네트워크 게임 프로그래밍

Term Project 기획서

게임공학부 게임공학과

2011182004 김기남

2011180047 정택수

**목 차**

**1. 게임기획**

**1-1. 게임소개**

**1-2. 게임목표**

**1-3. 개발환경**

**1-4. 상세기획**

**2. 하이레벨(HIGH\_LEVEL) 디자인**

**2-1. 클라이언트 플로우 차트**

**2-2. 클라이언트 플로우 차트 설명**

**2-3. 클라이언트 클래스 다이어그램**

**2-4. 서버 플로우 차트**

**2-5. 서버 플로우 차트 설명**

**2-6. 서버 클래스 다이어그램**

**3. 로우레벨(LOW\_LEVEL) 디자인**

**3-1. 프로토콜**

**3-2. 클라이언트**

**3-3. 서버**

**4. 개발계획**

**4-1. 개발일정**

**4-2. 역할분담**

**1. 게임기획**

**1-1. 게임소개**

FPS게임의 형식의 게임이므로 화면의 중앙에 표시되는 표적점을 기준으로 총알을 발사하여 마인크래프트 컨셉으로 만든 몬스터들을 사냥한다. 싱글 플레이가 아닌 2인 플레이이므로 다른 플레이어와 함께 즐길 수 있다.



전체적인 배경과 몬스터 디자인

**1-2. 게임목표**

기본적으로 맵에 랜덤하게 배치되어 있는 몬스터들을 사냥하는 방식이다. 몬스터 들을 좌우로 움직이며 플레이어의 조준을 어렵게 할 뿐만 아니라 날씨가 맑음->흐림, 흐림->맑음으로 계속 변화하기 때문에 정확한 조준이 요구된다. 플레이어가 잡은 몬스터의 수를 비교하여 게임 종료시 승패를 좌우 할 수 있게 된다.

**1-3. 개발환경**

1. OS는 Window 환경, 개발언어는 Visual C++을 기반으로 제작한다.
2. 라이브러리는 OpenGL을 이용하여 3D게임으로 제작한다.
3. 네트워크는 TCP 서버를 기반으로 제작한다.
4. GitHub를 활용한 팀원간의 작업 공유.

**1-4. 상세기획**

1. 게임의 클라이언트의 크기는 800x800이다
2. 한 캐릭터, 몬스터는 6개의 큐브로 제작되며 36개의 텍스처로 이루어진다.
3. 3종의 효과음(총알 발사, 몬스터 충돌, 몬스터 사망)이 있다.
4. 조명의 세기를 조정하여 맑음, 흐림효과를 사용한다.



<맑을때의 게임화면> <흐릴때의 게임화면>

1. 게임이 시작되면 다음과 같은 객체들이 존재한다.

* 게임 플레이의 주체가 될 플레이어 객체
* 랜덤한 위치에 생성되는 몬스터 객체
* 배경을 담당하는 스카이박스와 학교건물 객체
* 플레이어의 이동에 방해를 주는 컨테이너 객체

|  |  |  |
| --- | --- | --- |
|  | **조작키** | **설명** |
| **앞,뒤,좌,우 이동** | W,S,A,D키 | 해당 방향으로 이동한다. |
| **총알 발사** | 마우스 왼쪽 버튼 | 바라보는 방향으로  총알을 발사 |
| **카메라회전** | 마우스 이동 | 마우스 움직임에 따라 바라보는 위치가 바뀐다. |

1. 모든 몬스터를 사냥한다면 잡은 몬스터수를 비교하여 더 많은 몬스터를 잡은 플레이어가 승리한다.(사냥 성공시마다 KillCount를 계산한다.)

**2. HIGH\_LEVEL 디자인**

**2-1. 클라이언트 플로우 차트(Flow-Chart)**



**2-2. 클라이언트 플로우 차트 소개**

1. 클라이언트는 타이틀 화면에서 스페이스 바를 누르면 서버와 connect를 시도한다.
2. connect가 완료되면 클라이언트에게는 상대방을 찾는 중이라는 표시를 보여주고, 게임 시작을 위해 recvn()에서 대기한다.
3. 서버에서 시작에 대한 프로토콜을 송신한 것을 수신하면, 기본 게임 세팅을 시작하며, 클라이언트는 반복하여 매 간격(Term)마다 프로토콜의 수신을 대기를 한다.

(위의 프로토콜은 게임이 진행중이라면 클라이언트가 서버로부터 반복해서 수신한다.)

1. 서버에서 게임 프로토콜을 받으면 이어서 플레이어의 위치와 몬스터의 위치를 수신 후, 해당 프로토콜의 정보에서 라운드가 종료되었는가를 판별한다.

(해당 프로토콜의 정보에서 라운드가 종료되었는가를 판별할 수 있다.)

1. 서버에서 게임 라운드가 종료되지 않았음을 체크해서 보냈었다면, 서버에게서 받은 상태들을 모두 적용한다.

(만약 루프를 한번 이상 반복한 상태라고 한다면, 서버에게 얻은 계산 결과를 적용하는 단계이다.)

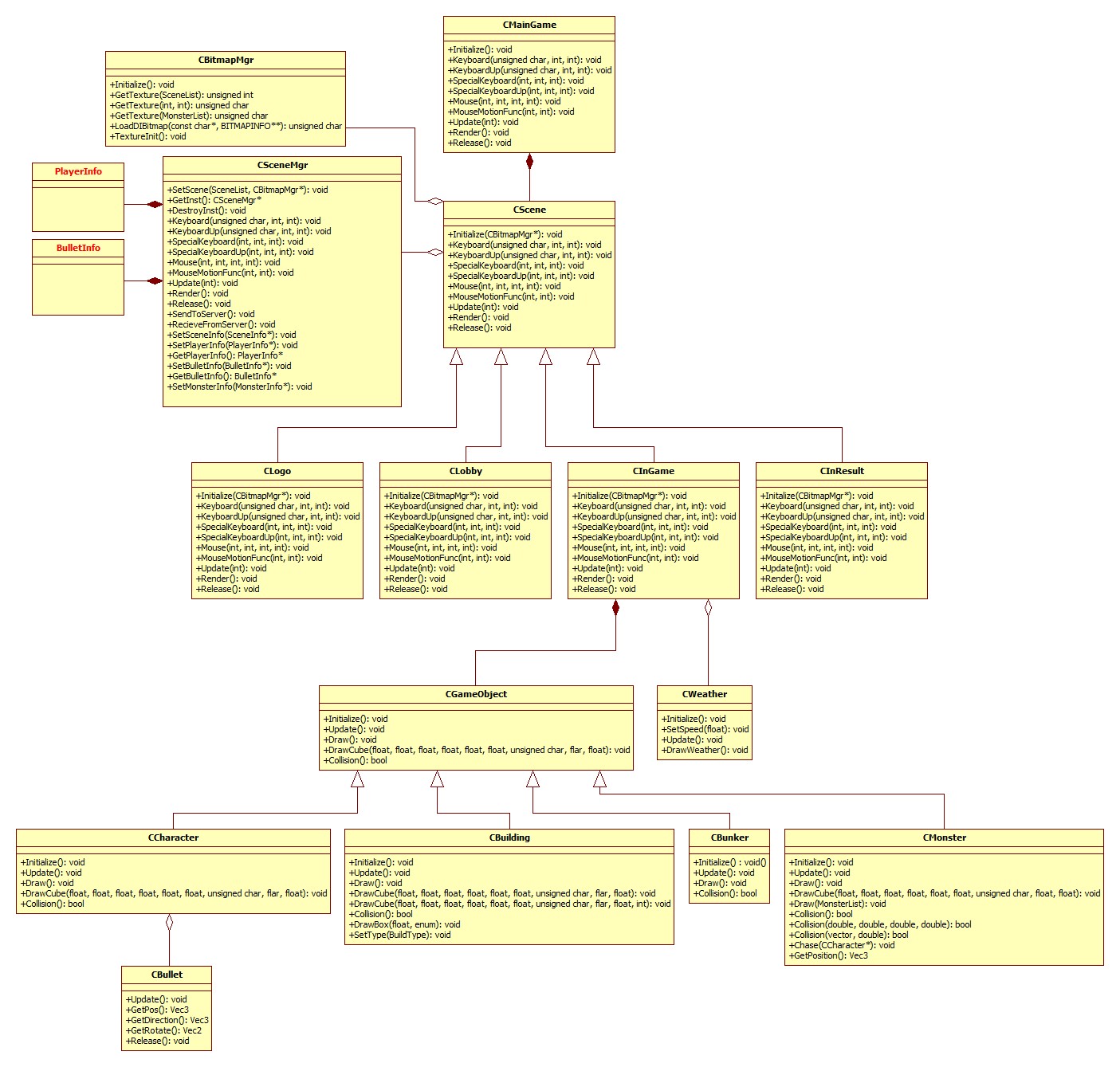
1. 플레이어가 게임 플레이한 정보를 적용한다. (키 입력, 마우스 입력)
2. 플레이어가 입력한 정보에 대한 클라이언트 자체 연산을 한다. 지형, 장애물 등 고정으로 존재하는 오브젝트들에 대한 충돌체크이다.

(캐릭터의 카메라 회전 정보, 이동 정보를 충돌결과와 합산하여 결과를 낸다.)

(연산 결과를 프로토콜에 적용하고 다음 단계로 진행한다.)

1. 서버에게 필요한 검사를 위해 해당 정보들을 send()로 송신한다.
2. ‘3)’과 동일하게 프로토콜에 대해 recvn을 호출한다.
3. 게임이 종료되었음을 서버로부터 recvn을 통해 판단하였으면, 결과를 서버로부터 수신받고 다시 시작을 한다면, 클라이언트에서의 connect를 다시 시도하기 위해 서버와 접속을 종료하고 대기상태로 돌아간다.

**2-3. 클라이언트 클래스 다이어그램**



클라이언트의 구조를 나타내고 있는 클래스 다이어그램이다. 크게 클래스 내부의 오브젝트들을 생성/해제하는 Initialize/Release와 렌더링해주는 Render, 매번 갱신을 담당하는 Update()를 갖고 있다.

* CScene : 게임의 모든 씬이 상속하는 부모 클래스

1. Keyboard, SpecialKeyboard는 키 입력을 관리한다.
2. Mouse, MouseMotionFunc은 마우스 입력과 움직임을 관리한다.

* CGameObject : 게임의 오브젝트들이 상속하는 부모 클래스

1. Draw는 해당 오브젝트를 그린다.

**2-4. 서버 플로우 차트**



**2-5. 서버 플로우 차트 소개**

1. 서버는 처음 실행 후 Socket 생성 이후 Bind 과정을 거치고 Listen 상태로 넘어가게 된다.
2. Listen 상태에서 클라이언트가 접속하면 Accept해주고 2개의 클라이언트가 모두 접속할 때까지 대기한다.
3. 2개의 클라이언트가 모두 접속하면 게임 내의 초기값 들을 양 클라이언트에 보내준다.

* 게임 캐릭터의 초기 위치 벡터 값(x,y,z), 및 게임 내 몬스터 들의 위치 벡터 값(x,y,z)를 전송한다.
* 위에 두 정보를 받고 나면 게임 클라이언트 들은 메인 게임 상태로 넘어가게 된다.

1. 클라이언트로부터 받은 데이터를 바탕으로 연산을 하고 결과를 양쪽 클라이언트에 뿌려준다.

* 클라이언트로부터 캐릭터(NPC 포함)들의 위치 정보, 방향, 애니메이션, 총알 생성 여부를 받는다.
* 클라이언트로부터 받은 데이터를 바탕으로 연산(충돌처리& 동기화 )을 하고 다시 양쪽 클라이언트에 데이터를 전송해준다.

1. 게임 끝 조건을 만족하면 양쪽 클라이언트에 게임이 끝났다는 메시지를 보낸다.
2. 양쪽 클라이언트가 모두 종료를 하면 프로그램을 종료하고, 한 플레이어라도 다시 한다는 메시지를 보내면, 위 플로우 차트에서 2명 모두 접속하였는가? 문구로 이동하게 된다.

**2-6. 서버 클래스 다이어그램**



1. **Class CGameServer**

* 전체 쓰레드를 관리해주는 클래스, 클라이언트간의 연결 및 해제를 관리한다.
* 클래스를 생성하면서 기본 네트워크 작업(윈속 초기화, Bind, Listen소켓 생성 작업을 하게된다.)
* 클라이언트 하나가 접속하면 Accept 함수를 통해 클라이언트 소켓을 생성하고 CThreadManager의 생성자값으로 넘겨준다. 클라이언트가 2개 모두 접속을 하게 되면, CThreadManager를 생성해주고 본격적인 송수신 작업을 하게된다.

1. **Class CThreadManager**

* 각각의 스레드 들을 관리하는 클래스.
* 게임의 실질적인 데이터 송수신을 담당한다.
* 공유자원들을 갖고 있다.

1. **Class COperater**

* 클라이언트로부터 받은 데이터들을 연산해주는 클래스.
* 총알과 NPC 간의 충돌체크.
* 캐릭터와 몬스터, 총알의 위치 동기화를 처리하게 해준다.
* 각각의 프로토콜에 맞게 데이터를 연산한다(Switch문을 통해서).
* 동기화작업은 임계영역을 사용해서 처리해준다.

공유자원(전역변수)

1. PlayerVector – 양쪽 클라이언트들의 캐릭터 정보 데이터를 갖고 있다.

* 위치벡터, 방향, 애니메이션

1. Player\_Socket – 두 클라이언트의 소켓 관리를 위한 배열
2. MonsterInfo – NPC들을 관리하기 위한 배열
3. BulletVector – 총알들의 관리를 위한 배열

* 자세한 설명은 밑에 Low-Level Design에서

**3. LOW\_LEVEL 디자인**

**3-1. 프로토콜(Protocol)**

* 타입 구분없이 공통으로 전송되는 정보

**struct InitInfo //게임의 초기값을 담당하는 구조체**

//Vec3는 float값 3개를 묶은 구조체를 의미한다.

Vec3 Player1Pos; //첫번째 플레이어의 위치를 담당

Vec3 Player2Pos; //두번째 플레이어의 위치를 담당

Vec3 MonsterPos[10]; //몬스터들의 위치정보를 담당

**enum DataType //데이터 타입의 구분 변수를 가진 열거형**

PLAYER, //구분타입 -> 플레이어정보

BULLET, //구분타입 -> 총알정보

MONSTER, //구분타입 -> 몬스터정보

SCENE //구분타입 -> 씬정보

**enum SceneList //씬 구분 변수를 가진 열거험**

LOBBY, //구분타입 -> 로비 씬

INGAME, //구분타입 -> 게임진행 씬

RESULT //구분타입 -> 게임결과 씬

* 타입 구분이 필요한 정보

**struct SceneInfo //씬의 구분을 담당하는 구조체**

int type; //데이터 타입의 구분을 담당

int SceneState; //현재 씬의 상태를 구분

**struct PlayerInfo //플레이어의 정보를 담당하는 구조체**

int type; //데이터 타입의 구분을 담당

Vec3 PlayerPos; //플레이어의 위치를 담당

Vec3 CameraDir; //플레이어의 방향을 담당

bool AniandKeyState; //키의 눌림 여부 담당

**struct BulletInfo //총알의 정보를 담당하는 구조체**

int type; //데이터 타입의 구분을 담당

Vec3 BulletPos; //총알의 위치를 담당

BOOL newBullet //총알이 새로 만들어진 총알인지 아닌지

//True면 서버내에 총알관리 벡터에 추가.

int bulletOwner; //누가 발사한 총알인지 구분

int bulletIndex; //총알들을 구분하기 위해서

**struct MonsterInfo //몬스터의 정보를 담당하는 구조체**

int type; //데이터 타입의 구분을 담당

Vec3 MonsterPos; //몬스터의 위치를 담당

int hp=3; //몬스터에 체력을 담당,

//기본적으로 3에 체력을 가지며 총알에 한발 맞으면, 체력이 1이 깎인다.

**Struct ResultInfo // 결과 조건을 보내기 위한 구조체**

int killCount // 플레이어가 몇 마리의 몬스터를 잡았는지

int SceneState // 양쪽 클라이언트에 결과화면을 띄우기 위해 보내는 변수

**3-2. 클라이언트**

**class CSceneMgr**

* 클라이언트와 서버의 송수신을 담당하는 클래스이다.
* CSceneMgr 클래스안의 함수들을 이용하여 송신데이터를 보내거나 수신데이터를 적용한다.
* 다음은 씬매니저의 서버와 통신을 위한 멤버 변수들이다.

**SOCKET m\_sock; // 나(클라이언트)의 소켓**

**InitInfo** CInitInfo**;** //게임의 초기화정보 수신 객체

**SceneInfo** CSceneInfo; //씬 구분정보 수신 객체

**PlayerInfo** CPlayerInfo[2]; //플레이어정보 송수신 객체

**BulletInfo** CBulletInfo[2]; //총알정보 송수신 객체

**MonsterInfo** CMonsterInfo[10]; //몬스터정보 P수신 객체

* 다음은 씬매니저의 서버와 통신을 위한 멤버 함수들이다.

void SendToServer(); //서버로 클라이언트 정보를 송신

void RecieveFromServer(); //서버로부터 필요한 정보를 수신

void SetSceneInfo(**SceneInfo**\* pSceneInfo); //클라이언트에 씬 구분을 수신

void SetPlayerInfo(**PlayerInfo**\* pPlayerInfo); //클라이언트에 플레이어 정보를 담는다.

**PlayerInfo**\* GetPlayerInfo(); //클라이언트의 플레이어 정보를 리턴

void SetBulletInfo(**BulletInfo\*** pBulletInfo); //클라이언트에 총알 정보를 담는다.

**BulletInfo**\* GetBulletInfo(); //클라이언트의 총알 정보를 리턴

void SetMonsterInfo(**MonsterInfo**\* pMonsterInfo);

//클라이언트에 몬스터 정보를 담는다.

void Update(); //클라이언트의 자체연산, 갱신을 담당

**3-3. 서버**



**Class CServer** – 클라이언트들의 접속 해체 및 기본적인 네트워크 작업을 하기위한 클래스

* 멤버변수
* int m\_ClinetNum; //몇 개의 클라이언트가 접속했는지 구분하는 변수
* SOCKET mListen\_sock //listen socket 변수
* 멤버함수
* CServer() – 생성자를 통해 기본적인 통신을 위한 작업을 해준다.(윈속 초기화, 리슨 소켓 생성, 바인드)
* Update( ) - 업데이트 함수 안에서 Accept함수를 통해 클라이언트의 정보를 얻어와서 listen소켓에 넣어준다음에 client소켓에 넘겨준다
* MakeThreadManager() – CThreadManager 클래스를 생성한다. Accept 한 소켓값들을 인자로 받는다.
* ~CServer – 소켓들을 닫고, WSACleanup()를 호출하면서 종료준비를 한다.

**Class CThreadManager** – 스레드를 관리해주기 위한 클래스. 데이터 송수신 및 공유(전역변수)자원들의 공유와 동기화를 하기 위해서 존재한다.

* 멤버변수
* SOCKET client\_sock[2]– 플레이어 소켓들을 관리하기 위한 배열
* vector<PlayerInfo> playerVector – 플레이어들의 정보를 관리하기 위한 동적배열
* vector<MonsterInfo> MonsterVector – 몬스터(NPC)들의 위치를 관리하기 위한 동적 배열, 충돌체크를 위해서도 사용된다.
* **vector<BulletInfo> BulletVector – 총알들의 위치를 관리하기 위한 동적배열, 충돌체크를 위해서 사용된다.**
* **vector<Obstacle> mapObstcle – 맵 상에 장애물들의 위치를 담고 있는 변수. 플레이어와 총알들의 충돌체크를 위해서 사용된다.**
* **Struct map – 맵 데이터를 담고 있는 구조체 변수, 플레이어와 총알들의 충돌체크를 위해서 사용된다.**
* CRITICAL\_SECTION cs – 임계영역을 사용하기 위해서 사용. Why?) FPS게임에선 빠른 연산이 중요하다고 생각해서 임계영역을 사용하기로 결정.
* 멤버함수
* MakeThread(SOCKET socket) – 하나의 클라이언트가 Accept를 성공하게되면 스레드를 만들어준다. 함수 인자로는, CServer클래스에서의 listensocket을 받아온다.
* **//CThreadMgr클래스가 생성되고, 생성자를 통해서 스레드를 두개 만들어준다. 생성자의 인자로 두개의 소켓정보를 받아오고 이 소켓인자들의 정보를 토대로 스레드 함수에 인자로 넘겨주게 되면서 스레드를 생성한다.**
* DeleteThread() – 스레드를 제거해주는 함수이다. 중간에 클라이언트가 끊기거나 하는 경우에 호출된다.
* AddBullet() – 총알 생성 메시지를 받으면 총알 관리 벡터에 Insert한다.
* Update() – 스레드 함수이며, 이 함수내에서 recvn, send 작업이 이루어진다. Send,Recv 작업은 고정+가변 길이로 이루어진다. 고정길이로 보내고자 하는 패킷의 크기를 먼저 보내게 되고, 가변길이로 보내고자 하는 데이터를 보낸다. 위 프로토콜에서 명시했듯이 Type + Data 형태로 보내게 된다.

**COperator** – 클라이언트로부터 받아온 데이터 들을 토대로 연산을 해준다. 실질적인 동기화 작업은 이 클래스에서 이루어진다. CThreadManager 클래스 안에서 생성되며 CThreadManager에 멤버변수들을 포인터로 받아와서 동기화 작업을 한다.

* 멤버변수
* vector<PlayerInfo> \*pPlayerVector – CThreadManager로부터 플레이어의 정보들을 받아오는 변수
* vector<MonsterInfo> \*pMonsterVector – CthreadManager로부터 몬스터들의 정보를 받아오는 변수.
* Vector<BulletInfo> \*pBulletVector – CThreadManager로부터 총알들의 정보를 받아오는 변수.
* **Struct Map \*pMap – CThreadManager로부터 맵의 데이터들을 받아오는 변수.**
* **Vector<Obstacle> \*pObstacleVector – CThreadManager 로부터 장애물들의 데이터들을 받아오는 변수.**
* 멤버함수
* Update() – if문을 통해서 들어온 패킷에 맞게 연산을 해준다. 함수를 빠져나가기 전에 최종적으로 총알과 몬스터 들의 충돌체크를 해준다.
* CollisionBulletWithMonster() – 총알과 몬스터들의 충돌체크를 해주는 함수이다. Update()안에서 연산하게 되고, 해당 총알객체를 지워준다.
* CollisionObstacleWithPlayer() – 지형 장애물들과 플레이어들의 충돌체크를 해주는 함수이다.
* **CollisionBulletWithMap() 총알과 맵과의 충돌체크를 연산해주는 함수이다.**
* **CollisionObstacleWithBullet() – 총알과 장애물들의 충돌체크를 연산해주는 함수이다.**
* PlayerPosUpdate() – 플레이어 들의 위치 업데이트 및 동기화.
* NPCPosUpdate() - NPC들의 위치 업데이트 및 동기화
* BulletPosUpdate() – 총알들의 위치 업데이트 및 동기화
* DeleteMonster() – Monster 배열에서 체력이 0이 되는 몬스터를 제거 해준다.
* sendResult() – 최종 결과를 양 쪽 클라이언트한테 보내준다.

**4. 개발계획**

**4-1. 개발 일정**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **일 정** | | **내 용** | | **비 고** |
| **김기남** | **정택수** |
| 11월 | 2(수) | GitHub 연동 작업 | GitHub 연동 작업 | 타과목 시험 |
| 3(목) | 통신 프로토콜 제작 | 클라이언트 변수, 함수 수정 | 타과목 시험 |
| 4(금) | 통신 프로토콜 제작 | 클라이언트 변수, 함수 수정 |  |
| 5(토) | CServer 제작 | 클라이언트에 필요한 장애물 추가 |  |
| 6(일) | CServer 제작 | 클라이언트에 필요한 장애물 추기 |  |
| 7(월) | CThreadManager제작 | 클라이언트에서 구현되지않은  자체 충돌체크 구현 |  |
| 8(화) | CThreadManager제작 | 클라이언트에서 구현되지않은  자체 충돌체크 구현 |  |
| 9(수) | COperator 제작 | 통신 프로토콜 제작 |  |
| 10(목) | COperator 제작 | 통신 프로토콜 제작 |  |
| 11(금) | 충돌 함수 구현 | 통신을 위한 SceneMgr 수정 |  |
| 12(토) | 충돌 함수 구현 | 통신을 위한 SceneMgr 수정 |  |
| 13(일) | 더미 클라 제작 | 송신, 수신을 담당할 코드 작성 |  |
| 14(월) | 더미 클라 제작 | 송신, 수신을 담당할 코드 작성 |  |
| 15(화) | 더미 클라와의 통신 | 통신 테스트 |  |
| 16(수) | 미흡한점 수정 | 미흡한점 수정 |  |
| 17(목) | 수정된 클라이언트와의  통신 테스트 및 오류사항 수정 | 수정된 서버와의  통신 테스트 및 오류사항 수정 |  |
| 18(금) | 수정된 클라이언트와의  통신 테스트 및 오류사항 수정 | 수정된 서버와의  통신 테스트 및 오류사항 수정 |  |
| 19(토) | 플레이어 동기화 | 몬스터 좌표 수신 후 그리기 |  |
| 20(일) | NPC 동기화 | 다른 클라이언트와  동기화 비교 및 수정 |  |
| 21(월) | 총알 동기화 | 총알 생성, 좌표 수신 후 그리기 |  |
| 22(화) | 총알 동기화 | 다른 클라이언트와  동기화 비교 및 수정 |  |
| 23(수) | 충돌체크 동기화 | 필요 UI제작 |  |
| 24(목) | 충돌체크 동기화 | 필요 UI제작 |  |
| 25(금) | 최종 동기화 점검 | 최종 동기화 점검 |  |
| 26(토) | 게임 종료조건 구현 | 게임 종료조건 수정사항 수정 |  |
| 27(일) | 가벼운 추가 기능 고려 및 제작 | 가벼운 추가 기능 고려 및 제작 |  |
| 28(월) | 최종 점검 | 최종 점검 |  |
| 29(화) | 최종 점검 | 최종 점검 |  |
| 30(수) | 타인 게임 플레이 및 수정사항 체크 | 타인 게임 플레이 및 수정사항 체크 |  |
| 12월 | 1(목) | 타인 게임 플레이 및 수정사항 체크 | 타인 게임 플레이 및 수정사항 체크 |  |
| 2(금) | 수정사항 재구현 | 수정사항 재구현 |  |
| 3(토) | 수정사항 재구현 | 수정사항 재구현 |  |
| 4(일) | 최종 점검 | 최종 점검 |  |
| 5(월) | 시연 준비 | 시연 준비 |  |

**4-2. 역할 분담**

|  |  |
| --- | --- |
| **김기남** | **정택수** |
| 프로젝트 매니저(PM) | 게임 클라이언트 구조 개선 |
| 서버 클래스 설계 | 클라이언트 클래스 설계 |
| 서버 스레드 설계 | 클라이언트에서의 연산 설계 |
| 서버에서의 연산 설계 | 추가기능 및 UI 제작 |
| 동기화 설계 |  |

**서버 변경 점 및 추가**

1. **프로토콜**

**struct playerSend**

int playerIndex;

Vec2 playerPos;

float playerCam;

int Killcount;

**struct SendPacket**

playerSend player1;

playerSend player2;

Vec3 MonstersPosition[10];

bool gameOver;

**struct RecvPacket**

Vec2 playerPos;

float playerCam;

bool makeBullet;

Vec3 bulletDirection;

Vec3 bulletPosition;

* 서버와 클라이언트는 SendPacket, RecvPacket이라는 두 구조체를 통해서 실시간으로 통신을 하게 됩니다. 서버에서의 SendPacket은 클라이언트에서는 RecvPacket이 되게 됩니다. SendPacket에서 PlayerSend 구조체에서는 3d게임이지만 y축 이동은 사용하지 않고 있고, 캐릭터 회전도 y축 회전만 사용하고 있습니다. 그렇기 때문에 사용하지 않는 부분은 필요없다고 판단하고 패킷의 크기를 줄이기 위해 지웠습니다.

1. **구조 변경**

* 원래 3개의 클래스를 사용하고 있었습니다. 동기화를 구현하는 과정에서 예상한것보다 상당히 많은 시간이 걸렸습니다. 구조에 문제가 있는것인지, 아니면 포인터를 잘못 받아오는건지, 직관적으로 바로 볼수 있게 메인함수 하나로 코드를 뭉치게 되었습니다. 결과는 변한 것은 없었습니다. 하다보니 이 부분이 편해가지고 계속 진행하게 되었습니다.

1. **동기화**

* 동기화 부분은 단순히 임계영역을 지정해서 하려 했습니다. 지금 생각해보면 공부가 많이 부족했던 것 같습니다. 이 부분은 계획 했던것보다 상당히 오랜 시간이 걸렸습니다. 현재 완성된 프로젝트에서의 방법은 이벤트 핸들을 사용하였습니다. Main함수내에 MonsterPosUpdate()->Player1Thread->Player2Thread 이 순으로 1초에 60번을 반복하게 됩니다. 클라이언트도 recv를 위한 스레드가 하나 존재하고 클라이언트도 프레임을 1초에 60으로 고정 하였습니다.

1. **충돌처리**

-bool CollisionBulletWithMonster(Vec3&, Vec3&); // 총알하고 몬스터 충돌함수

bool CollisionBulletWithMap(Vec3&); // 맵전체하고 총알

bool CollisionBulletWithObstacle(Vec3&bullet, Vec3& container); // 총알하고 장애물 충돌함수

bool CollisionMapWithPlayer(Vec2 &);

//맵과 플레이어의 충돌함수

bool CollisionContainerWithPlayer(Vec2 &);

//맵의 장애물과 플레이어의 충돌함수

충돌은 5개의 함수를 사용하며, 용도는 주석에 써있습니다.

총알의 업데이트 부분은 클라이언트로부터 총알의 데이터를 받아옵니다. 초기위치, 방향, 플레이어 인덱스. 받아온 데이터를 서버의 총알관리 배열에 추가하게 되고, 이 배열을 토대로 충돌함수를 실행하게 됩니다.

**클라이언트 변경 점 및 추가**

struct PlayerInfo //플레이어의 정보를 담당하는 구조체

{

int type; //데이터 타입의 구분을 담당

int playerIndex;

float CameraDir; //플레이어의 방향을 담당

Vec2 PlayerPos; //플레이어의 위치를 담당

};

struct BulletInfo //총알의 정보를 담당하는 구조체

{

Vec3 BulletDirection; //총알의 방향

Vec3 BulletPos; //총알의 위치를 담당

bool newBullet; //총알이 새로 만들어진 총알인지 아닌지

//True면 서버내에 총알관리 벡터에 추가.

int bulletOwner; //누가 발사한 총알인지 구분

};

struct playerRecv //서버로부터 수신받는 플레이어 정보 구조체

{

int playerIndex;

Vec2 playerPos;

float playerCam;

int Killcount;

};

struct RecvPacket //서버로부터 수신받는 인게임 구조체

{

playerRecv player1;

playerRecv player2;

Vec3 MonstersPosition[10];

bool gameOver;

};

struct SendPacket //서버로 보내는 게임정보 구조체

{

Vec2 palyer1Pos;

float player1Cam;

bool makeBullet;

Vec3 bulletDirection;

Vec3 bulletPosition;

};