1. 가변 길이 패킷, 패킷 구조 재정리

개인 프로젝트를 돌리다 보면 간혹 가다 같은 코드임에도 클라이언트가 이상한 값을 받아오는 등 받은 버퍼의 데이터에서 오류가 날 때가 있다.

패킷 재조립이 구현되지 않은 부분과 쓸데없이 큰 버퍼를 만들어 주고 받는 부분이 가장 의심스러운 원인이라 생각했다.

우선 패킷을 가변 길이로 바꾸는 것을 먼저 해결하려 했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

패킷 종류 정리

패킷 종류를 구분할 때 클라이언트에서 서버로 보내는 패킷은 ‘CS’,

서버에서 클라이언트로 보내는 패킷은 ‘SC’ 으로 이름을 붙여 구분하기 쉽게 만들었다.

참고로 패킷을 보낼 때 ZeroMemory()로 버퍼를 0으로 초기화하기 때문에 받는 쪽에서 패킷 종류를 0으로 판단하면 올바르게 전송되지 않았다는 의미이다. 따라서 이를 확인하기 위해 E\_PACKET\_NONE이라는 이름으로 만들어 두었다.

게임 서버 프로그래밍 수업에서 예제 코드를 보면 패킷을 구조체로 각각 만들어 관리한다.

텍스트이(가) 표시된 사진

자동 생성된 설명 게임 서버 프로그래밍 예제 코드 중 패킷 구조체 부분

예제 구조체 몇 개만 보더라도 개인적으로 꺼림칙한 부분이 몇 가지 생각났다.

첫째로, 코드 모습만 보면 패킷 크기(size)와 패킷 종류(type)의 부분이 모든 패킷 구조체에 필요하다는 것이고 이것이 반복된다는 것이 코드 가독성에 방해가 된다는 점이다.

텍스트이(가) 표시된 사진

자동 생성된 설명 

이 점을 해결하기 위해 우선 구조체가 아닌 클래스로 만들어 패킷 크기(size)와 패킷 종류(type)를 갖는 클래스를 상속받도록 하면 성능은 동일하나 코드가 더 간결해질 수 있다 생각했다.

그리고 클래스를 상속받을 때 부모 클래스의 데이터가 자식 클래스 데이터와 비교해 앞에 있는 지 뒤에 있는 지 몰랐는데 직접 실행해 본 결과 부모 클래스 데이터는 자식 클래스 데이터의 앞에 배치하는 것을 알아냈다.

두번째로, 패킷 종류가 늘 때 마다 클래스를 하나씩 늘려가야 한다는 점이 불편했는데, 패킷마다 가변길이를 요구해야 하기 때문에 이 점은 불가피한 것으로 보인다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 패킷 버퍼의 시작지점만 알면 패킷의 크기와 패킷의 종류, 그 패킷을 처리할 클라이언트 id를 알 수 있다.

따라서 확장 OVERLAPPED 객체도 이에 맞게 패킷의 주소만 받아 버퍼에 복사하도록 바꿔주었다.

패킷의 크기 – packet[0] (패킷의 1번째 바이트)

패킷의 종류 – packet[1] (패킷의 2번째 바이트)

패킷을 처리할 클라이언트 ID – packet[2] (패킷의 3번째 바이트)

(단, 서버에서 클라이언트로 가는 SC 패킷일 때)

클라이언트 수신 과정 문제

패킷을 가변길이로 바꾸면서 클라이언트에서 서버로 패킷을 수신하는 과정에 문제가 생겼다.

텍스트이(가) 표시된 사진

자동 생성된 설명 이전 코드

지금까지의 코드에서 패킷 송신을 하는 방법은 Scene에서 이벤트가 발생했을 때 서버로 보내야 할 패킷을 생성하고 SendQueue에 담아 NetworkMGR에서 한 번에 처리하는 방식이다.

하지만 패킷 길이가 바뀌면서 SendQueue가 담을 패킷들의 class 형태가 달라졌기 때문에 패킷을 담은 컨테이너를 사용하기 어려워진다.

바로 생각나는 해결법으로 다음이 있다.

New/delete를 이용한 동적할당을 이용하고 패킷 큐는 그 포인터를 갖는다.

위 방법을 구현하는 방식은 큐 컨테이너 템플릿을 PACKET\_HEAD\*으로 만들고,

Make\_shared<PACKET\_HEAD>()를 이용하여 큐에 emplace하면 포인터를 통해 각각 다른 크기의 클래스를 관리할 수 있다.

하지만 이는 가장 피하고 싶은 방법으로, 패킷 송신이 프레임 평균적으로 적지 않게 일어날 수 있는데 이 때 마다 힙 메모리 생성/삭제가 일어난다면 프로그램 성능에 악영향을 미치게 된다.

그래서 다음 방법을 생각했다.

패킷 단위로 큐에 넣지 말고 한 프레임에 보내야 할 패킷들을 하나의 버퍼에 저장해서 서버로 한번에 묶어 보낸다.

텍스트이(가) 표시된 사진

자동 생성된 설명

우선 큐 컨테이너를 위와 같이 적절한 크기의 char형 배열로 만들었고, 마지막 버퍼의 빈 공간 시작을 가리키는 인덱스도 만들어 주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

큐의 내용을 뺄 때 사용할 함수도 정의해 두었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

패킷을 큐에 추가시키려면 우선 패킷의 주소를 받아온다. 받아오는 주소의 자료형은 여러 패킷 종류가 들어올 수 있기 때문에 void\*로 정의해 두었다. 이 부분은 PACKET\_HEAD\* 로도 대체 가능한 부분이다.

Void\* 형의 주소가 들어오면 그 주소를 char\* 형으로 바꿔준 다음, SendQueue에 알맞게 추가하면 되는데,

만약 SendQueue의 마지막 버퍼에 추가하려는 패킷을 넣을 수 있다면 넣고, 그렇지 않다면 새로운 버퍼를 만들어 그곳에 추가한다.

이렇게 하면 한 프레임에서 여러 번 AddSendPacket()이 호출 될 때 SendQueue의 버퍼 들에는 여러 패킷이 포함 될 수 있고, 이는 NetworkMGR에서 WSASend()를 덜 호출하게 되는 것으로 이어져 네트워크 성능에 좋은 영향을 미칠 것이라 예상된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

NetworkMGR에서 서버로 패킷을 보낼 때는 SendQueue에서 버퍼를 받은 다음, 패킷의 크기를 이용하여 보내야 할 버퍼 크기를 찾는다. 그리고 나서 Send나 SendOverlapped 등을 이용하여 데이터를 보내고 사용한 버퍼를 비워주면 된다.

클라이언트에서 AddSendPacket() 호출 예시

텍스트이(가) 표시된 사진

자동 생성된 설명

클라이언트가 보내고 싶은 패킷이 생긴다면 보낼 패킷을 정의한 다음,

그 패킷의 주소를 MSGQueue::AddSendPacket()의 인자로 넘겨주면 된다.

1. 패킷 재조립 구현

가변 길이 패킷이 완성됐다. 그렇다면 이제 recv할 때 패킷이 잘려올 경우가 생길 것이다.

잘려온 패킷을 조립하여 제대로 수행하도록 하는 코드를 게임 서버 프로그래밍 수업을 참고하여 만들어 보려 한다.

1. 남은 데이터 크기를 저장할 변수를 하나 만들어 0으로 초기화 한다.



1. 받은 데이터의 크기에 패킷 하나 하나의 크기를 줄이며 데이터를 처리하고 남은 데이터가 있다면 저장한다.
2. 잘려온 데이터 크기가 0이 아니라면 남은 데이터를 앞으로

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 다음 데이터를 받을 때 남은 데이터 크기를 뺀 만큼 데이터를 받아온다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 2, 3, 4를 반복한다.
2. 패킷 처리 함수

패킷 재조립을 하면서 Process\_Packet() 함수에 버퍼의 주소를 인자로 보내 패킷 하나를 해석하고 처리할 함수를 만들었다.

 클라이언트로부터 오는 패킷을 처리하는 함수

첫번째 인자는 어떤 클라이언트의 패킷인가를 의미하고

두번째 인자는 처리할 패킷의 첫 주소를 의미한다.



클라이언트에서 패킷을 처리할 때는 다른 클라이언트 정보(Socket 정보 등)가 없으니 처리할 패킷의 주소만 있으면 된다.

패킷 처리 예시

텍스트이(가) 표시된 사진

자동 생성된 설명

p\_Packet[1]은 패킷의 unsigned char type이 저장된 공간이다. 따라서 이 값으로 패킷 종류를 구분한다.

패킷의 타입을 알아냈다면 char\*형인 p\_Packet을 패킷 타입에 맞는 class로 캐스팅하여 class의 멤버 변수로 값을 간단하게 추출하여 원하는 처리를 수행한다.

서버의 경우 받아온 데이터를 처리하고 다른 클라이언트들에게 처리 결과나 이벤트 발생을 알리기 위해 ‘SC’클래스를 만들어 정의한 뒤, 클라이언트들에게 다시 Send한다.

1. 클라이언트 여러 씬 관리

현재 클라이언트는 static GameScene\* MainScene을 이용하여 하나의 메인 Scene을 FrameAdvance()로 갱신하고 있다.

지금은 테스트를 위해 하나의 Scene만 생성해서 사용하고 있는데, 여러 Scene을 만들어 주소를 저장하고 그 중 하나를 MainScene으로 고를 수 있게 만들면 간단하게 Scene 전환이 가능할 것이라 생각했다.

여러가지 Scene을 구분하고 생성할 때 헤더 파일이나 cpp 파일로 분할할 수 있으면 좋겠다는 ‘이서연’ 학우의 의견이 있었다. 그래서 Scene을 cpp 파일로 분할하고 메인 코드에선 간단하게 추가할 수 있도록 구현했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

GameScene을 상속받는 새로운 종류의 Scene을 준비한다. Scene들마다 다른 내용을 가져야 할 부분이 가상함수로 정의돼 있어야 한다.



로그인 화면, 로비 화면, 스테이지 등 여러 Scene의 주소를 저장하는 컨테이너가 필요하다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Scene을 생성하고 그 주소를 m\_GameScenes 넣어서 여러 Scene을 컨테이너에 준비하는 모습이다.

Login 과 Lobby의 BuildObjects()는 아직 구현하지 않았기 때문에 확인을 위해 모두 Stage와 같은 Build를 수행한다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

임시로 Scene 전환을 확인하기 위해 위의 세 가지 키를 누르면 0번, 1번, 2번 Scene으로 전환되게 구현해 두었다.

이렇게 하고 프로그램을 실행하면 3가지 Scene을 전환할 수 있는 모습이 보여야 할 것이다.

하지만 프로그램은 실행되지 않는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

CreateShaderResourceViews()에서 예외가 발생해서 그 원인을 찾으려 했다.



여러 부분을 찾아보고 디버그해본 결과 Scene의 Player에 직접 동적 할당하여 생성하는 것이 지금 코드에선 문제가 되는 것으로 추측됐다. 위 코드를 추가하지 않으면 오류가 나지 않게 된다.

해당 오류는 다음 주 차에 같이 해결하겠다..