클라이언트 Class

Class Object // 역할 : 오브젝트의 위치, 크기 정보를 갖고 있는 Class

{

Private:

Vector2 m\_vec2Pos // 오브젝트의 위치를 2차원 벡터로 갖고 있는 변수

Vector2 m\_vec2Size // 오브젝트의 XY축 길이를 표현한 2차원 벡터 변수

}

Class Mario : public Object // 역할 : 마리오들의 정보를 관리해주는 Class

{

Private:

UINT m\_iMarioNum // 마리오에게 부여된 식별 번호

UINT m\_iPlayerNum // 마리오를 사용하고 있는 플레이어의 식별 번호

bool m\_bSelect // 마리오의 활성화 여부

true = 활성화, false = 비활성화

bool m\_bLookDirection // 마리오가 보고 있는 방향

true = 오른쪽, false = 왼쪽

MarioSprite m\_eSpriteState // 현재 스프라이트 번호에 대한 열거형 변수

}

Class Door : public Object // 역할 : 문의 위치 문이 열렸는지에 대한

{ 정보를 담고 있는 Class

Private:

bool m\_bOpen // 문이 열렸는지에 대한 정보

}

Class SceneManager : // 역할 : 특정 스테이지에 대한 오브젝트들을

{ 관리하는 Manager Class

Private:

WORD m\_wStageNumber // 현재 Scene은 몇 번째 Stage인가 대한 정보

Door m\_dDoor // 문 오브젝트

Key m\_kKey // 열쇠 오브젝트

Mario m\_pMario[MaxMario] // 해당 스테이지의 마리오 오브젝트들

Public:

int ApplyObjectsStatus ( char\* buf ) // FrameWork로부터 받은

오브젝트 위치, 상태에 대한 정보를 분할하고 오브젝트에게 Set해주고 받은 Buf

크기를 반환합니다.

}

Class FrameWork : // 역할 : 게임의 전반적인 Scene과 키 입력,

{ 서버와의 통신을 담당, 관리하는 클래스

Private:

SceneManager m\_pScene[MaxStage] // 최대 스테이지 개수의 SceneManager변수

WORD m\_wInputSpecialKey // 플레이어가 입력하고 있는 키보드 입력 값

UINT m\_iStageNum // 현재 스테이지 레벨을 표현하는 변수

SOCKADDR\_IN m\_sockServer // 연결한 서버의 주소 정보가 담겨있는 변수

Public:

int RecvObjectStatus(char\* buf) // 서버로부터 전송된 데이터들을

buf에 저장하고 수신된 데이터의 크기를 반환한다. 반환 값을 통해 전송 받은 데이터

의 크기를 활용하여 Server와 비교, 오류를 분석한다.

int ApplySceneStatus(char\* buf) // 서버로부터 전송된 데이터들 중 현재

현재 **스테이지 레벨 정보만 사용**하고 나머지 데이터는 SceneManager의

ApplyObjectsStatus 함수를 통해 넘겨주고 반환되는 값을 반환한다.

int SendKeyStatus() // m\_wInputSpecialKey를 Char\*형으로

캐스팅하고 m\_sockServer를 목적지로 설정 후 운영체제의 송신 버퍼에 저장하고,

송신 버퍼에 저장한 데이터 크기를 반환합니다.

int ConnectServer() // Console환경을 통해 ServerIP와

Port번호를 입력 받고 서버와 연결을 시도합니다. 연결에 문제가 있을 경우 ServerIP,

Port번호를 재입력 받고 다시 서버와의 연결을 시도합니다.

bool IsGameEnd() // 모든 스테이지가 클리어 될 경우 서버에

접속해제를 알리고, FrameWork에서 사용하기 위해 할당 받은 메모리들을 해제한 후

클라이언트를 종료합니다.

void SpecialKeyInput(int key, int x, int y) // 플레이어가 키보드를

누를 경우 OpenGL에서 DEFINE한 Key값들 중 클라이언트가 사용할 입력 값에 대해서만

입력을 감지 받고 m\_wInputSpecialKey와 해당 입력 값 대해 클라이언트와 서버에

DEFINE한 값과 비트 or연산을 해줍니다.

void SpecialKeyOutput(int key, int x, int y) // SpecialKeyInput 와는

반대로 플레이어가 해당 자판에서 손을 땐 경우 m\_wInputSpecialKey와 해당 Key

대해 클라이언트와 서버에 DEFINE한 값과 비트 xor연산을 해줍니다.

void KeyInput(unsigned char key, int x, int y) // 플레이어가 키보드

자판 중 문자( A~Z, 0~9 등 ) 입력할 경우 클라이언트가 사용할 입력 값에 대해서만

입력을 감지 받고 m\_wInputSpecialKey와 해당 입력 값에 대해 클라이언트와 서버에

DEFINE한 값과 비트 or연산을 해줍니다.

void KeyOutput(unsigned char key, int x, int y) // KeyInput와는 반대로

플레이어가 해당 자판에서 손을 땐 경우 m\_wInputSpecialKey와 해당 입력 값에 대해

클라이언트와 서버에에 DEFINE한 값과 비트 xor연산을 해줍니다.

}

키보드 입력 값 연산을 위해 클라이언트와 서버에 DEFINE된 변수들

const WORD KEY\_1 = 0x01 // 숫자 키 1에대한 DEFINE 값

const WORD KEY\_2 = 0x02 // 숫자 키 2에대한 DEFINE 값

const WORD KEY\_3 = 0x04 // 숫자 키 3에대한 DEFINE 값

const WORD KEY\_4 = 0x08 // 숫자 키 4에대한 DEFINE 값

const WORD KEY\_5 = 0x10 // 숫자 키 5에대한 DEFINE 값

const WORD KEY\_6 = 0x20 // 숫자 키 6에대한 DEFINE 값

const WORD DIR\_LEFT = 0x40 // 왼쪽 화살표 키에 대한 DEFINE값

const WORD DIR\_RIGHT = 0x80 // 오른쪽 화살표 키에 대한 DEFINE 값

const WORD KEY\_C = 0x100 // 문자 ’c’ ‘C’키에 대한 DEFINE값

const WORD KEY\_X = 0x200 // 문자 ‘x’ ‘X’키에 대한 DEFINE 값

서버에서 오는 데이터를 담기 위해 정의한 구조체

struct RecvMarioDataFormat { // 마리오의 정보를 담을 구조체

WORD iMarioNum // 이 정보의 주인인 마리오 Number

WORD iMarioPlayerNum // 마리오를 할당 받은 Player식별 번호

WORD wxPos // 마리오의 X포지션

WORD wyPos // 마리오의 Y포지션

bool bSelect // 마리오의 선택 여부

bool bLookDirection // 보는 방향

WORD eSpriteState // 마리오의 Sprite상태

}

struct RecvStageDataFormat { // 해당 스테이지에 대한 오브젝트 위치정보를 담을

담을 구조체

WORD wKeyXPos // 열쇠 오브젝트의 X좌표

WORD wKeyYPos // 열쇠 오브젝트의 Y좌표

WORD wDoorXPos // 문 오브젝트의 X좌표

WORD wDoorYPos // 문 오브젝트의 Y좌표

bool IsOpen // 문이 열렸는 지에 대한 정보

}

설계문서 내용 X

클라이언트에서 서버로 전송할 데이터

FrameWork::m\_wInputSpecialKey – 2Bytes

Total – 2Bytes

서버에서 클라이언트로 전송할 데이터 - 최대

WORD wStageNum – 2Bytes

RecvMarioDataFormat[6] – 72Bytes

RecvStageDataFormat – 9Bytes

Total - 83Bytes

서버에서 움직이지 않은 오브젝트 정보를 안 보낸다는 가정하에 설계 할 경우

RecvMarioDataFormat을 간단하게 RMDF라고 하겠습니다.

RecvStageDataFormat을 간단하게 RSDF라고 하겠습니다.

클라이언트 서버

<- StageNum

RMDF \* 개수메모리 할당 <- RecvMarioDataFormat의 개수 // 움직인 마리오 수

할당 받은 메모리에 저장 <- RecvMarioDataFormat[개수] // 움직인 마리오 정보

변화 있으면 메모리 할당 <- 오브젝트 변화 있는지에 대한 여부// bool 값

변화 없으면 Update함수 호출 <- 변화 없으면 다음 플레이어에게도 동일하게 전송