|  |  |  |  |
| --- | --- | --- | --- |
| [Cranding(크랜딩)] | | | |
| **6주차** | **2024.01.28 ~ 2024.02.03** | **작성자** | **정경서** |
| **주간 회의** | | | |
| **회의 날짜** | |  | |
| 해당 주차에는 회의 내용이 없다고 판단하여 회의를 진행하지 않았습니다. | | | |
| **이번주 한 일** | | | |
| 1. 신동엽(서버)   서버 프레임워크 재설계  지난주 서버에서 클라의 연결을 제대로 받지 못하는 오류가 발생했었다. 결론부터 말하면 이유는 IOCP의 포인터 주소를 넘기고 있었기 때문에 제대로 된 연결이 이루어 지지 않았었다.  기존에 오류를 수정하기 전엔, iocp에 소켓을 등록하기 위해 시도를 할 떄 네트워크 클래스와 IOCP클래스를 따로 나누어 설계를 하였고, 따라서 iocp register, 즉 listen 상태인 서버 소켓을 iocp 에 등록하기 위해 원본을 넘겨야 한다는 생각에 함수의 인자로 포인터를 받아 주소를 직접 넘겨주는 형태로 설계를 해두었다. 하지만 이 부분에서 치명적인 문제가 발생한다는 것을 늦게 깨닫게 되었다. 소켓 자체의 반환값은 INT형 PTR이였고, 내가 만든 문제는 포인터의 포인터 주소를 넘겨주었기 때문에 제대로 연결이 될 수 없었던 것이다. 따라서 포인터를 사용하지 않고, 레퍼런스를 이용하여 넘겨주니 제대로 클라이언트가 서버에 접속에 성공 할 수 있었다.  서버 프레임워크를 하나하나 고치고 수정하던 도중에, 이 코드는 내가 직접 다 파악하고 만드는 코드가 아니라는 생각이 들었다. 수업을 들으며 참고한 코드이지만 완전 100퍼센트 이해하지 못하는 코드라고 생각하여 순수하게 내가 이해한 코드들로만 프레임워크를 재작성 하였다.  서버 클래스를 싱글턴 패턴으로 이후에 있을 DB연결 등에 여러 차례 호출되는 것을 방지하고자 객체의 인스턴스가 오직 1개만 생성되게 만들었다.      또한 Overlapped과 Session부분을 클래스로 나누어 관리를 해주고, 서버를 연결하고 send하고 recv받고 처리하는 부분을 서버에 몰아 두었다. 클라이언트들을 관리하는 clients 컨테이너는 extern을 사용하여 우선 전역으로 처리하였으나, 추후에 필요에 따라 수정할 것 같다.  Send, Recv    우선 세션클래스 안에 do\_send와 do\_recv 함수를 만들어 send recv를 호출하고, 이를 처리해주는 worker Thread함수 내부는 작업중에 있다.  이렇게 수정한 이후 정상적으로 서버에 여러 클라이언트들이 차례로 접속에 성공하였다.  클라이언트 네트워크 설계  서버와 연결하는 쓰레드와 렌더링을 하는 쓰레드를 분리하여 네트워크를 설계하였다.  이 두 쓰레드는 공유 자원으로 게임에 접속한 다른 클라이언트들의 정보를 담고있는 clients 컨테이너를 공유하며 네트워크를 담당하는 쓰레드에서 서버 쪽에서 recv를 받으면 컨테이너에 정보를 저장하고 동시에 렌더링하는 쓰레드에선 매 프레임마다 이 컨테이너의 정보에 따라 렌더링을 하게 설계하였고, 현재 쓰레드 생성, 연결, 정보 업데이트까지 완료 하였다.  Echo 서버 X  C++ 가상함수  가상함수란 부모 클래스 에서 상속받을 클래스에서 재정의 할 것으로 “기대”하고 정의해놓은 함수이다. Virtual이라는 예약을 통해 파생 클래스에서 재정의 하면 이전에 정의 되었던 내용들은 모두 새롭게 정의된 내용들로 교체된다.  그렇다면 왜 가상함수를 사용해야 하는가?  컴파일러가 함수를 호출 할 때에는 정적 바인딩을 통해 함수를 호출하는 코드를 고정된 메모리 주소로 변환시킨다. 여기서 함수를 오버로딩하게 되면 정적바인딩으로 인한 문제가 발생한다.  가상함수가 아닌 보통의 경우 부모 클래스형 포인터로 멤버함수를 호출 할 때 컴파일러는 정적타입을 보고 이 타입에 맞는 멤버함수를 호출하기 때문에 문제가 발생한다.  부모에서 virtual 함수를 사용하여 동적바인딩을 하여 문제를 해결한다.  가상 함수 테이블 – 함수 포인터 배열   1. 정경서(클라이언트)   1. 3D게임프로그래밍 개념 공부  Direct3D 파이프라인  Drawprimitive 라는 함수로 나타난다    모델의 정점들의 정보(입력 리소스)가 입력 조립기에 들어가서 출력 병합기에서 출력이 되도록 하는 단계  오른쪽은 각 단계에서 사용하는 그래픽 메모리에 있는 데이터들  총 10개의 단계(쉐이더 5개 + 고정파이프라인 5개)들이 순차적으로 실행 된다  Direct3D 12 렌더링 파이프라인  리소스: 그래픽 메모리에 있는 데이터(정점/인덱스, 텍스쳐)  GPU를 사용하여 리소스를 2D 이미지로 렌더링 하는 과정  파이프라인은 프로그램 가능 단계와 고정 프로그램 단계로 구분  고정 프로그램 단계( 응용 프로그램에서 변경 불가 )  입력 조립 단계( IA : Input Assembler Stage )  테셀레이터 단계( TS : Tessellator Stage )  스트림 출력 단계( SO : Stream Output Stage )  래스터라이저 단계( RS : Rasterizer Stage )  출력 병합 단계( OM : Output Merger Stage )  위 단계는 하나의 함수처럼 보면 좋다. 입력 파라메터나 글로벌 변수는 변경 가능  프로그램 가능 단계( GPU에서 실행되는 프로그램을 쉐이더 프로그램을 통해 제공해야하는 단계 )  정점 쉐이더 단계( VS : Vertex Shader Stage)  헐 쉐이더 단계( HS : HullShader Stage)  도메인 쉐이더 단계( DS : Domain Shader Stage)  기하 쉐이더 단계( GS : Geometry Shader Stage)  픽셀 쉐이더 단계( PS : Pixel Shader Stage)  헐 쉐이더와 도메인 쉐이더는 테셀레이터 단계가 활성화 될 때만 실행된다.  기하 쉐이더 단계, 헐 쉐이더 단계, 도메인 쉐이더 단계는 모든 파이프라인에서 실행 되는것은 아니다  고정 프로그램 단계는 CPU에서 실행이 되고 프로그램 가능 단계는 GPU에서 실행이 된다  파이프라인을 하나의 함수로 이해하는것이 편하다  렌더 타겟 : 렌더링 파이프라인을 거친 결과물이 기록되는 버퍼 또는 화면을 말한다  Direct3D 렌더링 파이프라인의 단계들은 연결되어 연동되도록 구성되어있다  Direct3D 디바이스( = GPU )    하드웨어를 소프트웨어 적으로 표현한것  Direct3D 디바이스는 상태 기계(State Machine)이다. 상태를 바꿔주지 않으면 이전 상태를 유지한다  compute pipeline → 계산을 하는 파이프라인  리소스를 실제로 디바이스에 연결하기 위해서는 뷰의 상태로 연결해야한다  출력은 렌더 타겟에 실제로 그리거나 Depth/Stencil Buffer에 이후에 사용할 정보를 출력  2. 코드 수정  프로젝트에서 3D 객체를 불러오기 위해 LoadGeometryFromFile 함수를 기존 프로젝트에 이식하였다.  이때 기존 코드에서 사용하는 LoadFrameHierarchyFromFile함수와 이식하려는 코드에서 LoadFrameHierarchyFromFile의 인자가 서로 달라서 이 부분을 아래와 같이 해결해 주었다.  코드를 살펴보니 pShader와 pnSkinnedMeshes를 NULL로 설정해준다면 기존 코드와 이식하려는 코드 사이에 차이가 없어서 아래와 같이 해결이 가능하였다.  static CGameObject \*LoadFrameHierarchyFromFile(ID3D12Device \*pd3dDevice, ID3D12GraphicsCommandList \*pd3dCommandList, ID3D12RootSignature \*pd3dGraphicsRootSignature, CGameObject \*pParent, FILE \*pInFile, CShader \*pShader = NULL, int \*pnSkinnedMeshes = NULL);  또한 코드에서 RootSignature, Viewport, shaderVariables 등 다양한 설정을 Render 함수에서 진행하고 있는데, 이는 Render를 준비하는 OnPrepareRender 함수에서 진행하도록 코드를 수정하였다.  3. 디퍼드렌더링 공부  Deferred Rendering이란 여러 빛의 연산을 지연 계산하는 것으로, 여러 라이팅 효과를 더욱 효율적으로 계산하는것 입니다.  필요한 정보들을 GBuffer에 저장을 하고, 이를 합성하여 화면에 렌더링 하는 것 입니다.  실제로 화면에 렌더링 되는 픽셀만 계산을 수행하기에 부담이 덜합니다  또한 GBuffer를 다른 기법들에 다시 사용할 수 있습니다.  GBuffer는 Geometry Buffer로 렌더 타겟들을 모아둔 버퍼입니다.  색상값을 위한 Albedo, 조명 연산을 위한 Normal, 위치 값을 얻기 위한 Depth, 재질을 위한 Sepcular 값 등을 기록합니다.  하지만 반투명한 객체의 렌더링은 Normal과 Depth에서 문제가 생기기 때문에 포워드 렌더링으로 처리해주어야 한다.   1. 홍예나(클라이언트)   원래 하려던 방식으로 구현하면 추가해야하는 예외 상황이 많아져서 복잡해지고  애니메이션 블렌딩을 하기 위해서는 애니메이션1의 키가 눌린 지점의 포지션과, 애니메이션2의 첫 번째 포지션의 변환 정보만 계속해서 사용해야하는데 자꾸 포지션이 갱신되는 문제가 있다는 것을 알게 되어서 다른 방식으로 구현하였다    animationController 클래스 내부에 m\_bIsBlending 변수를 만들어주어서 애니메이션 블렌딩을 해야하는 상황과 아닌 상황을 for문이 돌기 전에 구분되도록 하였다  따라서 애니메이션 블렌딩이 필요한 상황에서는 position이 갱신되지 않을 수 있다  또한 m\_fBlendingTime(0.0f)라는 가중치를 대신할 변수를 생성하였고, fTimeElapsed에 따라 계속 증가하도록 구현하였다  만약 m\_fBlendingTime이 1.0f보다 커지게 된다면 블렌딩을 종료하고 m\_fBlendingTime을 다시 0.0f로, animation의 position또한 0.0f로 다시 되돌아가게 설정하였다.  마지막으로 m\_nAnimationBefore, m\_nAnimationAfter 변수를 생성하였다. 실행되던 애니메이션의 track 번호를 전자에, 이제 실행될 애니메이션 track의 번호를 후자에 저장하고 어떤 애니메이션 블렌딩이 되어야 하는 경우를 일일이 설정하지 않아도 알아서 애니메이션 블렌딩이 진행될 수 있도록 구현하였다        이때 문제가 발생하는데 키를 누르게 되면 애니메이션이 블렌딩되고 다음 애니메이션으로 넘어가야 하는데 어떤 애니메이션의 1번째 포지션(기본 자세)로 렌더링된다  키를 눌렀다가 떼었을 때 점프 동작으로 이루어지는 것을 보아 애니메이션 블렌딩 코드 자체에 문제가 있지는 않고 아마도 m\_fBlendingTime이 계속 1.0f보다 크게 설정되고 있는 것 같다 | | | |
| **다음주 할 일** | | | |
| 1. 신동엽(서버)   - 여러 클라이언트 동시접속 이후 한 맵 안에서 띄우기  - 패킷 전송, 수신 처리  - 프레임워크 수정   1. 정경서(클라이언트)  * 애니메이션 관련 오류 끝내기 * 프레임워크 수정 * DirectX 개념 공부  1. 홍예나(클라이언트)  * 가우시안 블러 공부 시작 * 애니메이션 블렌딩 구현 끝 | | | |
| **비고** | | | |
|  | | | |