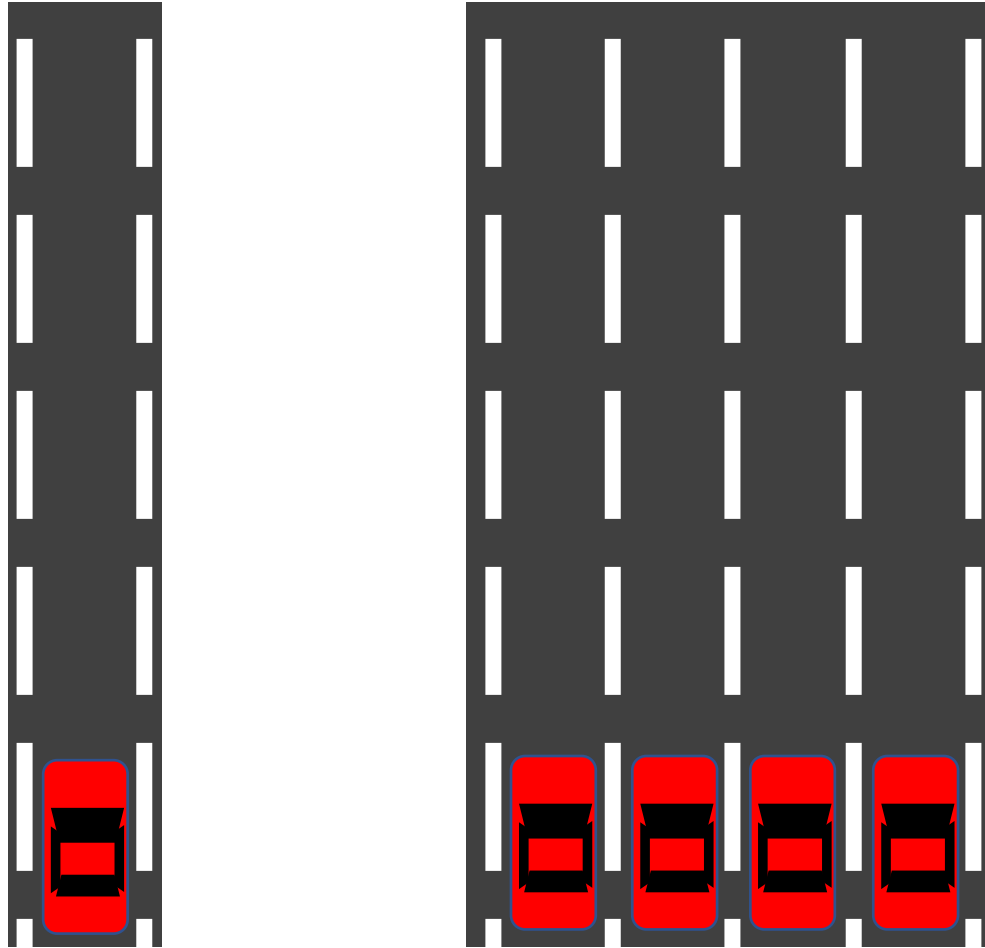


# 서버가 하는 일 in Voxel Horizon

- 유저로부터의 패킷에 의한 이벤트 처리 OnRecvPacket()
- 플레이어 or NPC(몬스터포함)에 대한 이동처리
- 플레이어 or NPC(몬스터포함)에 대한 무기 ray 테스트
- DB액세스 작업(읽기, 쓰기, SP호출)
- 유저로부터의 요청에 의해 복셀 편집
- 변형된 복셀 데이터에 대한 압축처리
- 플레이어의 개인맵 로드/세이브

응답성 향상을 위한 기법

# 처리량과 응답성의 차이



제한속도 100km

4차선 도로는 1차선 도로에 비해서 시간당 4배 많은 차량을 이동시킬수 있다.

-> **처리량(시간당)**

하지만 1대든 4대든 목적지까지 걸리는 가장 빠른 시간은 똑같다.

-> **응답성**

차량의 수가 많아지면 1차선 도로는 4차선 도로에 비해 명백하게 시간이 더 걸린다.

-> **처리량 부족으로 인한 응답성 저하**

# 응답성

- 실시간 온라인 게임에서는 플레이어의 ms단위의 조작을 그대로 반영해야 한다.
- 여기서 말하는 응답성이란 플레이어의 입력에 대한 전체 게임 시스템의 반응의 신속한 정도.
- 응답성과 시간당 처리량이 반드시 일치하지는 않는다.
- 시간당 처리량이 부족하면 응답성이 떨어질 수 있다.
- 처리량이 부족해서 응답성이 떨어질 경우 처리량을 개선한다.

# 응답성을 향상을 위한 주요 기법

- 비동기 처리
  - 특정 작업 때문에 전체 응답성이 떨어지는 상황을 방지
  - 멀티 스레드 - 별도의 worker thread가 처리
  - 싱글 스레드 - 작업을 쪼개서 원래의 작업 스레드에서 여러 번에 걸쳐 처리.
- Thread Pool을 이용한 병렬처리
  - 서로 의존성을 가지지 않는 N개의 항목에 대해 M개의 worker thread가 병렬로 처리. Thread Pool이 작업을 완료할 때까지 원래의 작업 스레드는 대기
- 코드 최적화 -> SIMD등
- 조합해서 사용 가능

# 비동기 처리 (멀티스레드 or 싱글스레드)

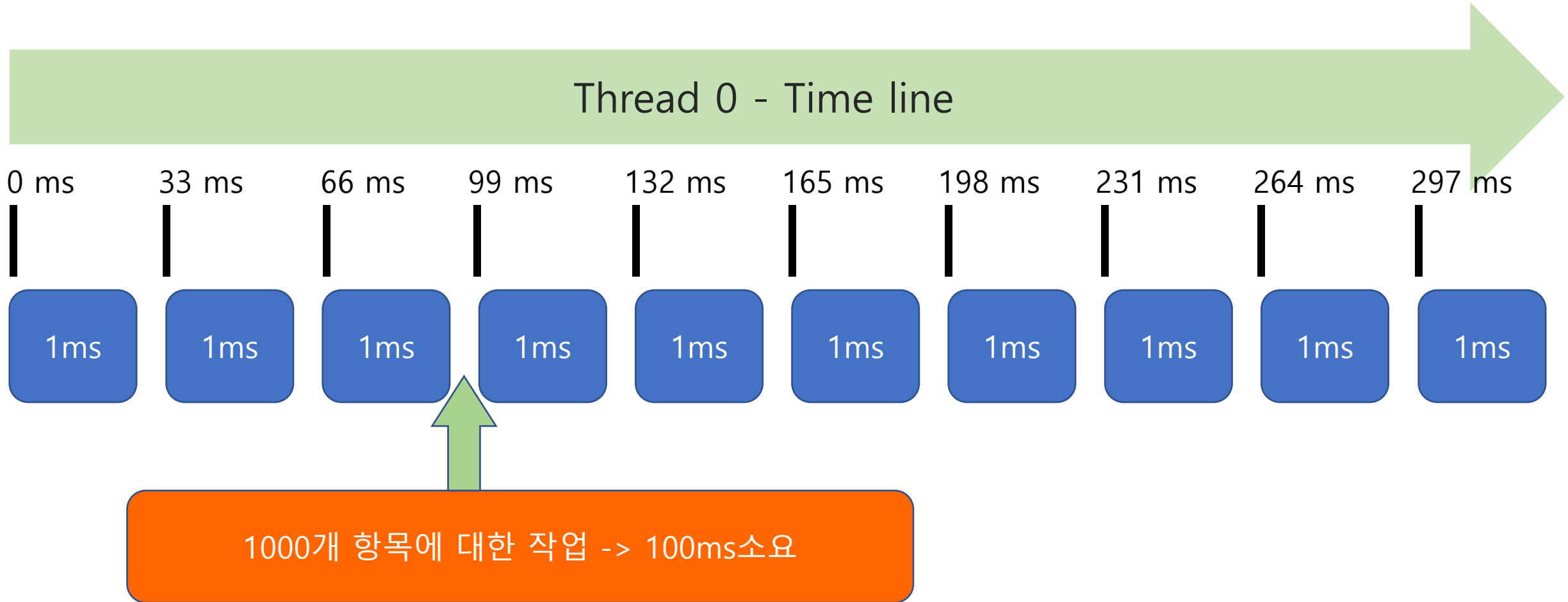
- 급하지 않은 작업.
- 특정 작업 때문에 전체 응답성이 떨어지는 상황을 방지
- 작업 자체의 완료시간은 늦어짐.
- 로그인 , 캐릭터 생성/삭제, 아이템 획득/거래 등 DB 종속적인 작업들.
- 저장소에 대한 I/O작업.

# 비동기 처리 - 싱글 스레드

- 별도의 worker thread를 생성하지 않는다.
- 긴 시간이 걸리는 작업을 쪼개서 메인 스레드의 타임라인에 분산해서 처리.
- 시간 전체 처리 소요 시간은 그대로, 처리까지 걸리는 시간은 상승, **응답성은 상승**.<- 가장 중요한 목표
- 스레드간 동기화가 복잡해질 경우 선택.
- 스레드간 동기화 비용 0.
- 멀티 스레드는 아니지만 비동기 처리.
- 메인 스레드의 응답성을 약간 저하시킬 수 있다.

# 동기식 처리

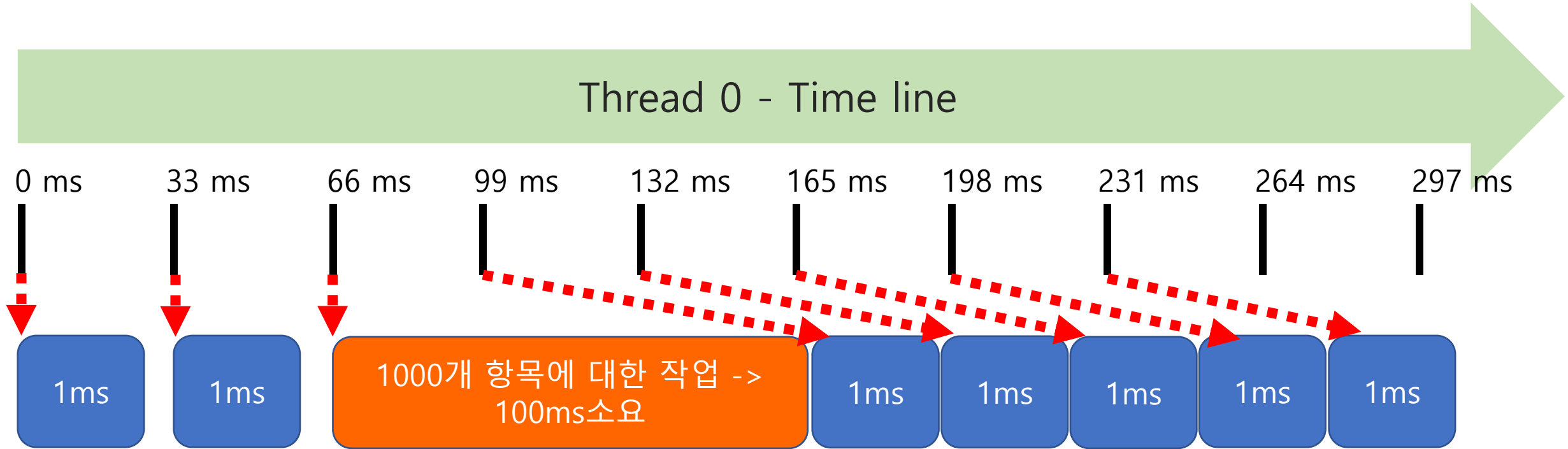
타이머에 의해 33ms마다 콜백함수 호출중.





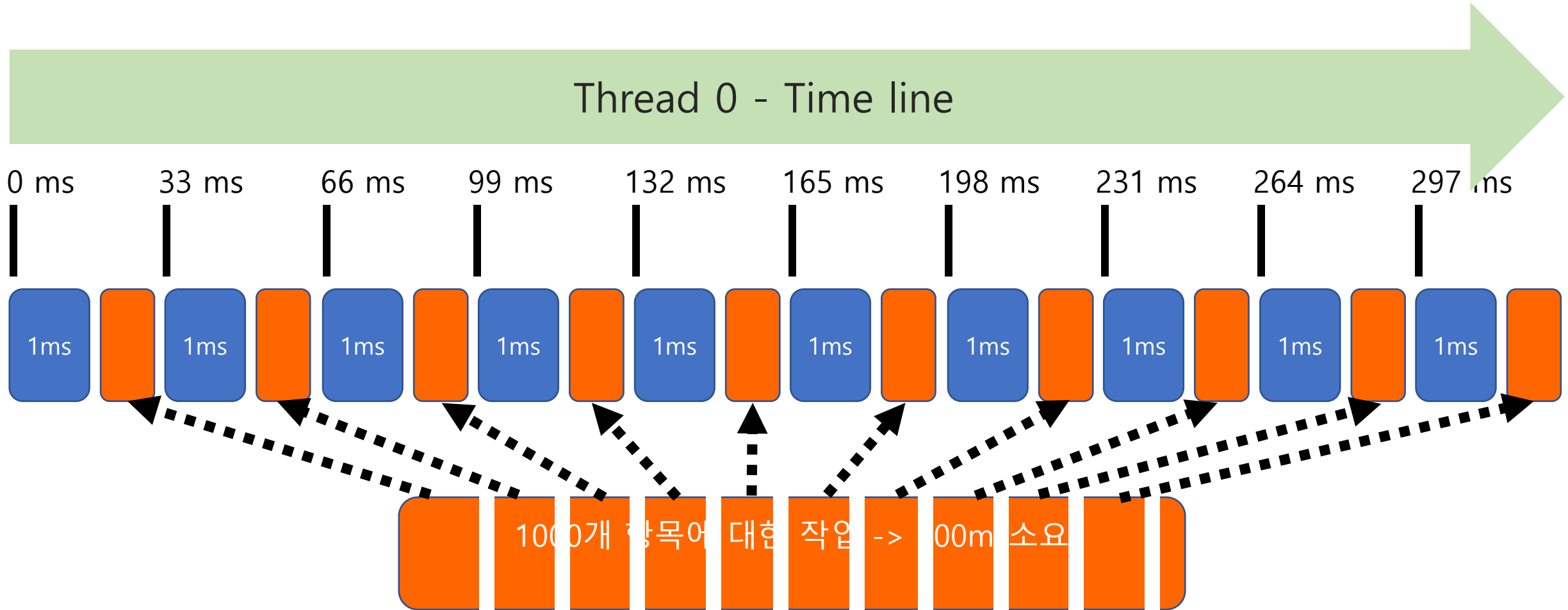
# 동기식 처리

타이머에 의해 33ms마다 콜백함수 호출중.



# 비동기식 처리 - 싱글 스레드

타이머에 의해 33ms마다 콜백함수 호출중.



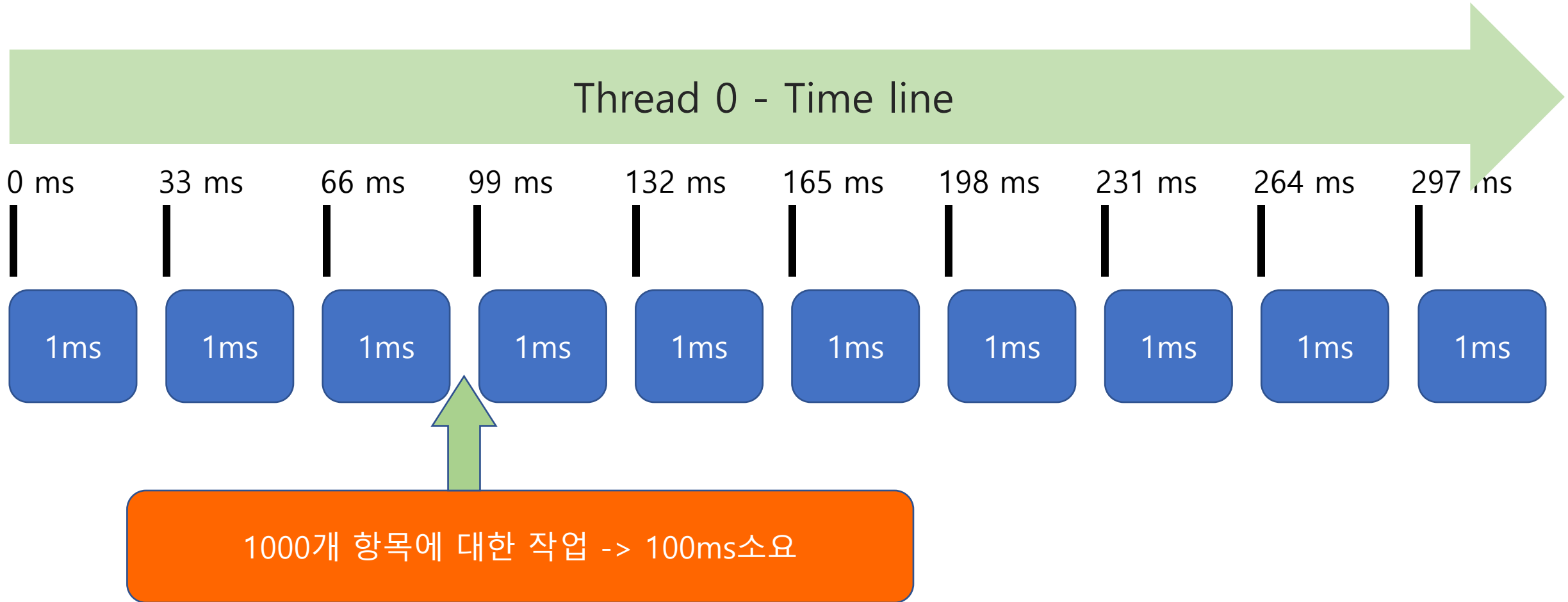
1000개씩 처리 -> 100개씩 처리  
작업단위당 100ms -> 10ms로 감소.

# 비동기 처리 - 멀티 스레드

- 별도의 Worker thread에 작업을 할당하여 완전히 백그라운드로 작업
- 남는 CPU자원을 조금이라도 더 사용할 수 있다.
- Worker thread로 작업을 넘겼을 때 때 필요한 자원(메모리 등)도 완전히 분리할 수 있을 경우(동기화 부담이 없을 경우)에 적합.
- 메인 스레드 응답성에 거의 영향을 끼치지 않는다.
- 스레드 동기화 비용이 아주 적을 경우만 사용해야한다.

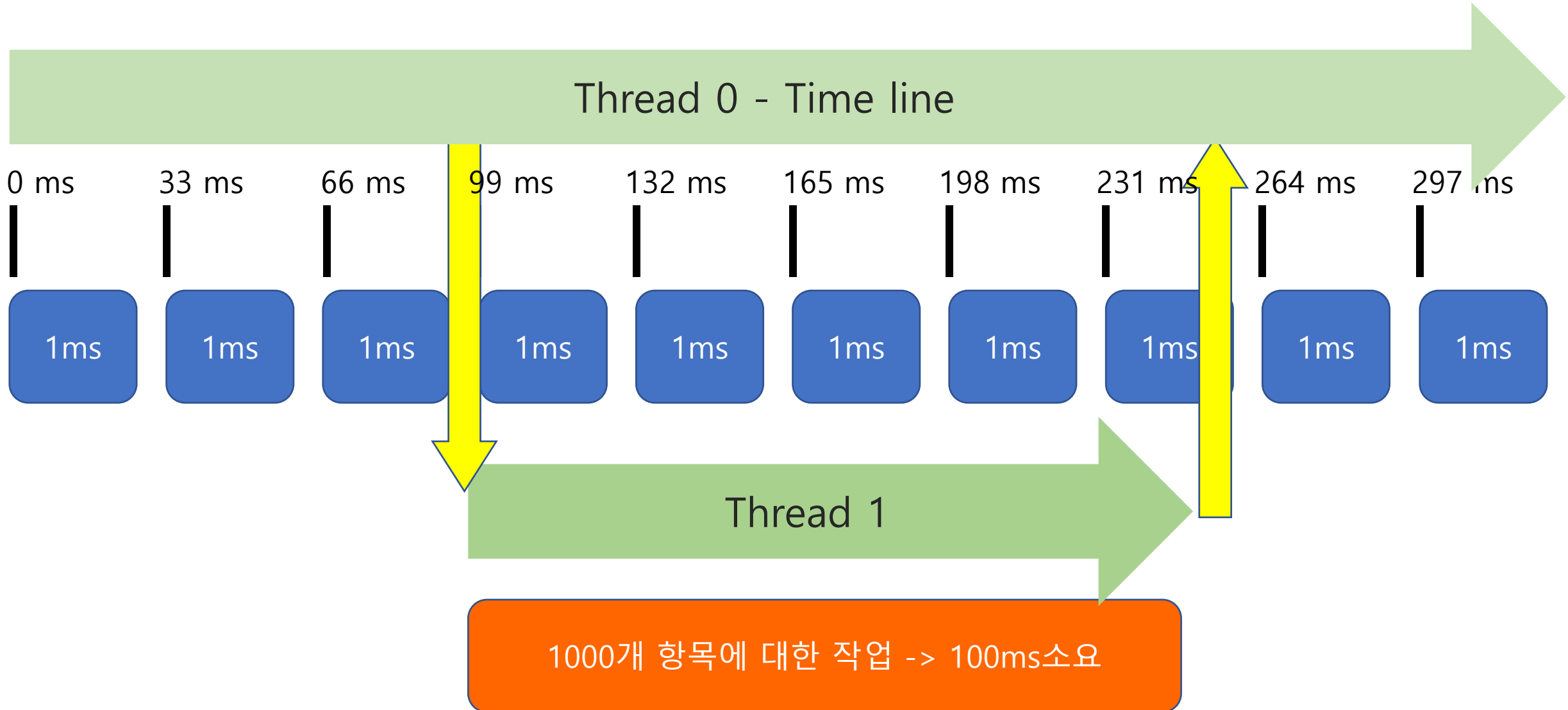
# 동기식 처리

타이머에 의해 33ms마다 콜백함수 호출중.



# 비동기식 처리 - 멀티 스레드

타이머에 의해 33ms마다 콜백함수 호출중.

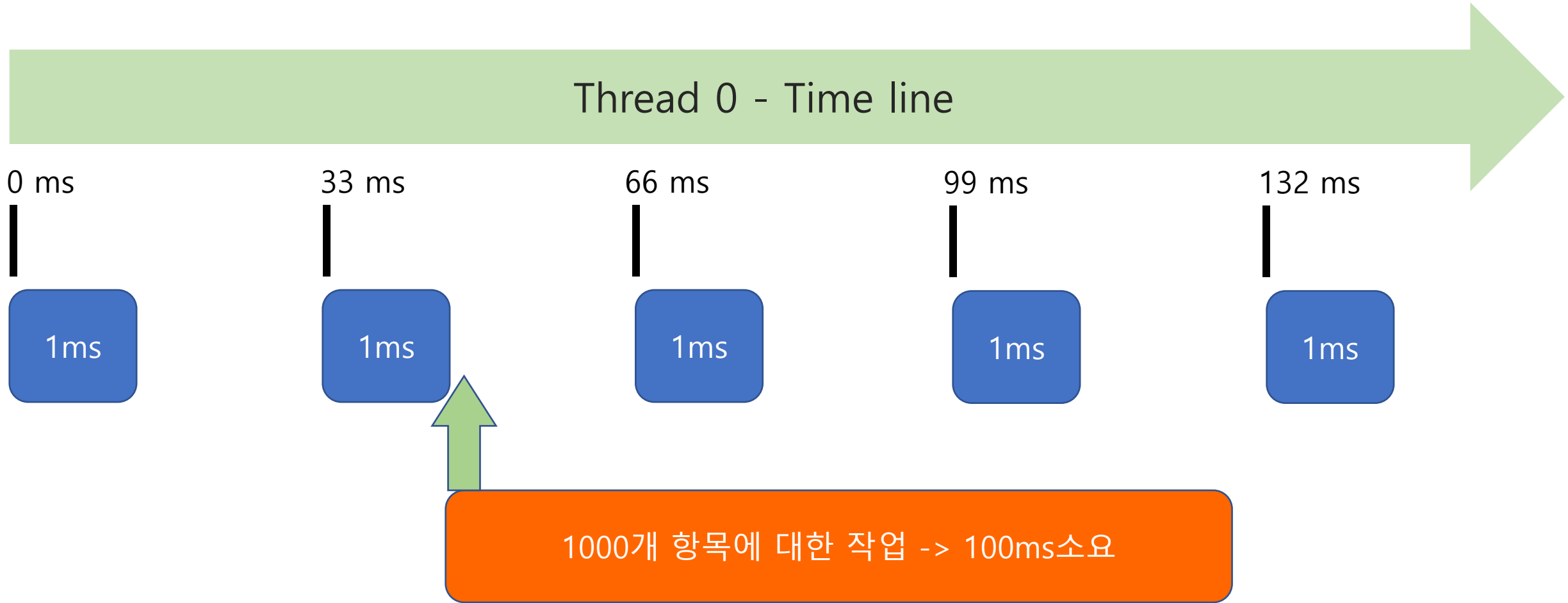


# Thread Pool을 이용한 병렬처리

- 1 차선 도로를 N차선 도로로 확장.
- 처리량 때문에 지연이 발생하는 경우 사용.
- 처리해야할 항목(원소) 각각이 서로 의존성이 거의 없어야 함.
- Thread per core일 때 (이론상) 거의 선형적으로 처리량 증가.
- 비동기 처리와 병행 가능.

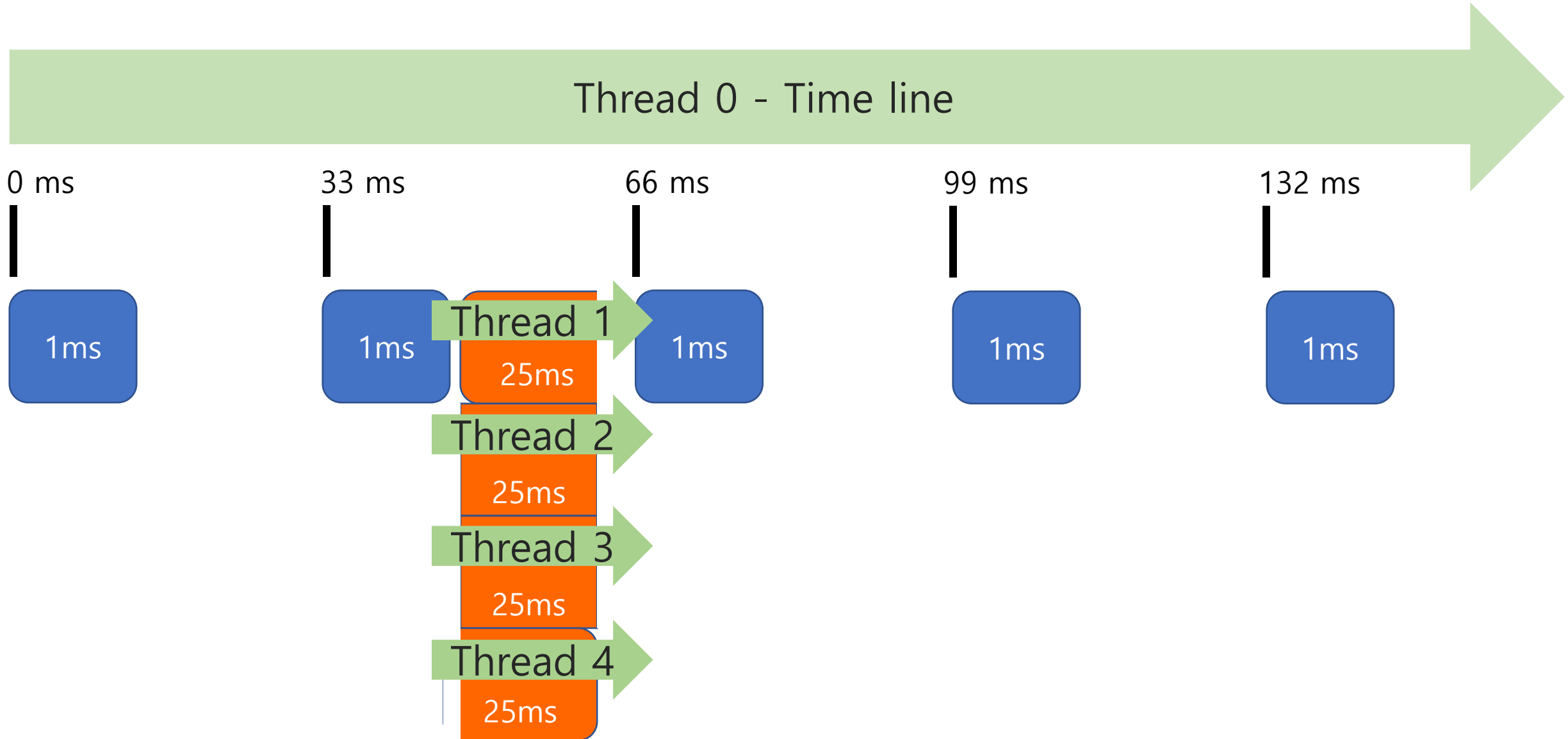
# 동기식 처리

타이머에 의해 33ms마다 콜백함수 호출중.



# 동기식 처리 – Thread Pool

타이머에 의해 33ms마다 콜백함수 호출중.





# 코드 최적화 -> SIMD등

- 당장 수행되어야 할 코드.
- 컨텍스트 스위칭 없이 현재 컨텍스트 상에서 당장 빠르게 코드가 실행되어야 하는 경우
- Thread Pool을 쓰는 경우 Thread Pool 이 작동하기 시작해서 결과를 동기화할 때까지의 비용이 더 클 수 있다(배보다 배꼽이 더 커요).
- 그냥 정직하게 빠른 코드를 짜야한다.
- 수학 연산이 많다면 SIMD는 반드시 사용한다.

# 응답성 향상을 위한 코딩

- 범용 Heap보다는 Stack을 사용.
- 범용 Heap보다는 고정 사이즈 Memory Pool(병합기능x)
- 적절한 동기화 객체 선택
- SIMD 최적화의 경우 전체적으로는 최대 30%정도까지 성능향상을 기대한다.

# SIMD vs no SIMD 소모 clock 비교

unit : clocks(cycles)	using SIMD	Surface Pro X	Surface Pro X(x86)	i7-8700K Desktop	Azure VM(Xeon8168)	Alienware 15(2015)	Surface Book 2	Surface Book 1	Surface Pro 2	GPDWIN2
Processor		Microsoft SQ1	Microsoft SQ1	intel i7-8700K	Xeon 8168	Intel Core i7-4710HQ	intel i7-8650U	intel i5-6300U	intel i5-4300U	intel Core m3-7Y30
Equations of planes from a triangle	No SIMD	49.43 (G)	315.20 (G)	148.98	143.68	103.36	68.2	118.56	173.85	113.47
1 triangle = (float3, float3, float3)	SIMD			37.58	36.64	33.63	23.92	82.36	131.44	67.73
	No SIMD	119.86 (S)	1667.20 (S)							
	SIMD									
Distance between 2 points (float3)	No SIMD	22.07 (G)	205.73 (G)	18.69	25.07	26.16	12.34	26.51	44.75	32.64
	SIMD	26.01 (G)	156.62 (G)	12.74	11.89	12.72	8.41	13.47	22.56	22.02
	No SIMD	93.55 (S)	555.23 (S)							
	SIMD	90.22 (S)	369.50 (S)							
Dot product (float3)	No SIMD	17.56 (G)	57.02 (G)	9.37	10.32	12.68	10.77	10.44	50.31	7.28
	SIMD									
	No SIMD	64.19 (S)	369.31 (S)							
	SIMD									
Cross Product (float3)	No SIMD	18.35 (G)	99.60 (G)	13.62	18.36	53.47	14.4	15.37	32.34	14.6
	SIMD			12.56	12.2	13.82	9.08	13.34	25.03	9.62
	No SIMD	80.35 (S)	758.30 (S)							
	SIMD									
Matrix multiplication (float 4x4)	No SIMD	64.01 (G)	408.00 (G)	87.34	67.52	69.98	46.86	96.98	125.55	68.45
	SIMD	19.14 (G)	163.20 (G)	21.58	23.8	21.44	14.8	36.02	40.25	34.03
	No SIMD	123.94 (S)	5616.00 (S)							
	SIMD	87.39 (S)	3672.00 (S)							
Normalization vector (float3)	No SIMD	25.63 (G)	302.99 (G)	22.39	27.35	35.56	31.15	25.38	46.95	32.47
	SIMD	20.69 (G)	155.02 (G)	12.92	13.59	13.6	12.1	13.76	24.2	23.16
	No SIMD	83.63 (S)	694.21 (S)							
	SIMD	70.37 (S)	458.32 (S)							
Vector3 x 4x4 Matrix (float3 , float4x4)	No SIMD	11.83 (G)	44.82 (G)	9.69	13.52	37.26	6.82	9.94	15.34	8.62
	SIMD	10.35 (G)	27.22 (G)	5.37	4.78	5.36	3.63	5.94	9.6	4.07
	No SIMD	61.09 (S)	203.44 (S)							
	SIMD	31.25 (S)	75.47 (S)							
Vector4 x 4x4 Matrix (float4 , float4x4)	No SIMD	18.58 (G)	66.90 (G)	14.85	14.28	13.09	9.7	18.71	24.67	13.6
	SIMD	3.59 (G)	20.40 (G)	4.26	3.73	4.36	2.63	5.02	8.36	3.22
	No SIMD	81.88 (S)	307.94 (S)							
	SIMD	22.70 (S)	72.08 (S)							

# SIMD vs no SIMD 소모 clock 비교

[illegible]

# 실제 게임서버에 적용

Voxel Horizon에서의 사례

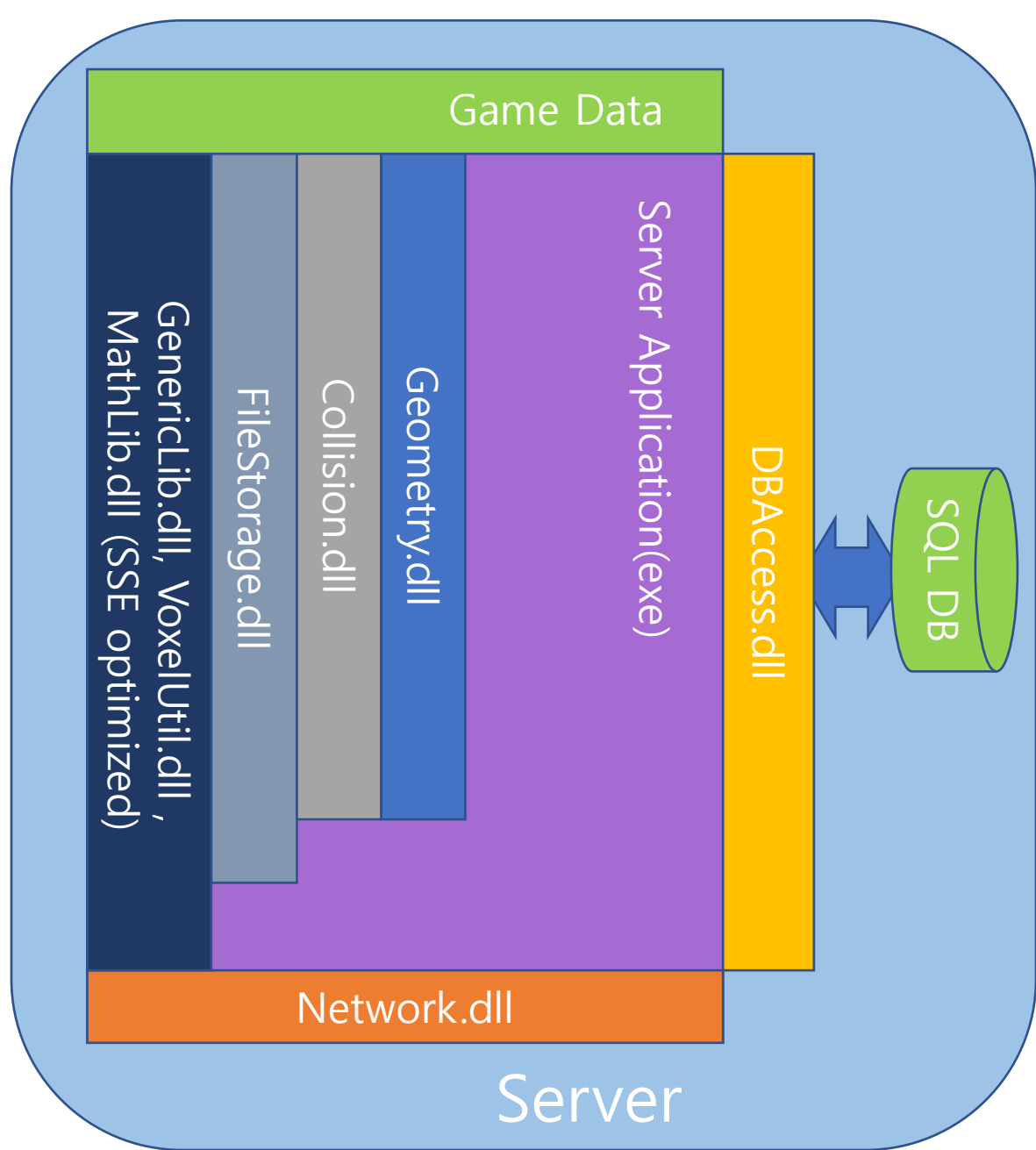
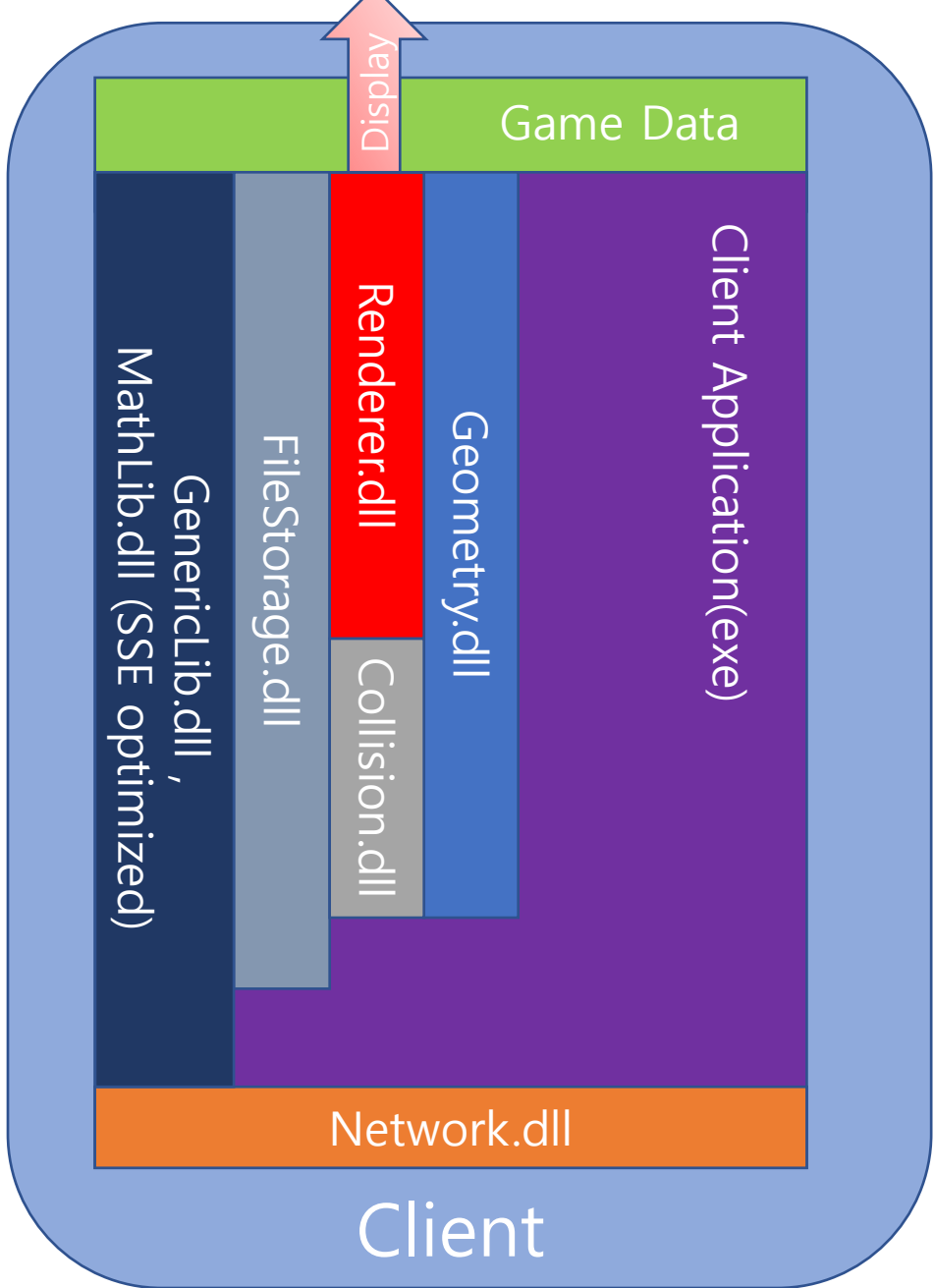
# 기본전략

- 응답성 최우선
- 1개의 메인스레드 + N개의 보조 스레드 풀
  - 동기화로 인한 성능 저하 방지
  - 유지보수 용이(디버깅, 후임자 인수인계)
- 클라이언트와 서버가 대부분의 코드를 공유할 것
  - 동일 입력 동일 결과 보장
  - 유지보수 용이
- 멀티 스레드 남용 금지!
- 가능하면 SIMD사용 -> 그러나 큰 기대는 금물

# 서버와 클라이언트의 코드공유

- 클라이언트와 서버는 최대한 코드를 공유한다.
- 서버와 클라이언트가 동일한 입력에 대해 동일한 결과를 보장.
- 엔진 코드는 서버에서도 그대로 사용하므로 서버에서 필요한 기능을 지원할것.
  - 다수의 인스턴스 생성
  - 멀티 스레드를 이용한 병렬 처리
  - 비동기 처리

# 서버와 클라이언트의 코드 공유

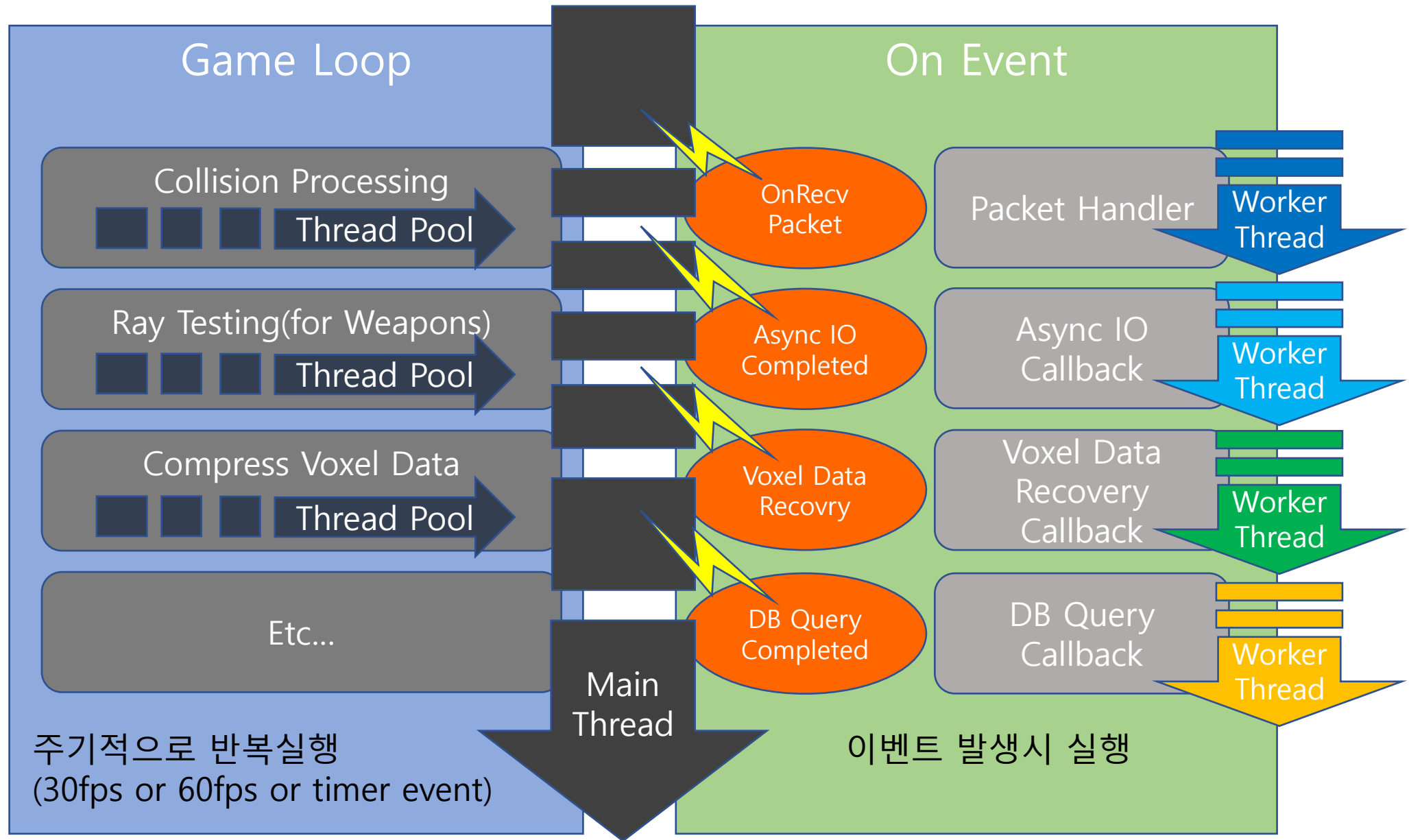




# 서버의 작업 스케줄링 전략

- 1개의 메인스레드 + N개의 보조 스레드 풀
  - 동기화로 인한 성능 저하 방지
  - 유지보수 용이(디버깅, 후임자 인수인계)
- 메인 스레드의 명령에 의해 다수의 워커 스레드가 작업
- 작업을 마치면 워커 스레드가 메인 스레드에 통보

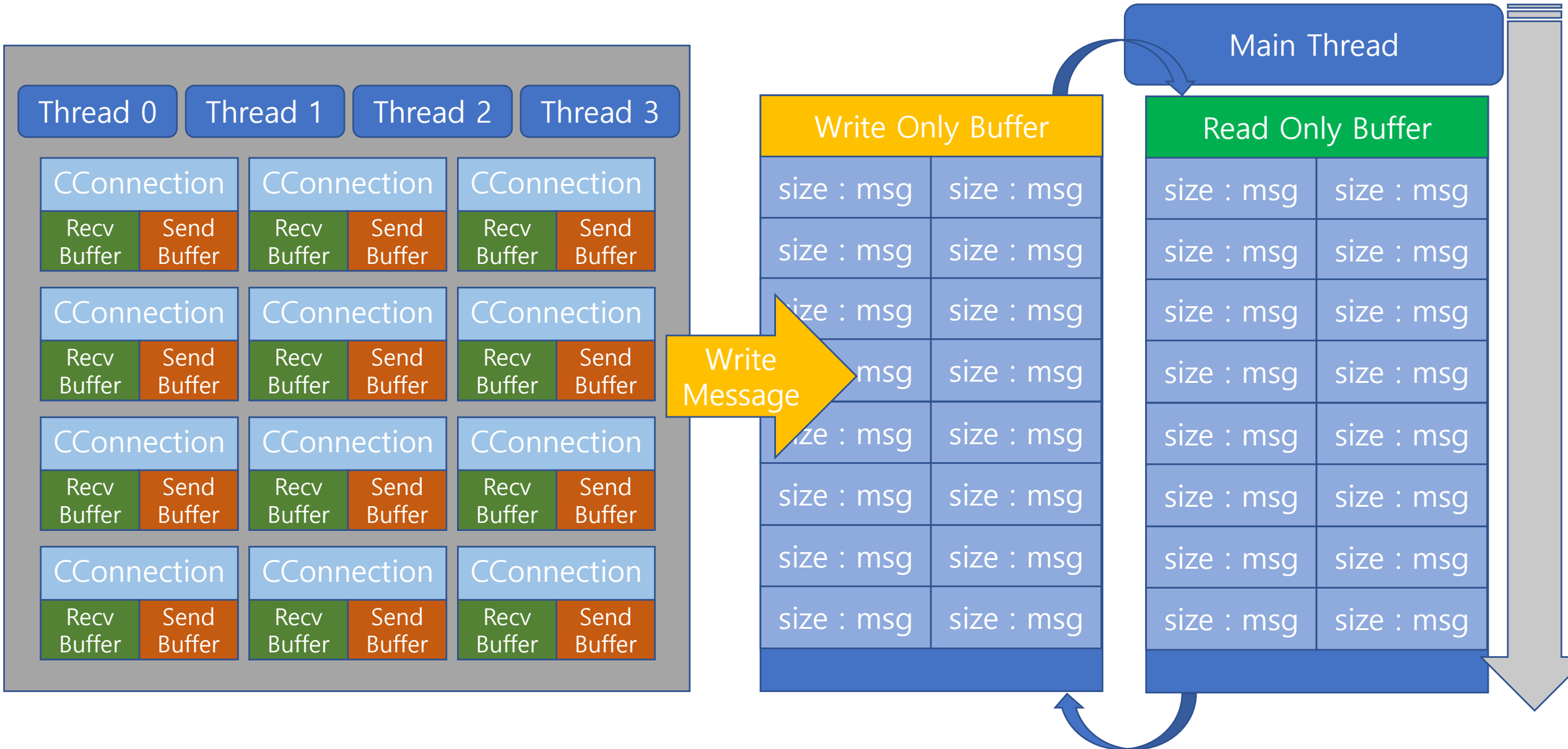
# 서버의 작업 스케줄링



# 패킷 수신 -> 메시지 처리

- 최소 패킷 구조 = size(4 bytes) + body(N bytes)
- I/O 워커 스레드의 메시지 수집과 메인 스레드의 경쟁 상태를 줄인다.
- Double buffering
  - 하나 이상의 패킷이 수집되면 Network측 Worker thread가 쓰기 버퍼에 수집된 패킷을 써넣는다.
  - Main Thread는 패킷 수신이 통보되면 쓰기 버퍼와 읽기 버퍼의 포인터를 swap한다(가벼운 lock사용).
  - Main Thread는 읽기 버퍼의 쌓인 패킷을 처리한다.
  - 처리가 완료되면 쓰기 버퍼와 읽기 버퍼의 포인터를 swap한다.

# 패킷 수신 -> 메시지 처리



# 이동(충돌)처리



<https://youtu.be/JZ2Dza1qhrl>

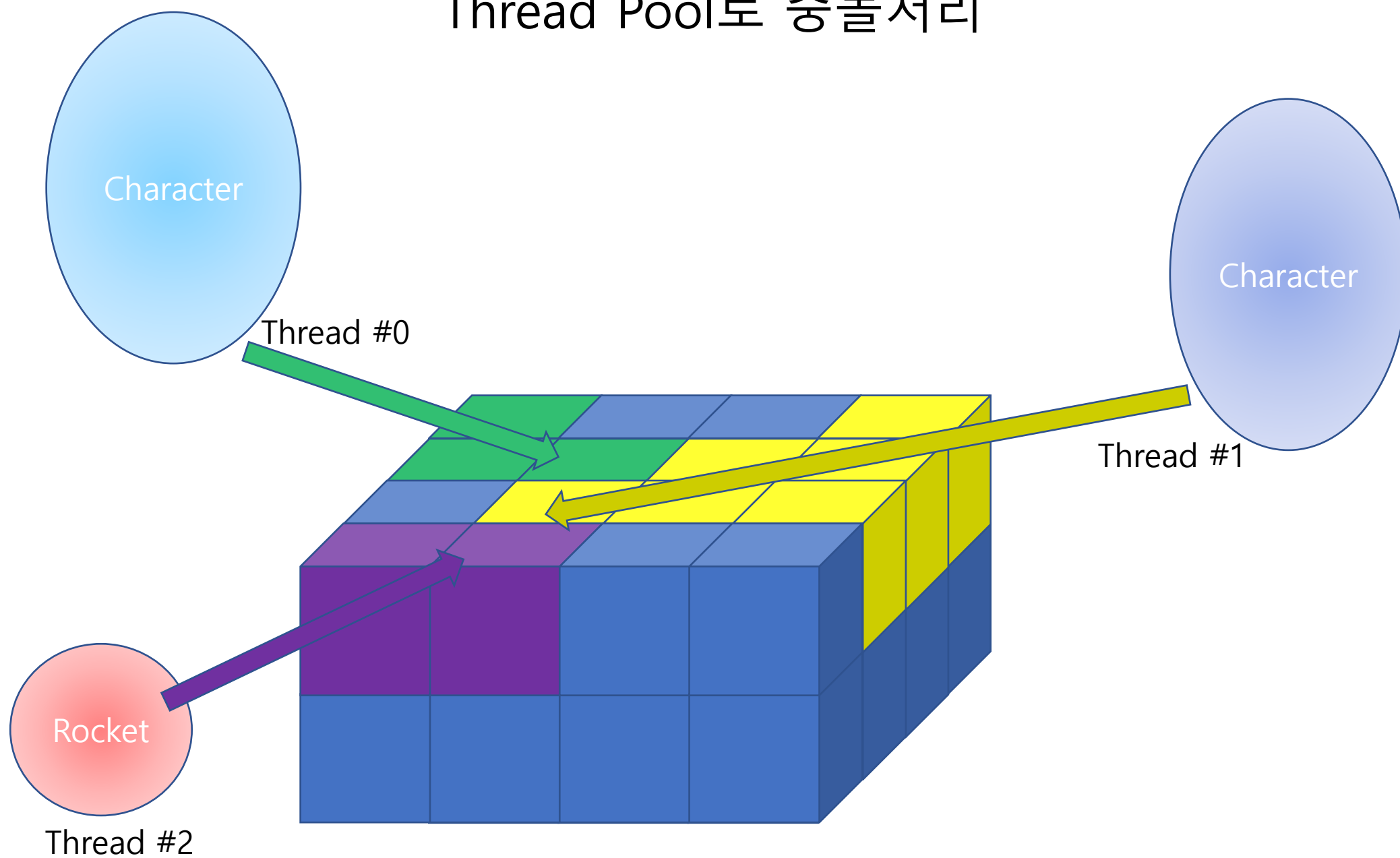
# Voxel Horizon에서의 이동

- 복셀 지형을 따라 뛰어다닌다.
- 일정 각도의 복셀지형과 충돌하면 미끄러진다.
- 일정 각도의 타원체와 충돌하면 미끄러진다.
- 로켓탄은 캐릭터 한마리와 동등하게 처리한다.
- 로켓탄은 복셀 지형이나 타원체나 어디든 충돌하면 폭발한다.

# 서버에서의 이동(충돌)처리

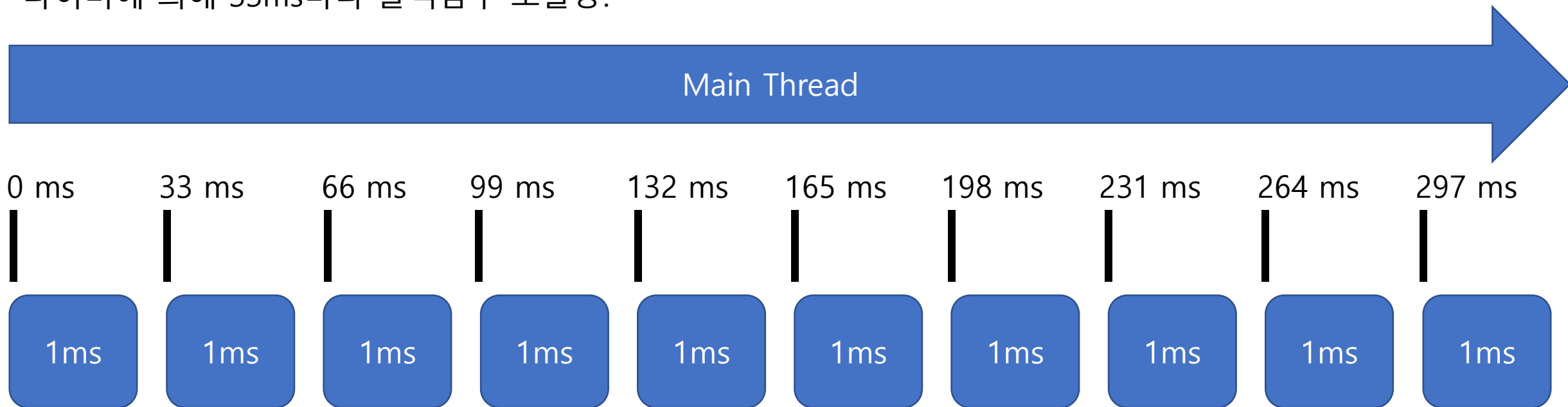
- 클라이언트와 마찬가지로 30fps or 60fps로 처리.
- 키 입력에 대한 순서가 보장되어야 한다.
- 플레이어간 입력 순서가 보장되어야 한다.
- 따라서 완전 동기식으로 처리하되 개별 이동처리 시간을 줄이는 수밖에 없다.
- 처리해야할 캐릭터 수에 비례해서 성능 하락.
- 처리량이 늘어서 응답성이 떨어지는 경우이므로 Thread Pool을 이용하여 병렬처리 한다.

# Thread Pool로 충돌처리





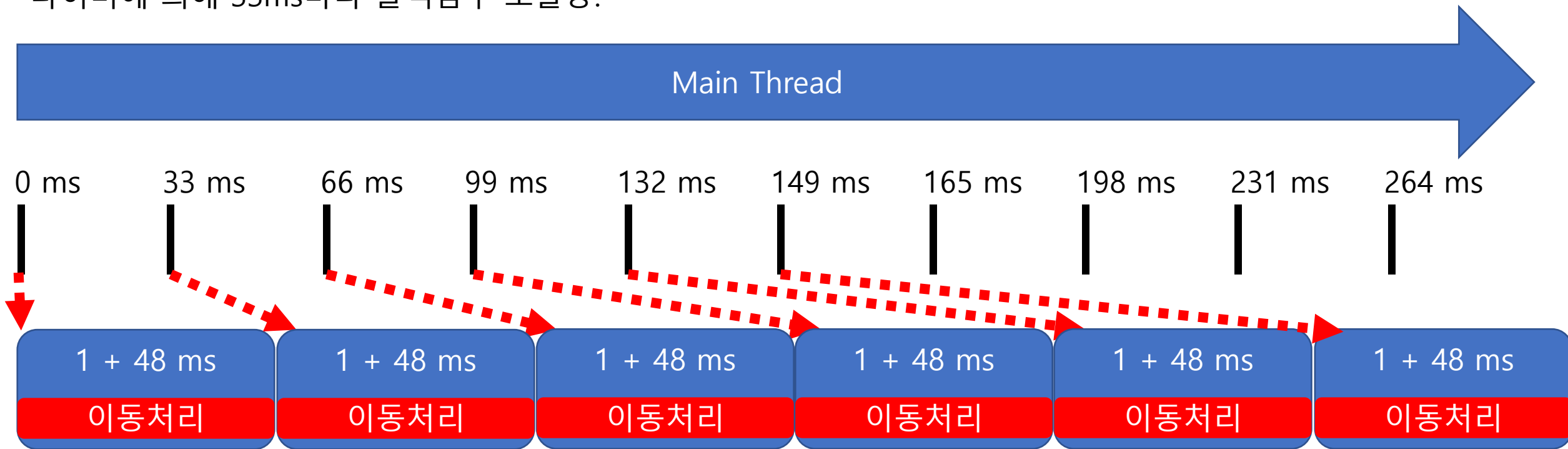
타이머에 의해 33ms마다 콜백함수 호출중.



서버에서 메인 스레드가 게임루프 실행중. 평화로운 상태.

# 1000명분의 이동(충돌)처리 에 48ms가 소요될 경우

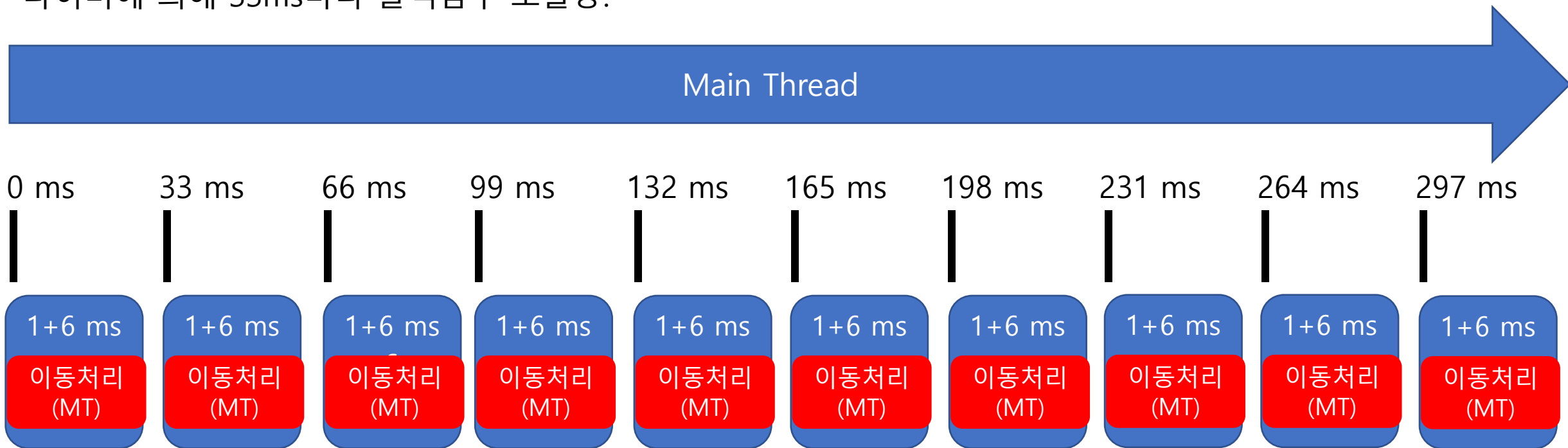
타이머에 의해 33ms마다 콜백함수 호출중.



매 루프 이동처리, 전투판정, 그 외 시간마다 처리할 작업이 계속 지연된다.  
이런 경우 클라이언트/서버간의 위치, 전투판정 결과가 오차가 상당히 커진다.

# 1000명분의 이동(충돌)처리 에 48ms가 소요될 경우

타이머에 의해 33ms마다 콜백함수 호출중.



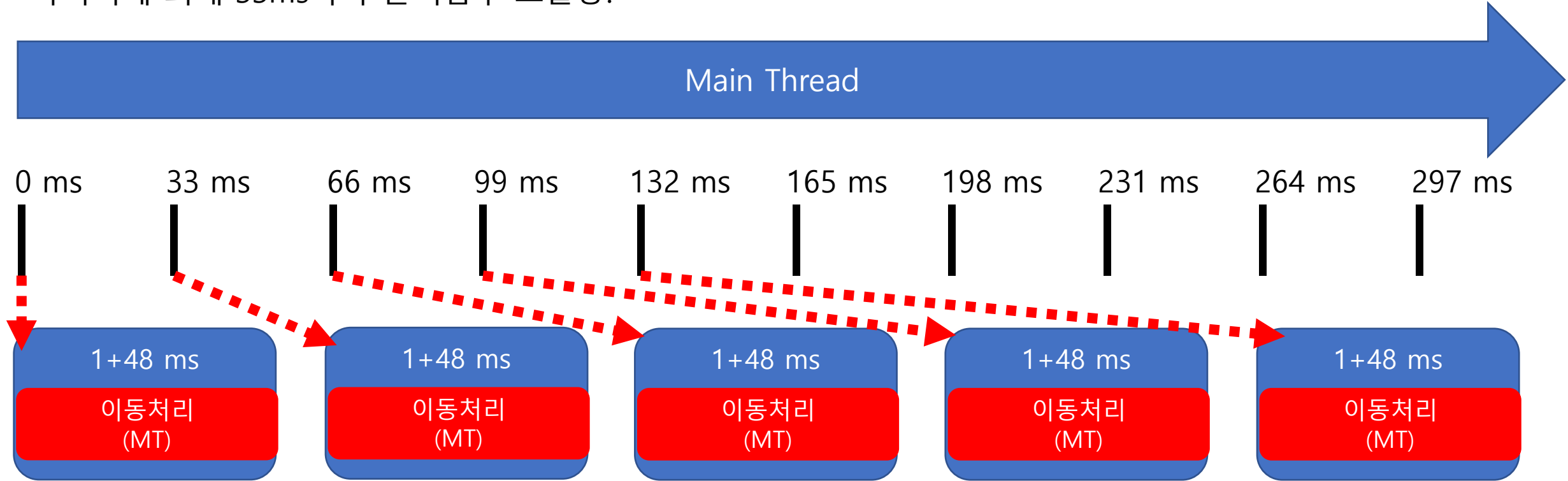
Thread Pool로 병렬처리.

예를 들어 8 threads로 동시에 처리할 경우 (이상적으로 ) 이동처리 48ms -> 6ms로 감소.

매 루프 1+6 ms로 감소.

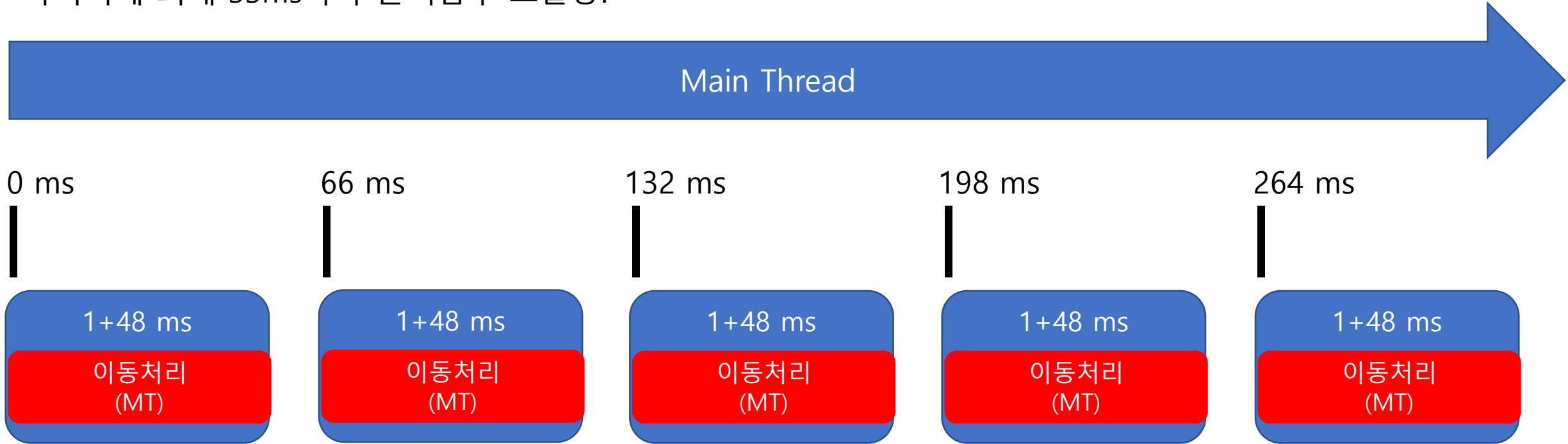
# 멀티 스레드로 처리해도 48 ms가 걸리는 상황

타이머에 의해 33ms마다 콜백함수 호출중.



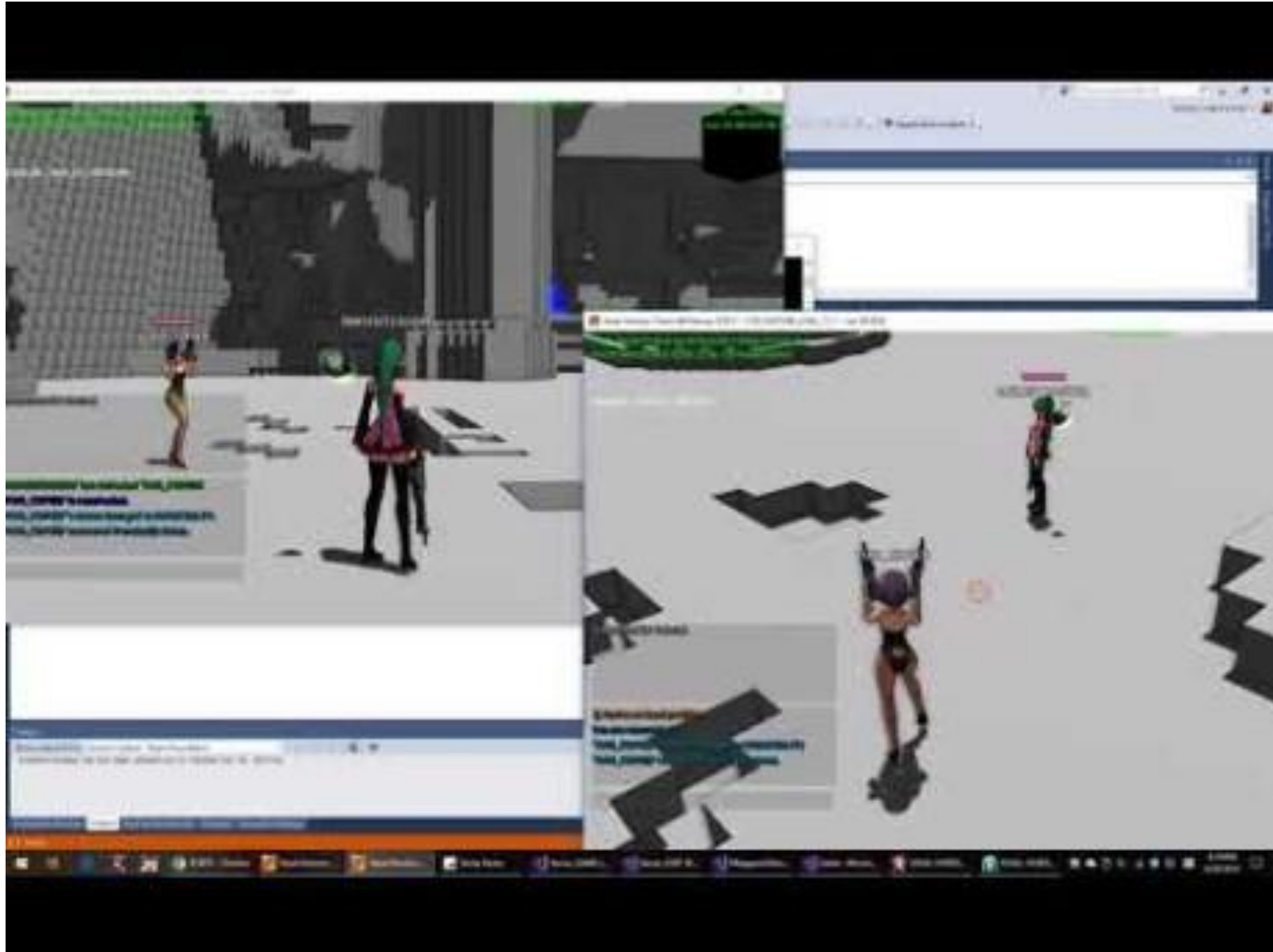
# 멀티 스레드로 처리해도 48 ms가 걸리는 상황

타이머에 의해 33ms마다 콜백함수 호출중.



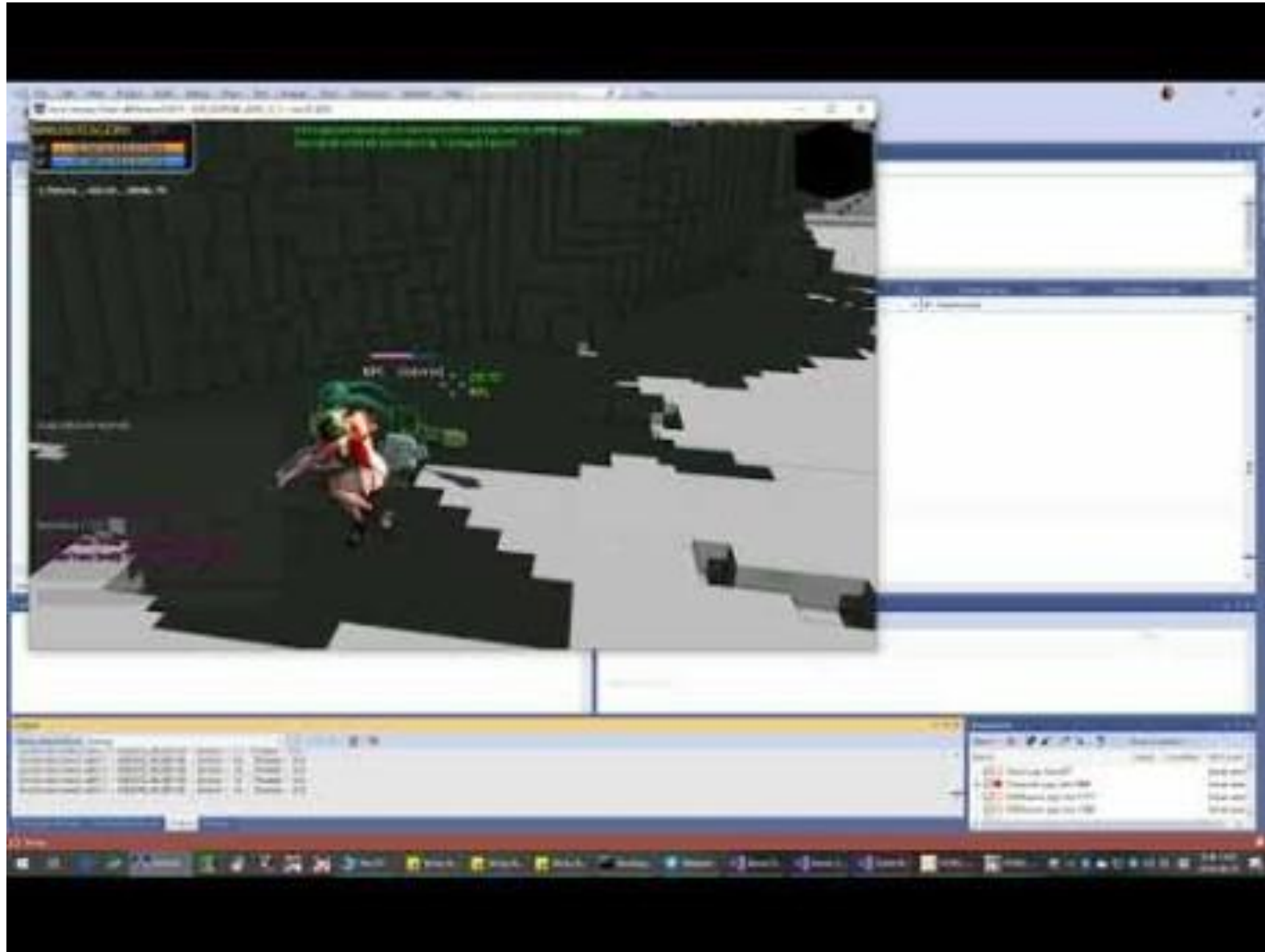
현재 처리 가능한 수준의 프레임 레이트를 판단, 30fps -> 15fps로 게임 프레임 레이트를 낮춘다.  
전체적으로 정밀도가 조금 떨어지지만 게임 플레이에 치명적인 지연은 발생하지 않는다.

# 전투판정



<https://youtu.be/kEEyJ5aSGSo>

# 전투판정



<https://youtu.be/vBTMdtwEPLg>

# Voxel Horizon의 전투

- 총으로 쏜다.
- 로켓탄이 날아가서 충돌시 폭발한다.
- 탄환과 플레이어간의 타격 판정.
- 탄환과 복셀 오브젝트간의 타격 판정.
- 탄환에 의한 복셀 오브젝트의 변형.



# 총에 대한 판정 SMG, 라이플

1. 캐릭터(플레이어와 몬스터)가 총을 들고 있을 경우 서버에서는 30fps로 총의 ray를 갱신.
2. 이때 총의 ray가 복셀 지형에 충돌하는지, 다른 캐릭터에 충돌하는지 테스트. 그 결과를 저장해둔다.
3. 플레이어의 요청에 따라 ray에 걸친 캐릭터가 있는지 확인.클라이언트의 요청이 사실로 판별되면 Hit처리.
4. 서버가 그 즉시 전투판정을 내려도 되지만 이 경우 그렇게까지 적극적일 필요가 없으므로 ("내가 적을 쏘서 맞췄지만 맞춘게 아닌걸로 하고 싶어" 라는 바보같은 해킹은 하지 않으므로) 서버는 매 프레임마다 ray테스트 결과만 유지한다.



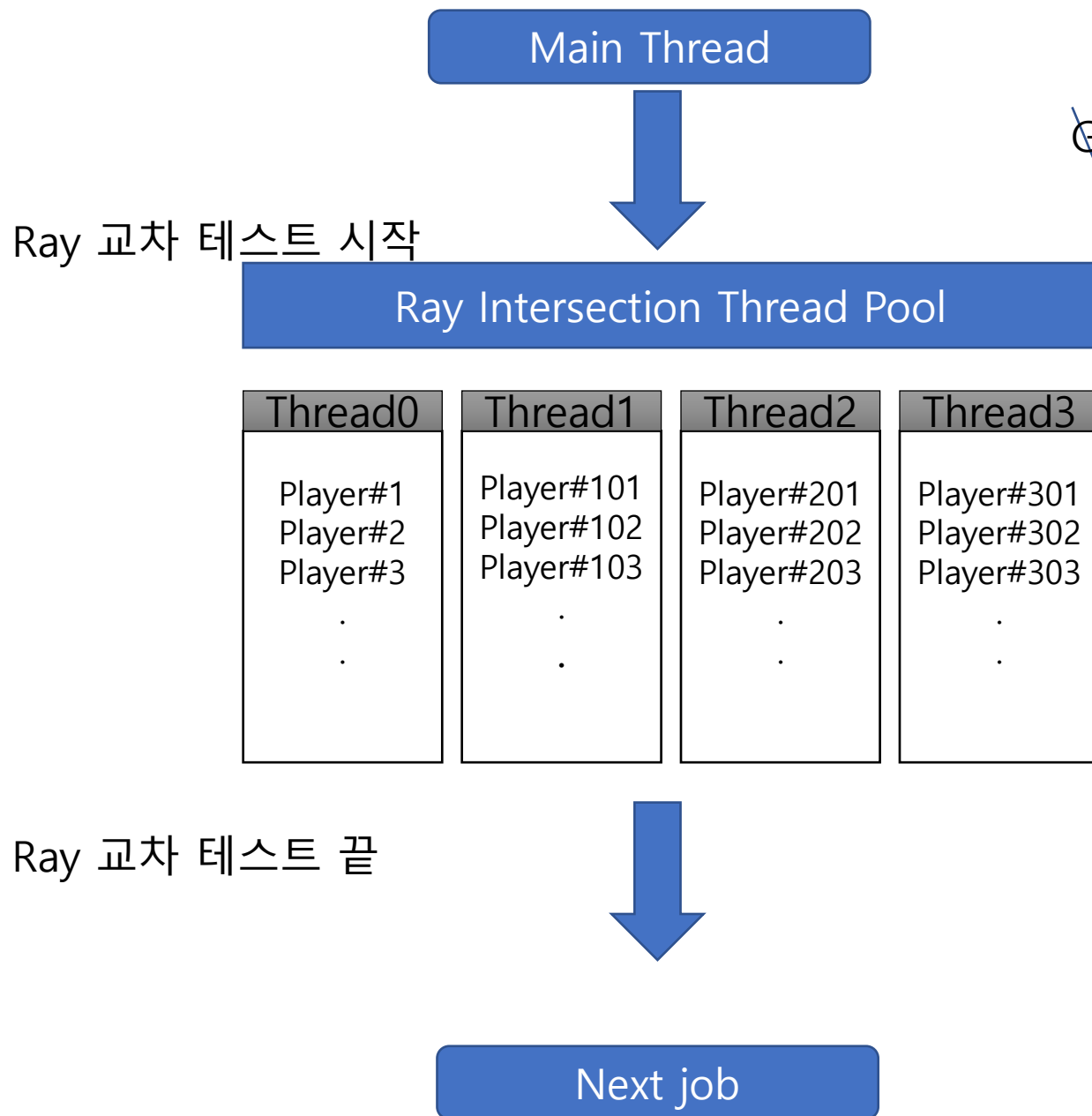
# 로켓런처에 대한 판정

1. 로켓탄이 캐릭터 혹은 지형지물에 충돌하면 폭발한다.
2. 서버와 클라이언트 모두 동일하게 작동하지만 클라이언트는 폭발시 아무것도 하지 않는다. 애니메이션 처리만 한다.
3. 서버의 경우 로켓탄이 폭발하면 폭발 위치에서 폭발 범위에 들어가는 캐릭터 오브젝트들을 찾는다.
4. 찾은 캐릭터들에 대해 거리별로 다이렉트 데미지와 스플래시 데미지를 적용한다.
5. 폭발 영향 범위에 복셀 오브젝트가 포함된 경우, 해당 복셀 오브젝트를 변형시킨다.
6. 클라이언트에 폭발 위치와 영향 범위, 변형된 복셀 오브젝트의 목록을 전송한다.
7. 클라이언트는 폭발 위치와 영향 범위에 따라 복셀 오브젝트를 변형시킨다. 서버와 클라이언트가 똑같은 코드를 사용하므로 입력(위치, 반지름)이 같다면 복셀 오브젝트의 변형 결과는 똑같다.



# 전투판정

- 로켓의 유도 , 타격은 캐릭터 이동(충돌)처리와 동일하다.
- GUN타입의 무기들은 총구로부터의 ray가 존재한다.
- Ray와 지형/캐릭터의 충돌처리가 사실상의 전투판정이다.
- 다수의 삼각형(복셀지형)/타원체에 대한 수학연산이 필요하다.
- 총을 들고 있는 플레이어/NPC가 수가 증가하면 당연히 소요시간이 길어진다.
- 이동(충돌)처리와 마찬가지로 Thread Pool을 이용한 병렬처리를 사용한다.



Game Loop

1. Core 개수만큼 충돌처리 스레드를 만들어둔다(Thread Pool)

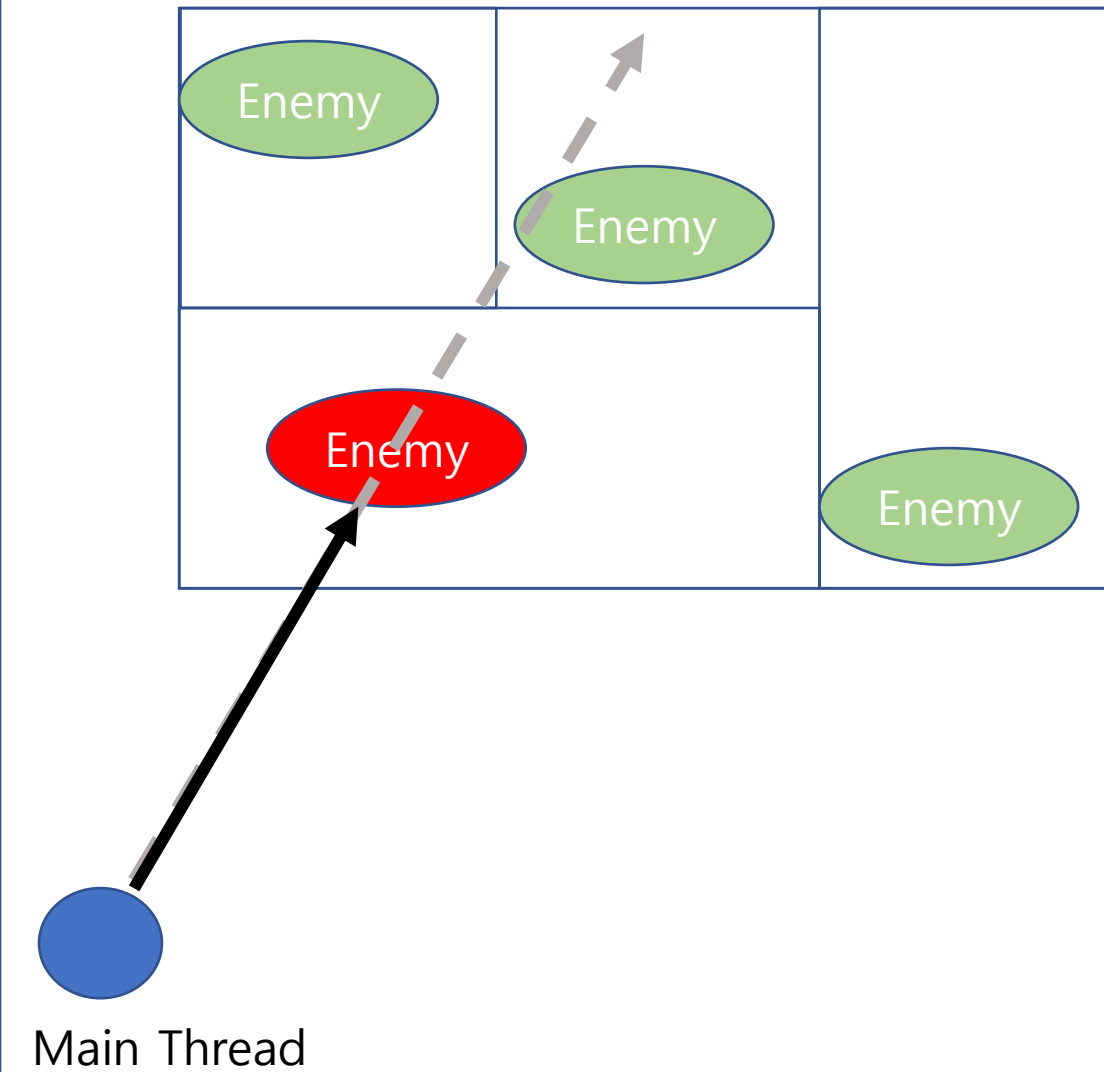
2. Ray 테스트 타이밍이 되면 스레드풀의 스레드들이 플레이어들이 들고 있는 총의 Ray에 걸치는 오브젝트와 지형지물에 대한 충돌점을 계산한다.

3. 메인스레드는 스레드풀의 모든 스레드가 계산을 완료할때까지 대기한다.

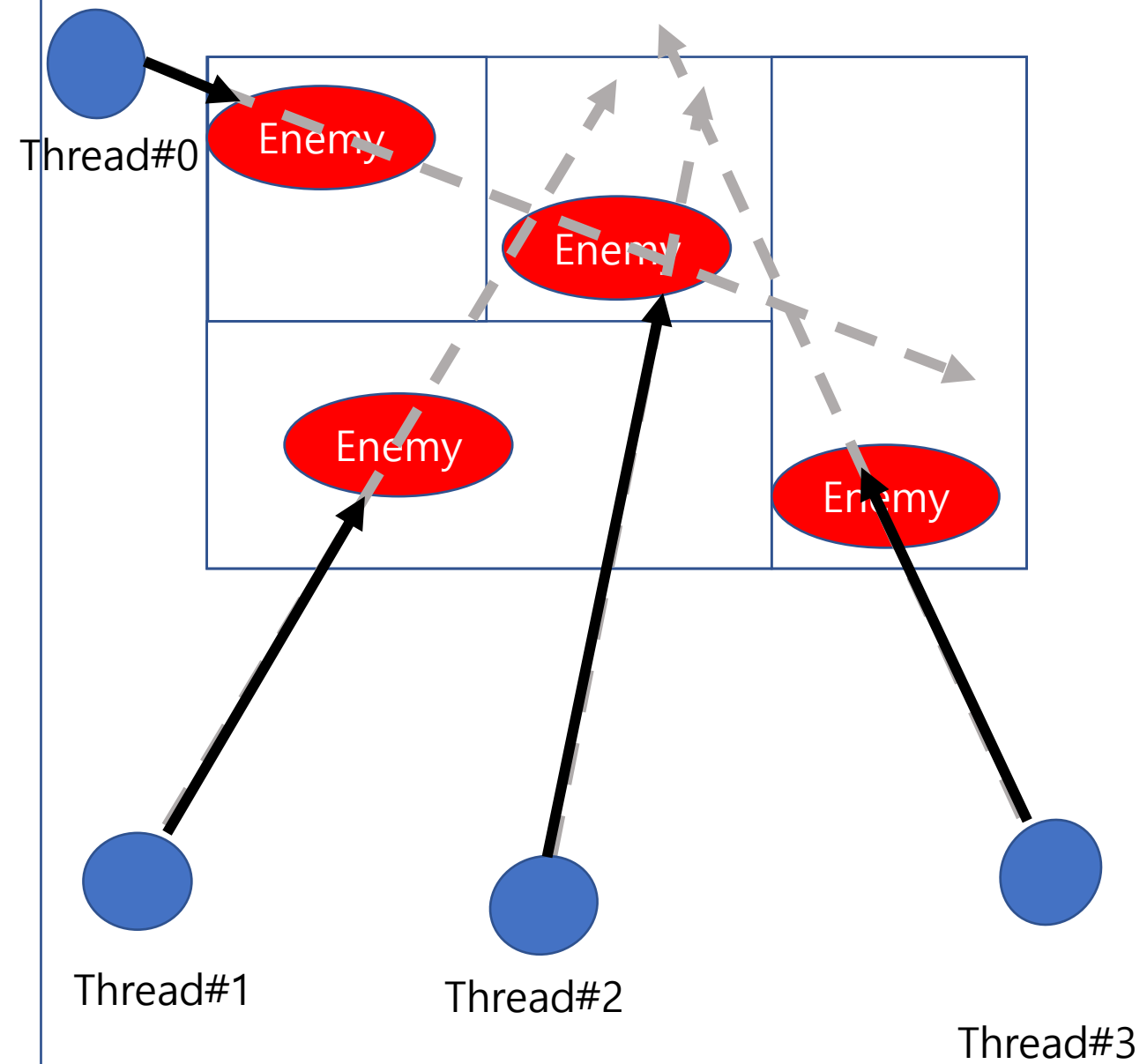
4. 모든 충돌처리가 끝나면 메인스레드가 완료된 Ray Intersection 정보를 갱신한다.

# 총 들고 다닐때

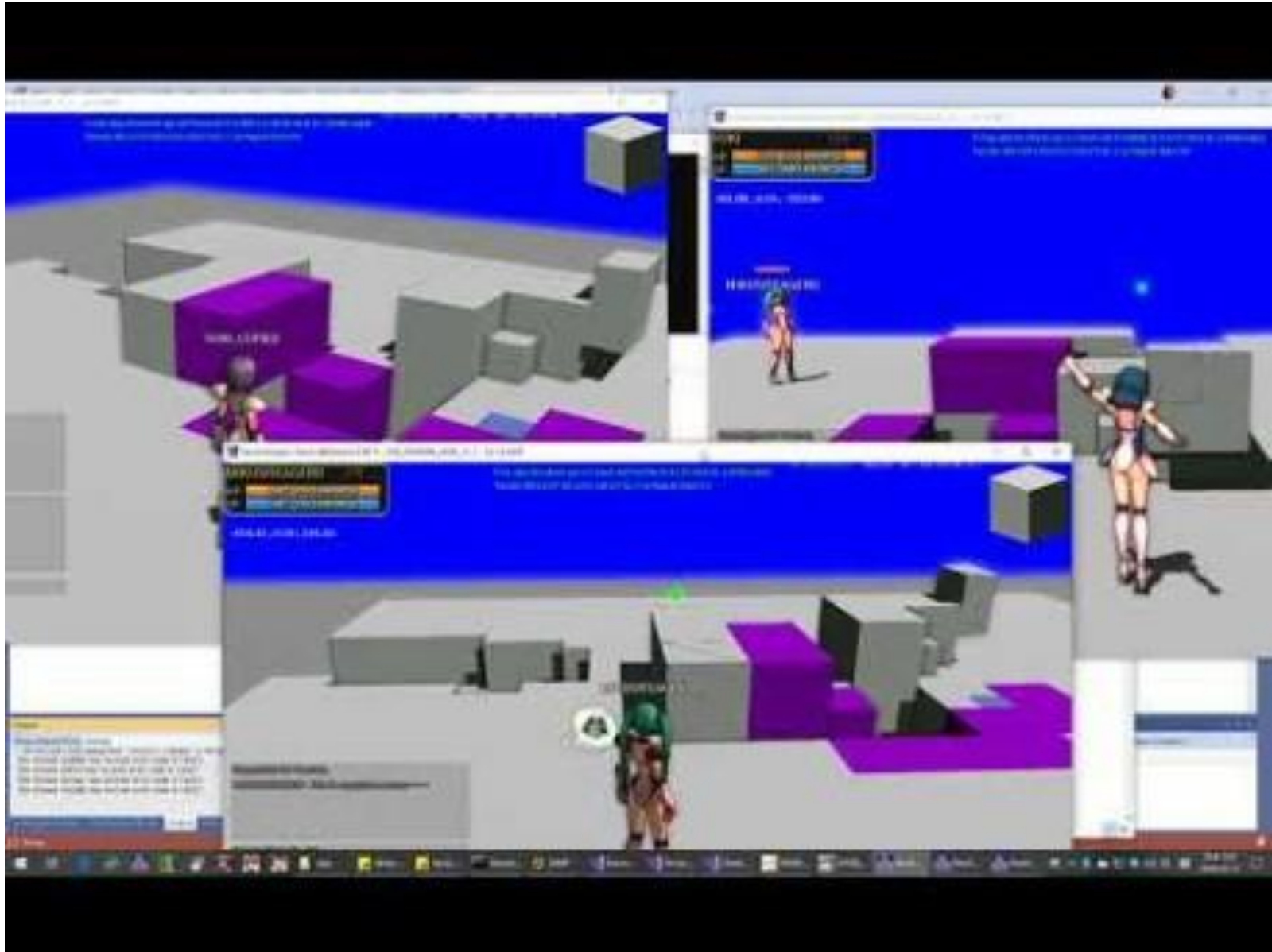
## Single Thread



## Multi Threads



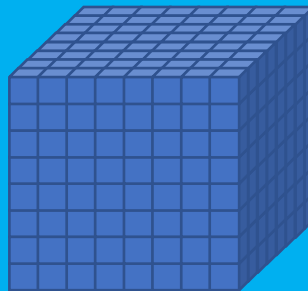
# 복셀 지형 편집



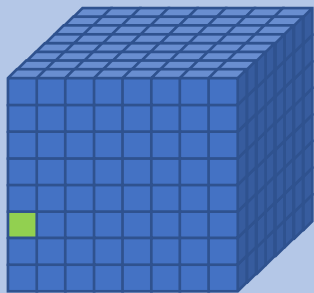
<https://youtu.be/JwlTmDvCKFk>

"ObjPos(1,1,1), VoxelPos(0,2,0)  
위치에 복셀 하나를 제거  
(remove)해줘."  
서버의 응답을 받을때까지 메  
모리상의 복셀 데이터를 건드  
리지 않음.

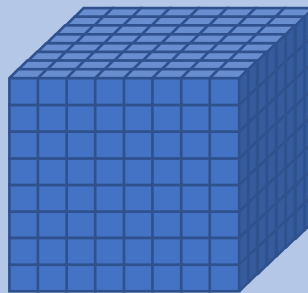
Server



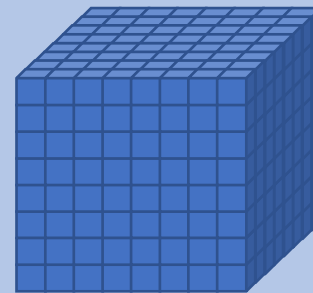
Client



Client

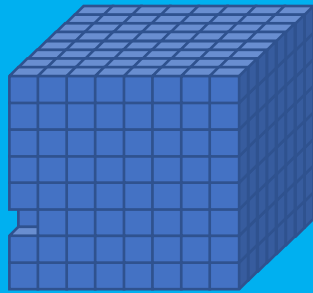


Client

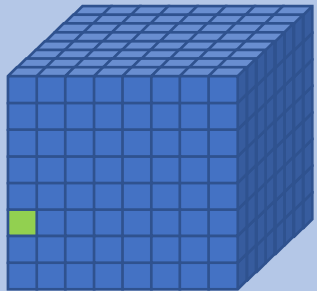


ObjPos(1,1,1), VoxelPos(0,2,0) 위치  
에 복셀이 실제로 존재하나?  
Player#1 은 이 복셀 오브젝트에 대  
해 편집권한을 가지고 있나?  
OK라면 서버측의 복셀 데이터 변경

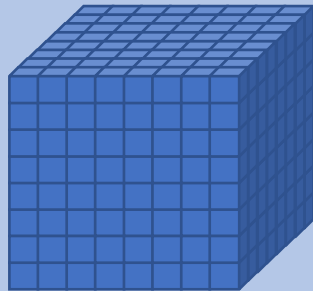
Server



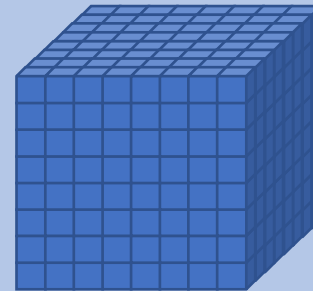
Client



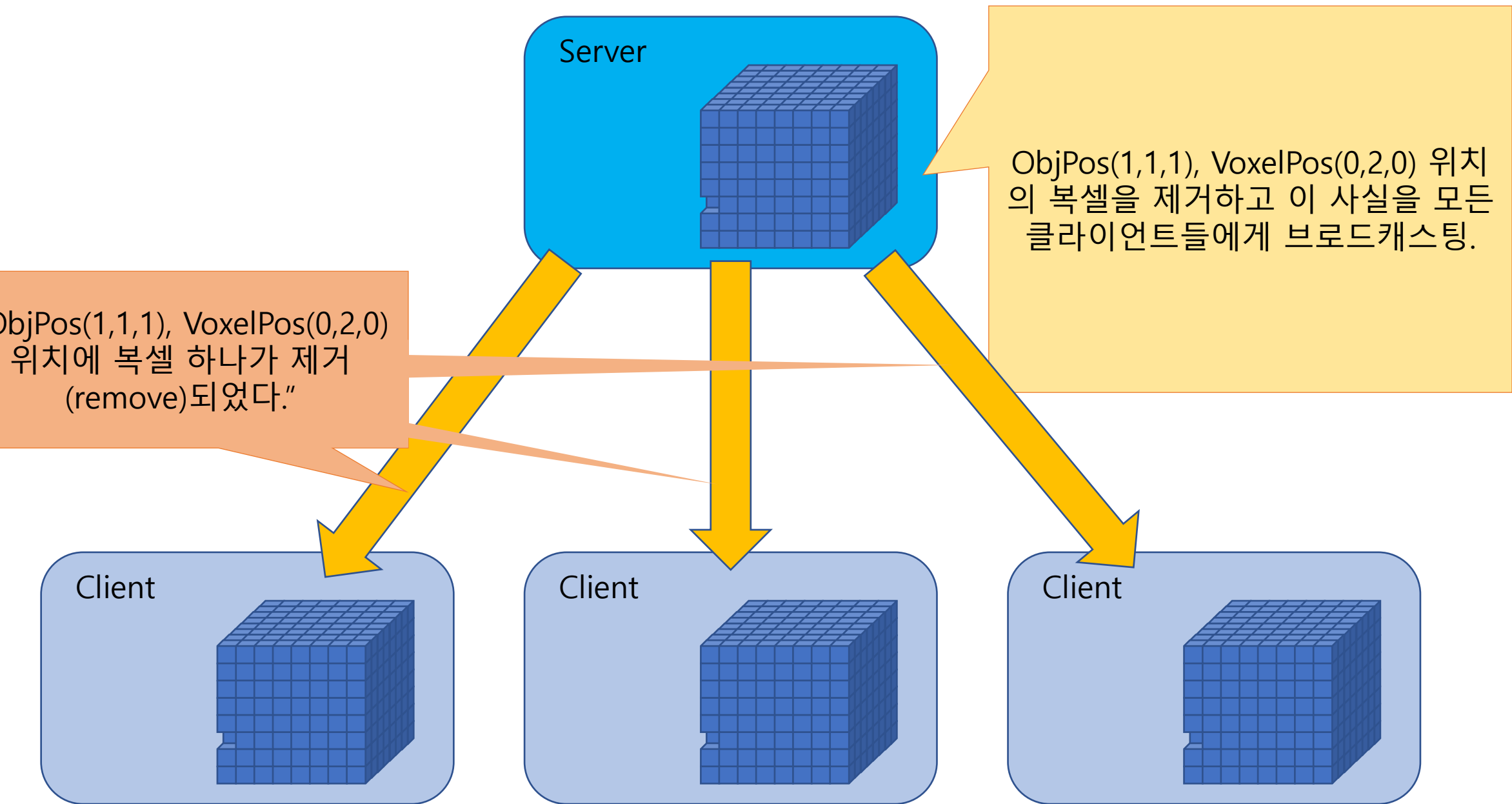
Client



Client



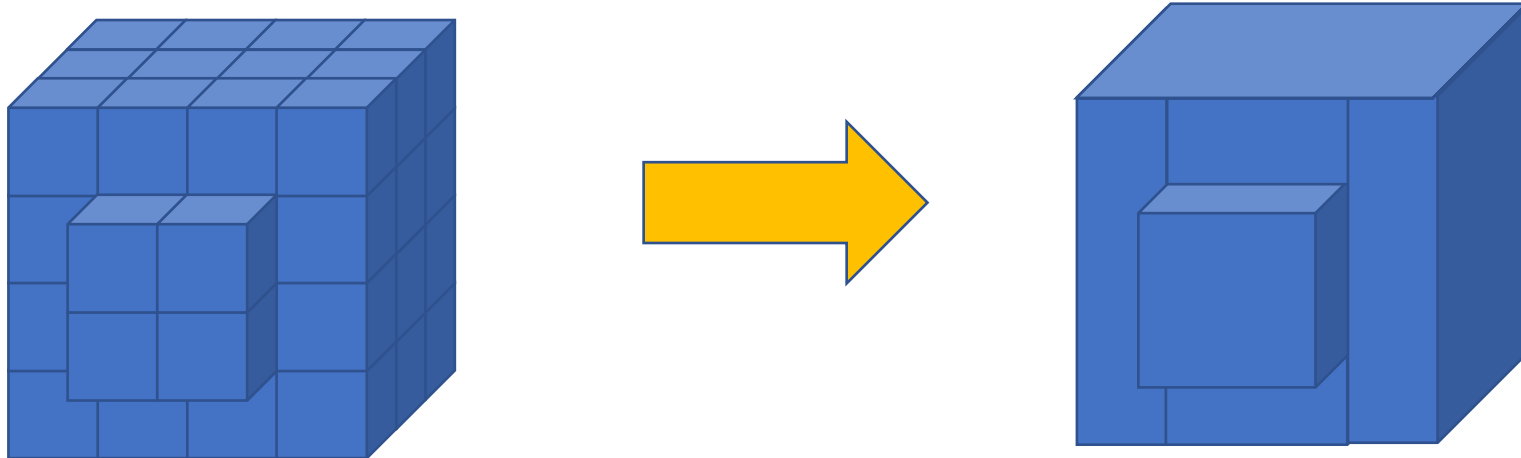




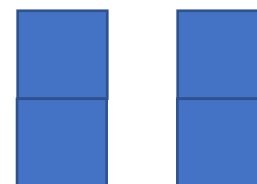
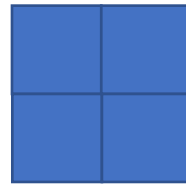
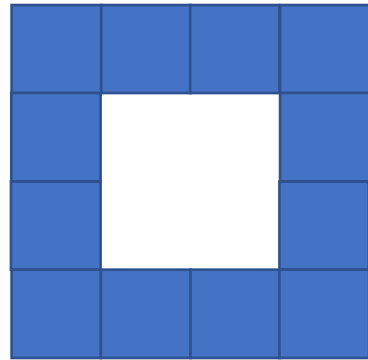
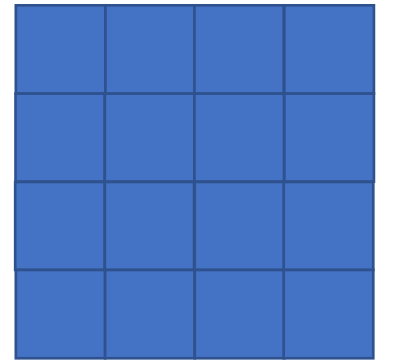
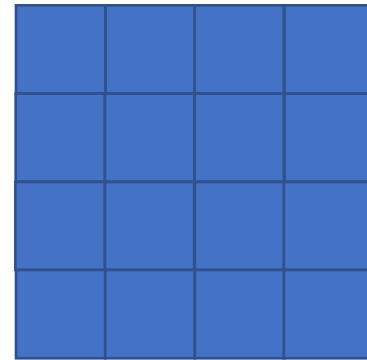
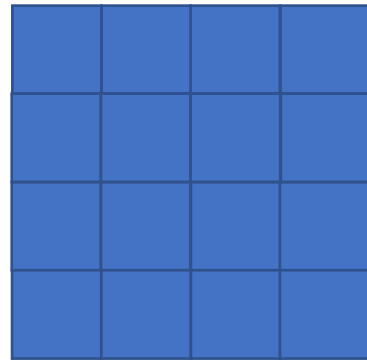
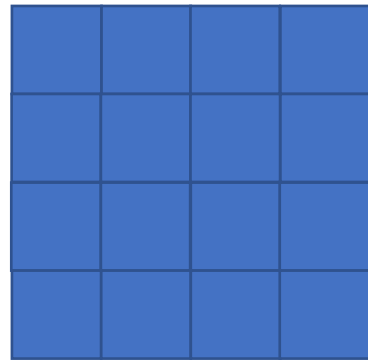
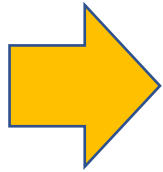
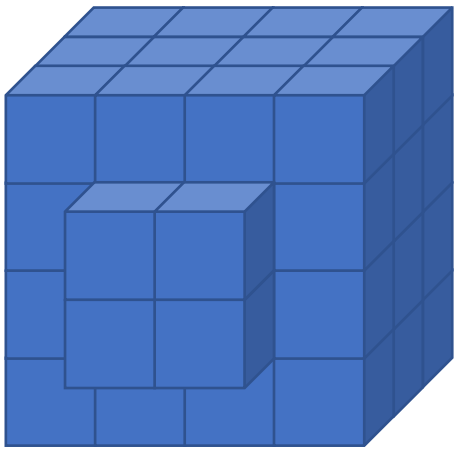
# 복셀 편집 이벤트가 발생하면...

- 복셀 오브젝트의 bit table 수정(1bit 단위 3차원 배열과 같다).
- 변경된 bit table을 가지고 충돌 처리용 삼각형 매시 빌드.
- 이 삼각형 매시를 최적화.
- 로켓탄이 폭발해서 복셀 오브젝트가 변형된 경우도 동일.
- 결코 가벼운 연산이 아니다!

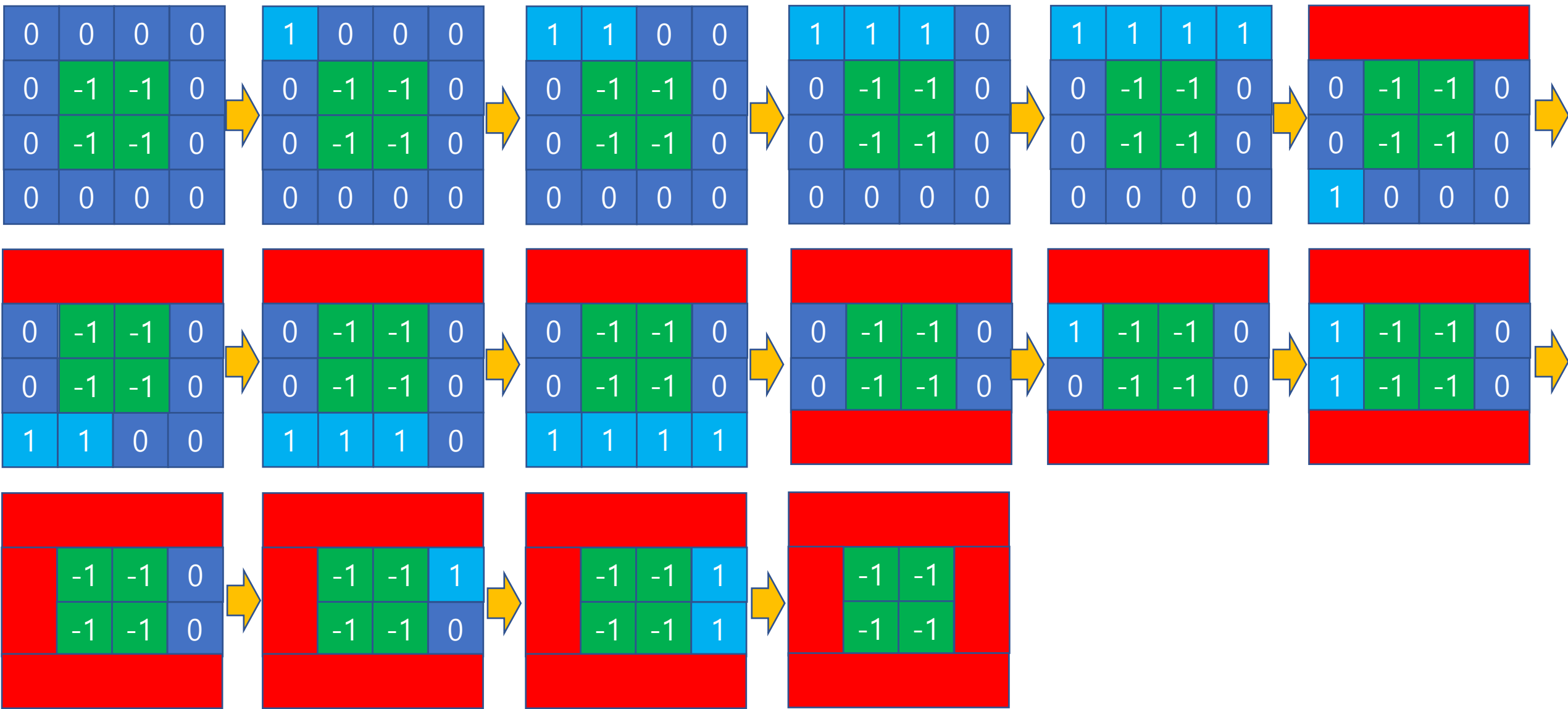
충돌처리를 위해 인접한 면들을 병합하여 최적화된 메시로 변환.



사각형들을 분해해서 같은 평면 방정식에 따라 그룹화



## 2D 비트맵상에서 x,y축으로 한칸씩 성장



# 복셀 편집 이벤트 처리 전략

- 플레이어의 요청에 의해서 처리된다.
- 동일한 복셀 오브젝트에 대해 여러 플레이어의 편집 요청이 들어올수 있으므로 순서는 보장되어야 한다.
- 따라서 비동기 처리 불가.
- 의존성이 없는 다수의 항목을 동시 처리하는 케이스가 아니므로 Thread Pool도 사용 불가.
- 코드 최적화 외에 다른 방법은 없다.

# 복셀 데이터 스트리밍 - 복셀 데이터 압축



<https://youtu.be/HpyTC-qInMI>

# 복셀 데이터 압축 for 스트리밍

- 미리 배포되는 정적인 맵 데이터가 없다.
- 월드의 지형은 항상 변화한다.
- 로그인 할 때, 위치가 변경될 때 주변의 지형을 서버로부터 스트리밍 한다.
- 패킷량을 줄이기 위해 복셀 데이터는 **반!드!시!** 압축되어야 한다.
- 다만 복셀 지형의 변형이 일어났다 하더라도 그 즉시 스트리밍을 해야하는 것은 아니기 때문에 당장 압축을 할 필요는 없다. 즉 **비동기**로 처리해도 된다.

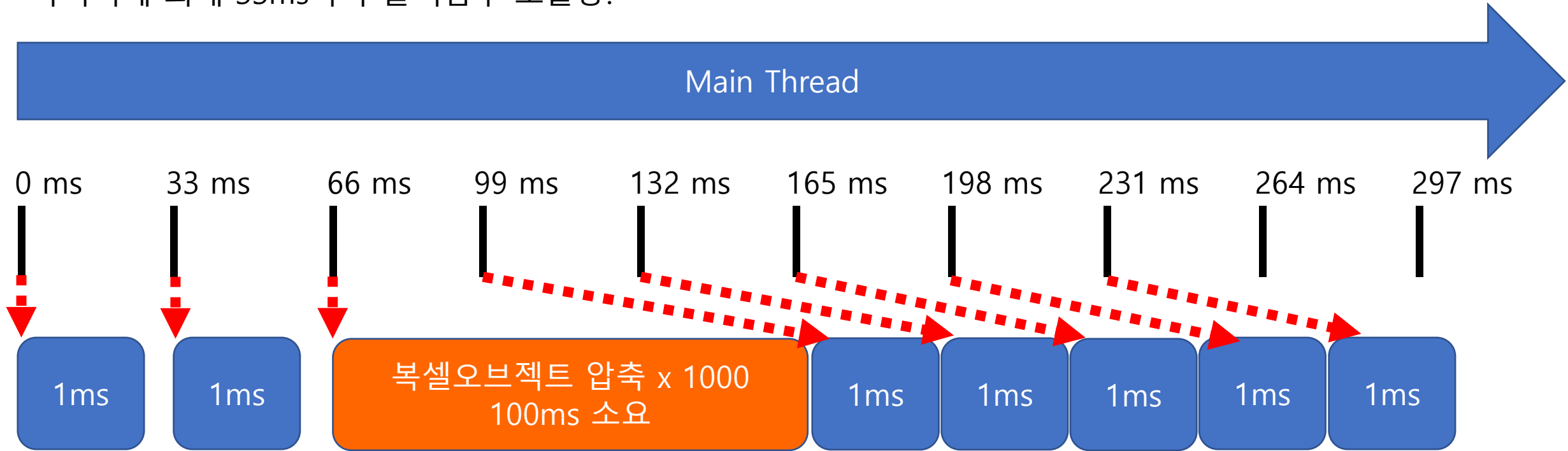


# 복셀 데이터 압축 전략

- Thread Pool을 이용한 병렬처리
- 코드 최적화
- 싱글 스레드로 비동기 처리
- 복셀 편집 이벤트가 발생한 경우
  - 무기에 의한 파괴/복셀 추가/색상변경/삭제/정밀도 변경 등
  - 압축 대기 목록에 추가
- 플레이어의 위치 변경에 의해 복셀데이터를 전송해야하는 순간까지 압축처리를 하지 못한 경우, 해당 오브젝트는 그 즉시 압축.

# 다수의 복셀 오브젝트들을 압축할 때

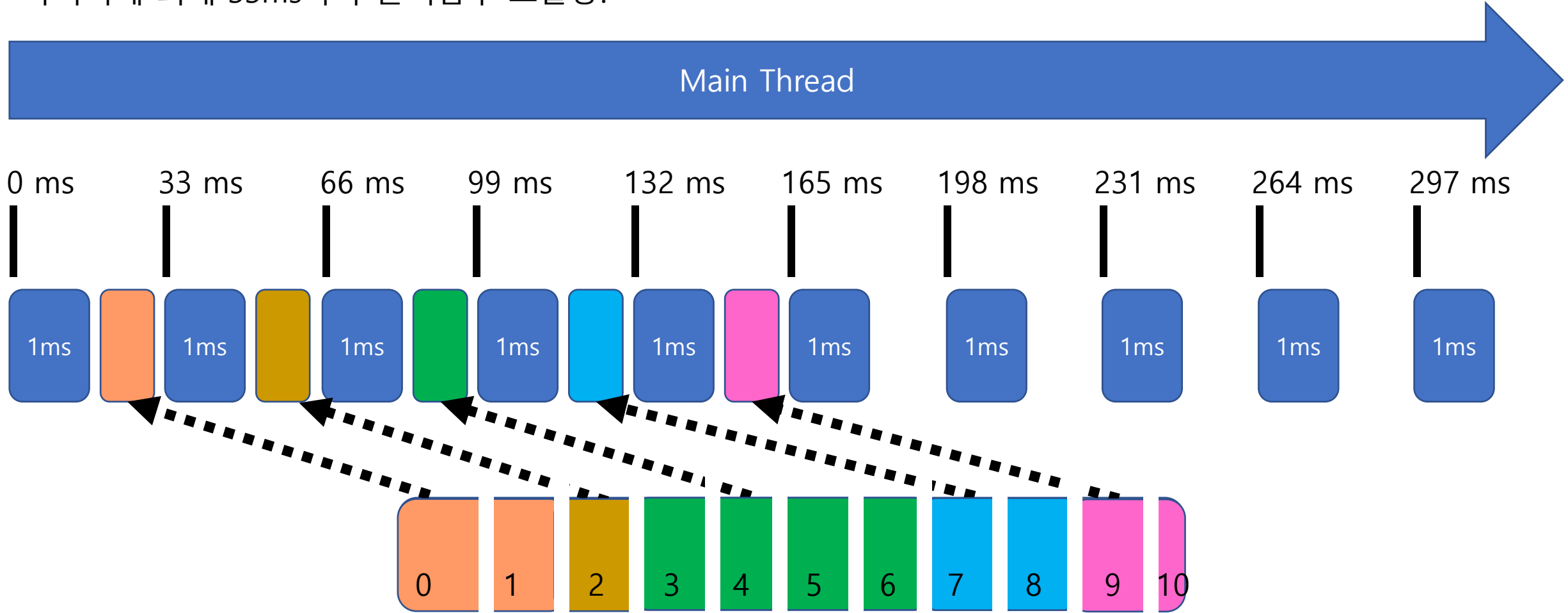
타이머에 의해 33ms마다 콜백함수 호출중.



1000개의 복셀 오브젝트를 압축할때 100ms 걸린다고 하면 , 이후 매 프레임 수행할 작업들이 계속 밀린다.

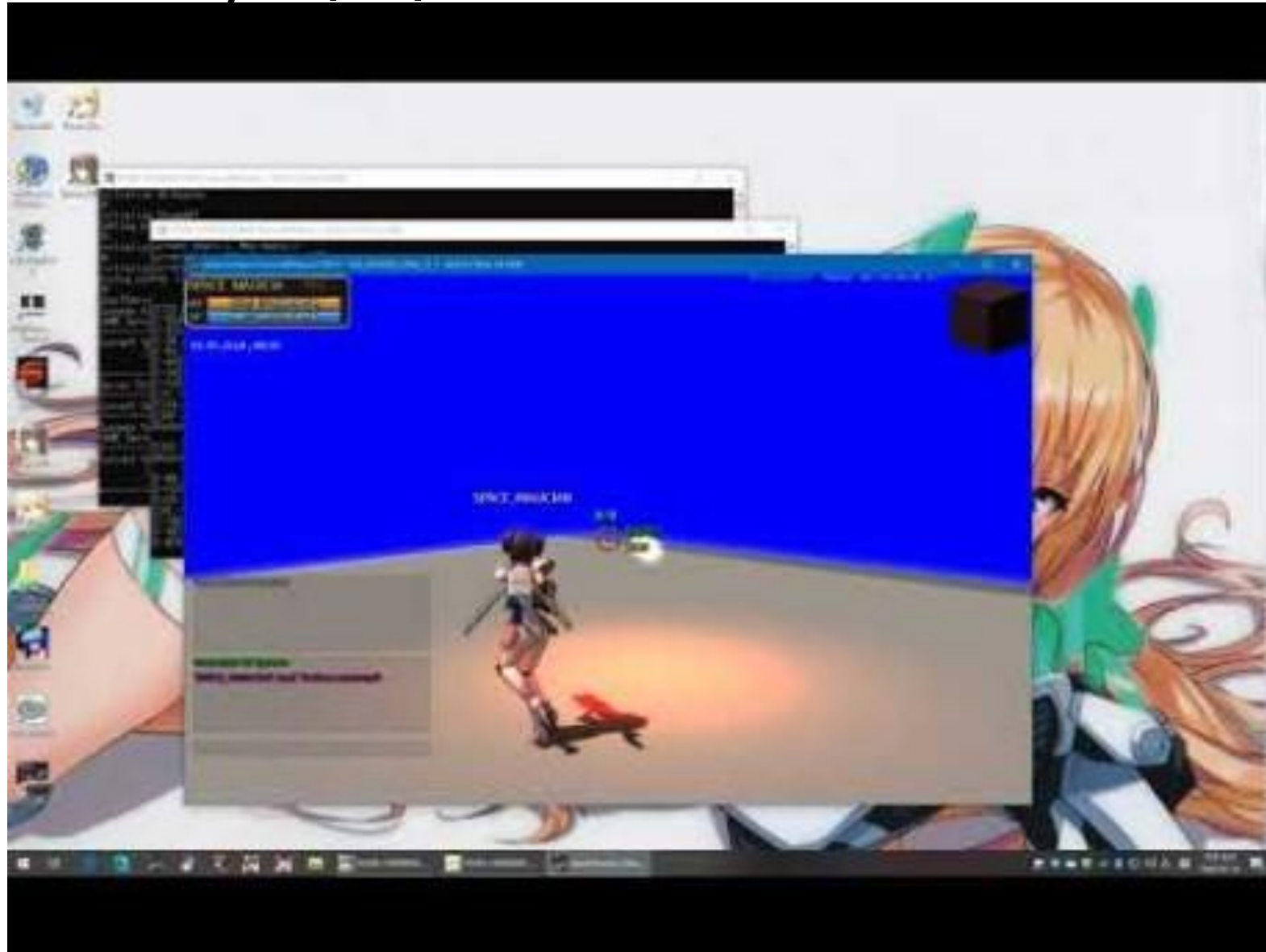
# 다수의 복셀 오브젝트들을 압축할 때

타이머에 의해 33ms마다 콜백함수 호출중.



리스트에 쌓여있는 오브젝트들의 압축.  
주어진 시간을 초과한 경우 빠져나감. 다음 프레임에 처리.

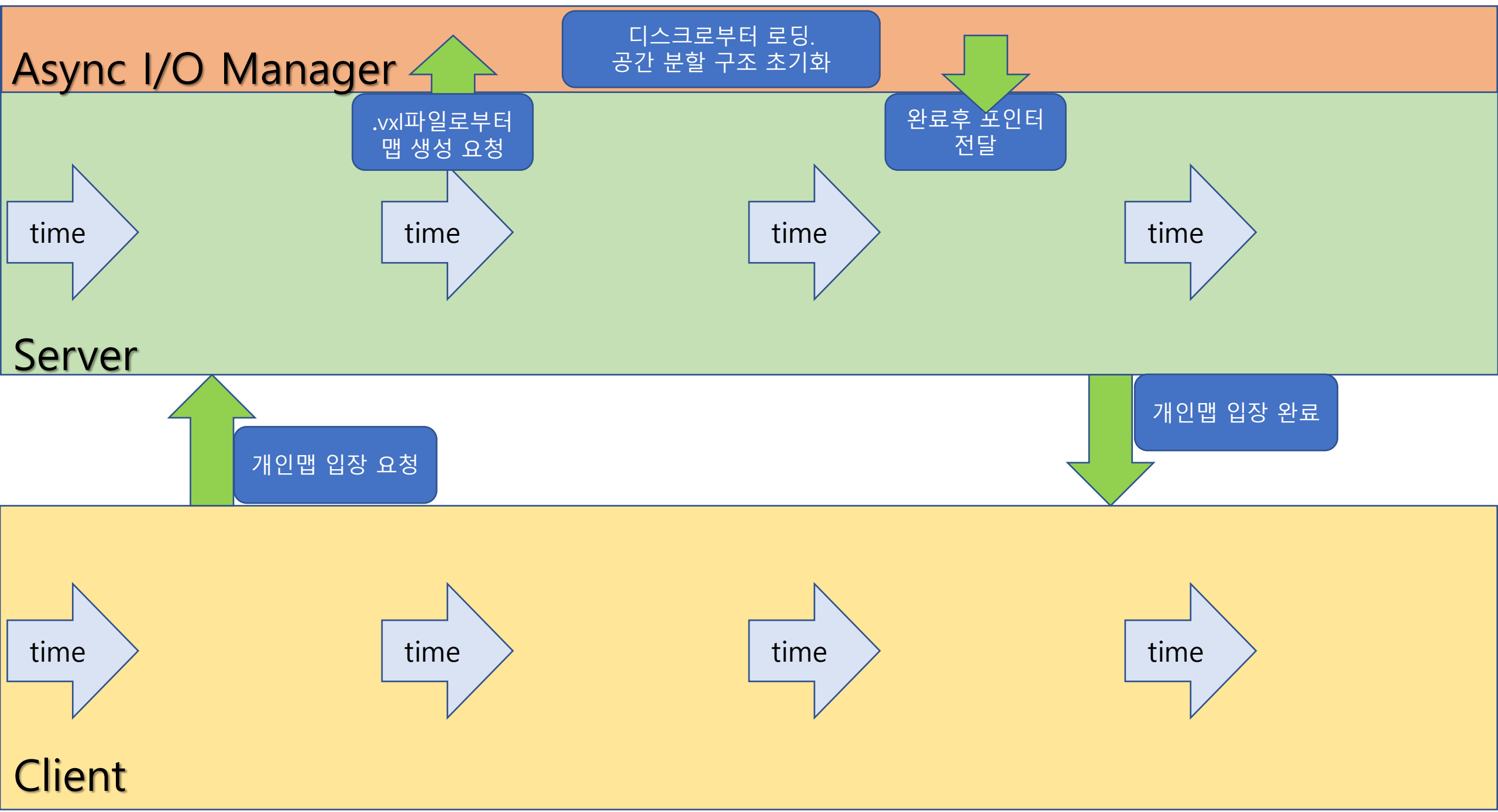
# 개인맵 로드/세이브



<https://youtu.be/CrAfsWSpCkM>

# 개인맵 로드/세이브

- 플레이어가 자신의 개인맵에 들어가려고 할 경우 파일로부터 개인맵을 읽어야 한다. 처음 들어가는 경우 새로 생성해야한다.
- 플레이어가 개인맵에서 나갈 경우 개인맵을 저장해야한다.
- 디스크 I/O가 발생하므로 느리다. (메모리 연산의 지연시간과 차원이 다르다!)
- 다수의 유저가 동시다발적으로 개인맵에 들어가거나 나갈 경우 메인 스레드의 응답성이 크게 저하될 수 있다.
- 별도의 워커스레드로 작동하는 Async I/O Manager를 만들어서 여기에 디스크 I/O작업을 전담시킨다.

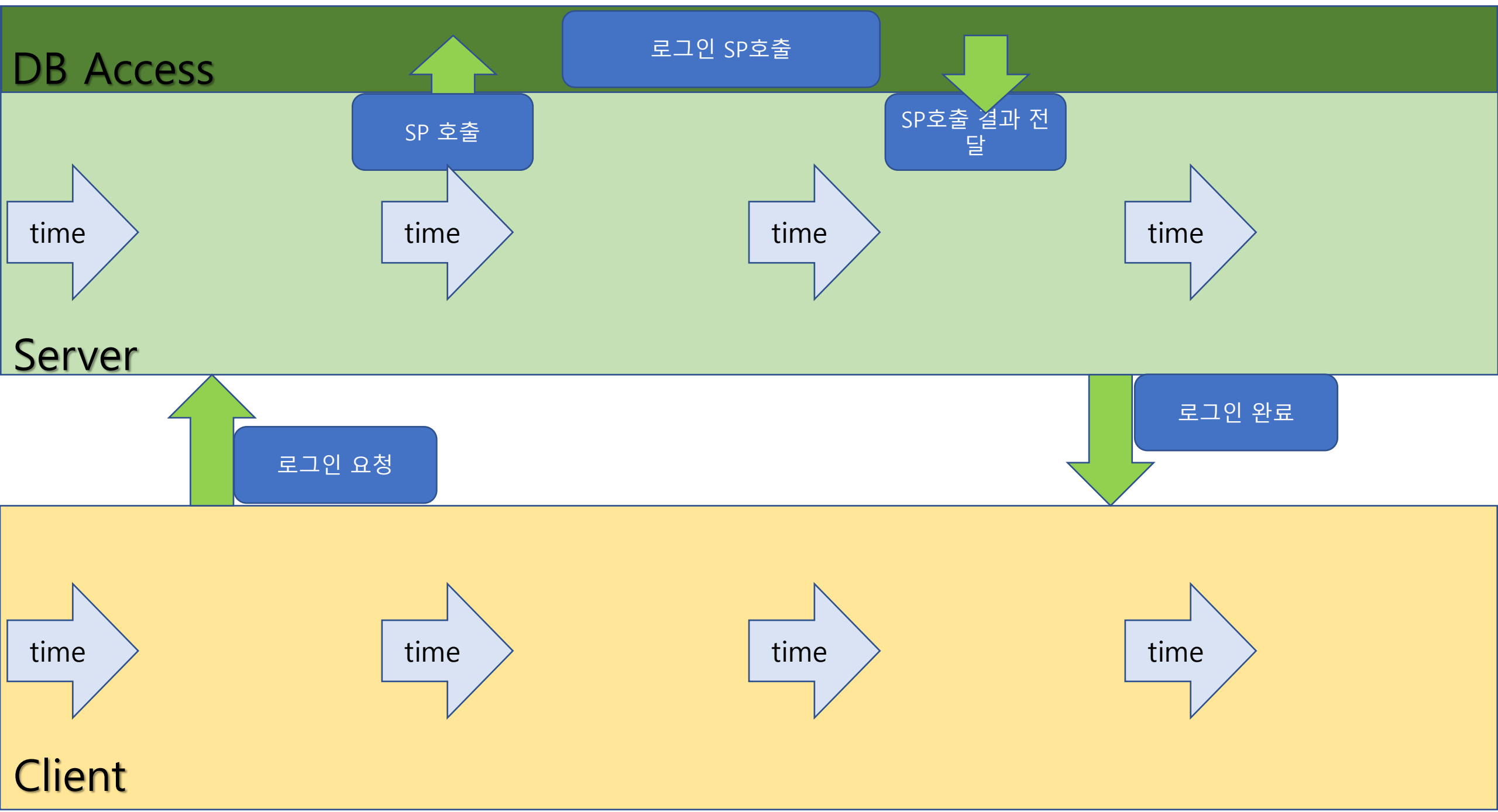


# DB쿼리

# 비동기 DB쿼리

- DB에 쿼리 후 응답 수신 후 메모리에 업데이트
  - 로그인, 아이템 획득 등
- 서버의 메모리 업데이트 후 DB에 쿼리(저장)
  - 총알 소모, 캐릭터 데이터 세이브, 기타 등등
- 어느쪽이든 비동기 처리
- 절.대.로 DB의 게임서버의 패킷 처리 스레드가 DB의 응답을 대기해서는 안된다.

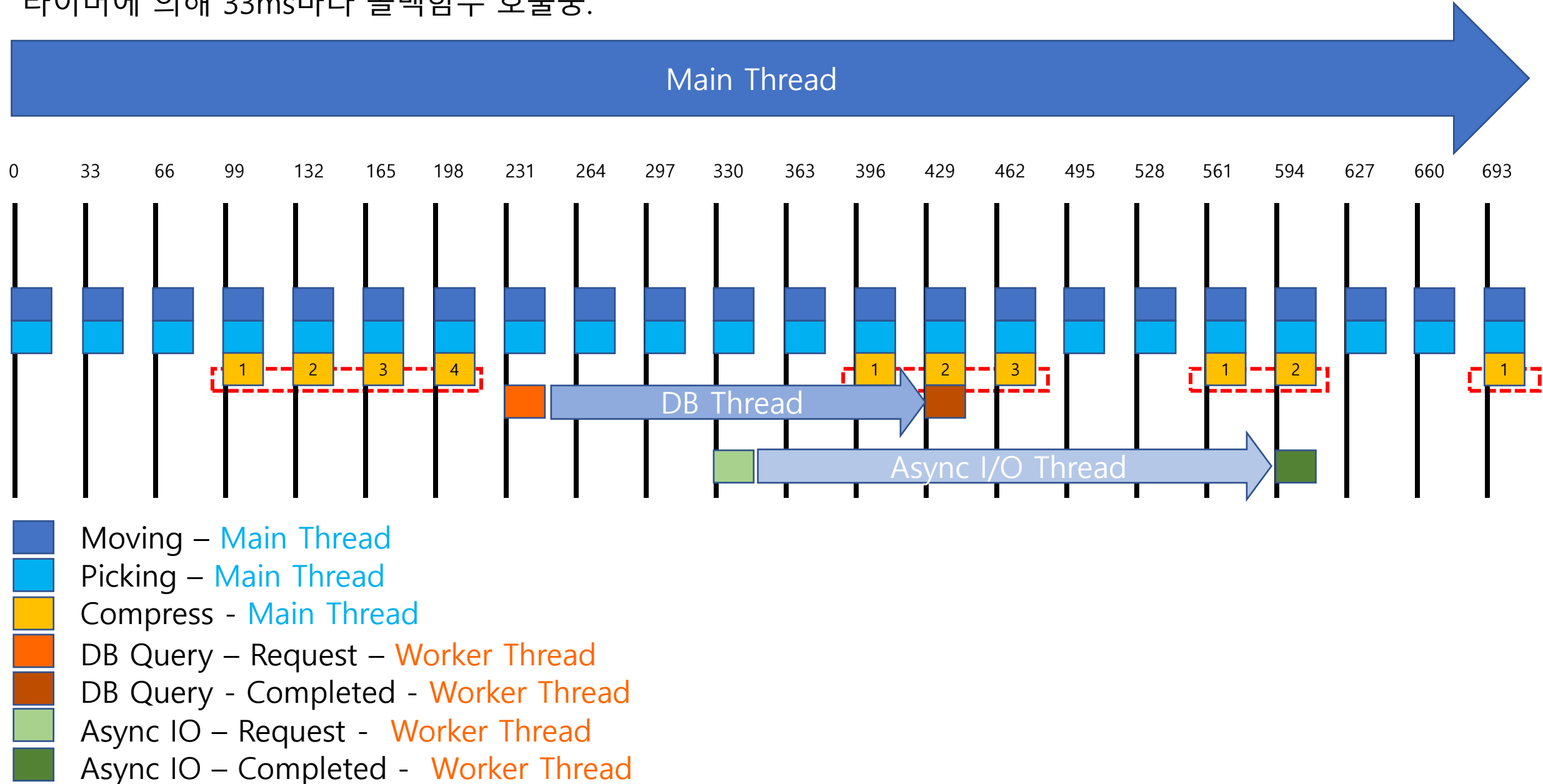




# 서버의 작업 스케줄링 정리

# 서버의 작업 스케줄링 정리

타이머에 의해 33ms마다 콜백함수 호출중.



# 비용절감

메모리 절약 + CPU자원 절약(안해도 되는 작업 안하기)

# 다수의 인스턴스맵 생성

- 보스전을 위한 인스턴스맵.
- PvP를 위한 인스턴스 맵.
- 개인맵을 위한 인스턴스맵.

# 인스턴스맵 - 보스전 맵, PvP맵등...



<https://youtu.be/V1M1edjBULQ>

# 인스턴스맵 - 개인맵



<https://youtu.be/bwEOJTAT3JU>

# 다수의 인스턴스맵 생성

- 맵 생성시에 Tree빌드가 필요하다. 제법 시간이 걸린다.
- 플레이어가 1000명 접속해서 개인맵을 만들면?
- 플레이어가 1000명 접속해서 보스전을 뛰면?
- 메모리 사용량을 줄일 수 있을까?
- 인스턴스맵 생성/로드 시의 CPU비용도 크다.
- CPU가 아무리 빨라도 디스크 I/O비용을 줄이지 못한다.



# 복셀 오브젝트 단위의 인스턴싱(참조)

- 변형 불가 맵이라면 읽기전용이므로 메모리를 완전히 공유할 수 있다.
- 최초로 개인맵을 만들때의 템플릿은 모든 플레이어에 대해 동일하다.
- 복셀 오브젝트 단위로 참조를 구현하자.

# 복셀 오브젝트 단위의 인스턴싱 + Copy on Write

- 복셀 지형은 언제든지 변형될 수 있다. (편집 이벤트, 무기에 의한 파괴 이벤트)
- 따라서 단순한 인스턴싱(참조)만으로는 부족하다.
- 복셀 오브젝트 단위로 참조를 구현하되....
- 추가로 **Copy on Write** 구현.

# 복셀 오브젝트 단위의 인스턴싱

- src가 주어지고 참조로 복제가 일어나는 경우, src의 포인터를 보관. 이때 src오브젝트의 ref\_count를 증가시킨다.
  - `src->ref_count++`
- 플레이어의 편집요청, 무기에 의한 변형(로켓탄의 폭발, 총알에 피탄 등)이 일어날 경우 별도의 메모리를 할당하고 src오브젝트의 내용을 복사. 이때 src오브젝트의 ref\_count 감소.
  - `src->ref_count--;`
- 오브젝트를 새로 만들어서 복사할때 공간자료구조(KD-Tree등)는 새로 생성하지 않고 src로부터 메모리 단위로 카피.

# 참조를 이용한 copy & paste , Copy-on-Write



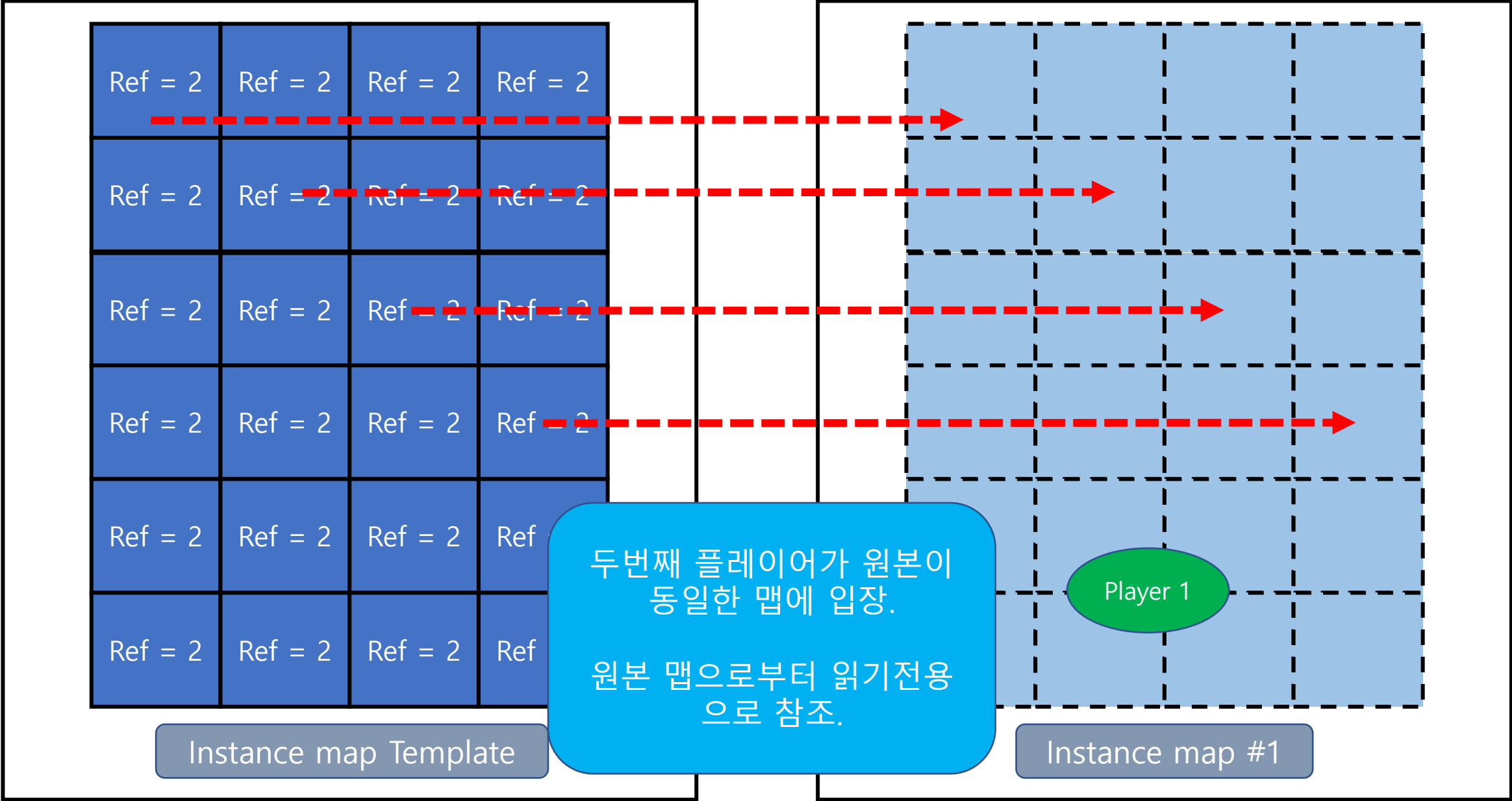
<https://youtu.be/QUOlrX8lfag>

참조를 이용한 인스턴스맵, Copy-on-Write

Ref = 1	Ref = 1	Ref = 1	Ref = 1
Ref = 1	Ref = 1	Ref = 1	Ref = 1
Ref = 1	Ref = 1	Ref = 1	Ref = 1
Ref = 1	Ref = 1	Ref = 1	Ref = 1
Ref = 1	Ref = 1	Ref = 1	Ref = 1
Ref = 1	Ref = 1	Ref = 1	Ref = 1

Instance map Template

참조를 이용한 인스턴스맵, Copy-on-Write



## 참조를 이용한 인스턴스맵, Copy-on-Write

Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2

Instance map Template



무기에 의한 파괴, 혹은 복셀 편집 이벤트 발생

Instance map #1

Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 1	Ref = 2
Ref = 2	Ref = 1	Ref = 1	Ref = 2
Ref = 2	Ref = 2	Ref = 1	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2
Ref = 2	Ref = 2	Ref = 2	Ref = 2

Instance map Template



Instance map #1



# 복셀 오브젝트 단위의 인스턴싱

- 맵 생성시 초기화 시간 대폭 감소(Tree빌드 시간 등)
- 메모리 사용량 대폭 감소

멀티 스레드 적용시 고려할 사항

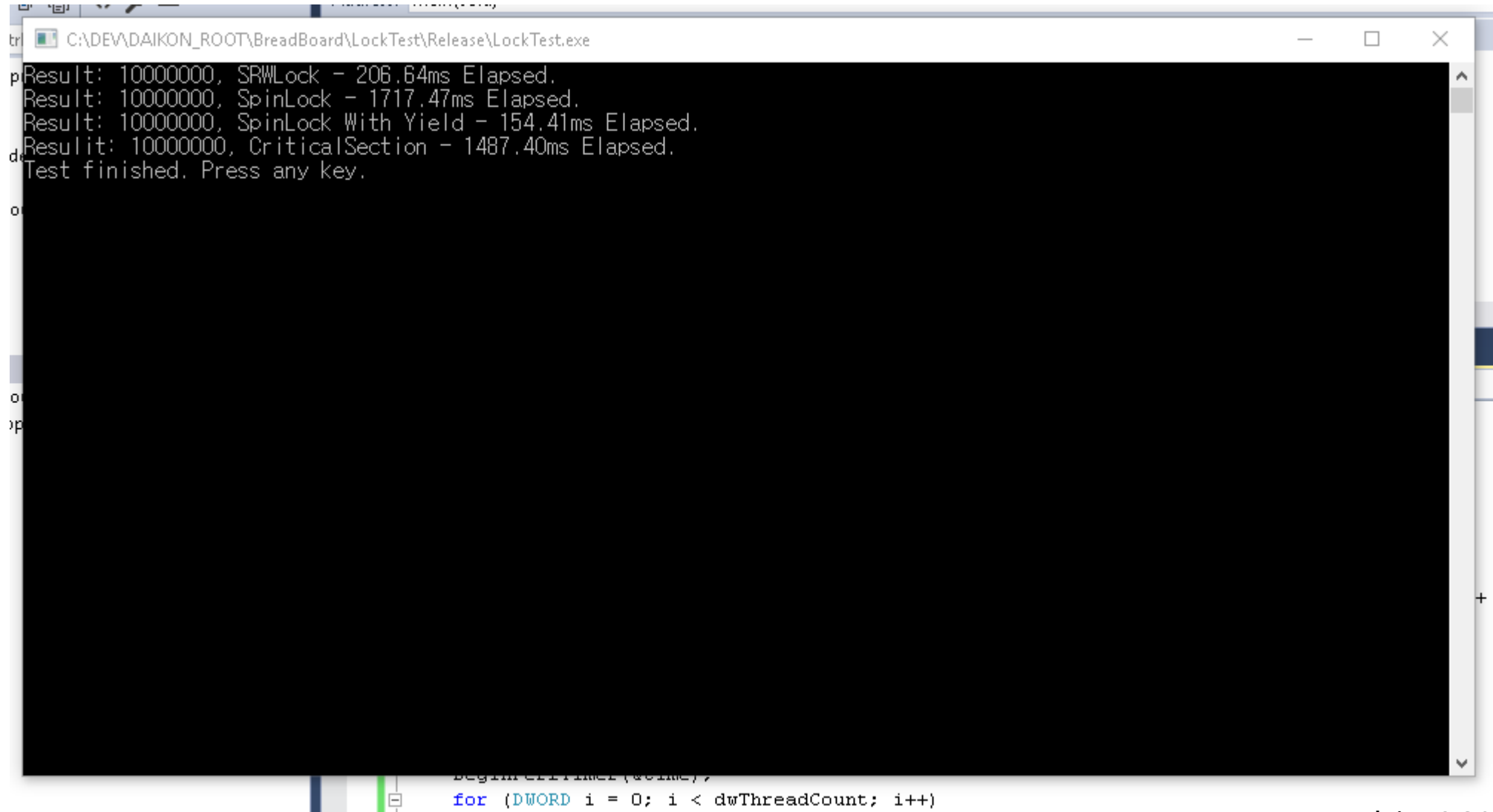
# 멀티 스레드가 능사가 아니다.

- 동기화 비용은 생각보다 크다.
- 초기 코딩의 어려움.
- 디버깅의 어려움.
- 유지보수의 어려움.
- 처리량의 한계로 응답성이 떨어지는 경우만 사용할 것.
- 활성화된 core개수에 따라 Turbo boost가 꺼지므로 예상보다 더 성능이 안나옴.

# 동기화 객체별 응답시간(작을수록 빠름)

Atomic operation < Custom Spin Lock < SRWLock  
< CriticalSection < Mutex, Event

# 10000000까지 카운트 소요시간



```
tr C:\DEV\DAIKON_ROOT\BreadBoard\LockTest\Release\LockTest.exe
p Result: 10000000, SRWLock - 206.64ms Elapsed.
Result: 10000000, SpinLock - 1717.47ms Elapsed.
Result: 10000000, SpinLock With Yield - 154.41ms Elapsed.
d Result: 10000000, CriticalSection - 1487.40ms Elapsed.
Test finished. Press any key.
```

```
BeginCriticalSection(&gLock);
for (DWORD i = 0; i < dwThreadCount; i++)
```

Intel i7 2600k, 4 threads)

# 결론

- 처리량에서의 병목으로 응답성이 떨어지는가?
  - Thread Pool을 이용한 병렬 처리
- 당장 처리할 필요가 있는가?
  - 당장 처리할 필요가 없으면 비동기 처리
  - 메인 로직과 분리할 수 있는가? -> 별도의 스레드를 사용하는 비동기 처리
- 당장 처리해야 하고 병렬화가 어려운 경우.
  - 코드 최적화.
- 읽기전용 메모리는 공유, 쓰기가 발생하는 경우 copy-on-write적용

# Reference

- [SRWLock 빠른 성능의 비결](#)
- [복셀기반 네트워크게임 최적화 기법](#)
- [MMOG Server-Side 충돌 및 이동처리 설계와 구현](#)
- [서버와 클라이언트 같은 엔진 사용하기](#)