**네트워크 게임 프로그래밍**

**Term Project 추진 기획서**

게임공학과

2012180005 김건우

2012180023 여성우

목차

1. 게임 소개
2. 요구사항 분석
3. High Level 디자인
4. Low Level 디자인
5. 팀원 역할 분담
6. 개발환경
7. 개발일정
8. **게임 소개**

* 게임 제목 : Mario Exodus
* 게임 장르 : 협동
* 게임 설명

이 게임은 각 플레이어가 협동하여 장애물을 지나 열쇠를 획득한 후 그 열쇠를 사용하여 다음 스테이지로 넘어가는 것이 목표인 2D 사이드뷰 게임입니다.

최소 1인 ~ 최대 2인까지 플레이가 가능합니다.

각 스테이지마다 6명의 마리오가 주어지고 플레이어는 혼자 혹은 다른 플레이어와 협동하여 마리오를 적절하게 조작할 수 있습니다.

마리오는 1 ~ 6까지 고유한 번호를 지닙니다. 플레이어는 한 번에 6개의 마리오를 동시에 조작할 수도 있습니다.

단, 다른 플레이어가 조작중인 마리오는 조작할 수 없습니다.

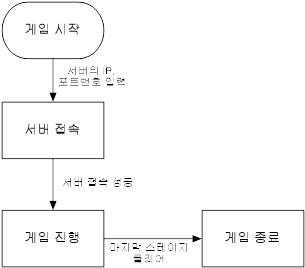
* 게임 조작법

좌우 방향키를 이용하여 좌우로 이동할 수 있습니다.

1 ~ 6까지 숫자키를 이용하여 해당 번호를 가진 마리오의 제어권을 얻을 수 있습니다.

X키를 이용하여 열쇠를 획득 할 수 있습니다.   
(단, X키를 누르고 있는 동안에만 열쇠를 가지고 있습니다. 열쇠를 들고 있는 상태에서 X키를 떼면 열쇠를 떨어트리게 됩니다.)

C키를 이용하여 점프를 할 수 있습니다.

게임의 흐름

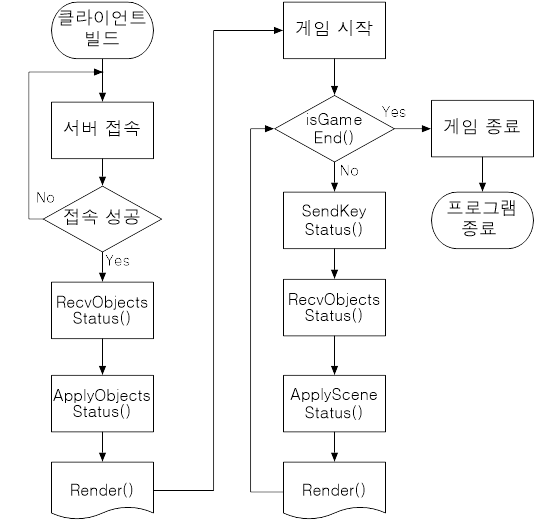
게임 스크린샷 예

<그림 1> 게임 예상도

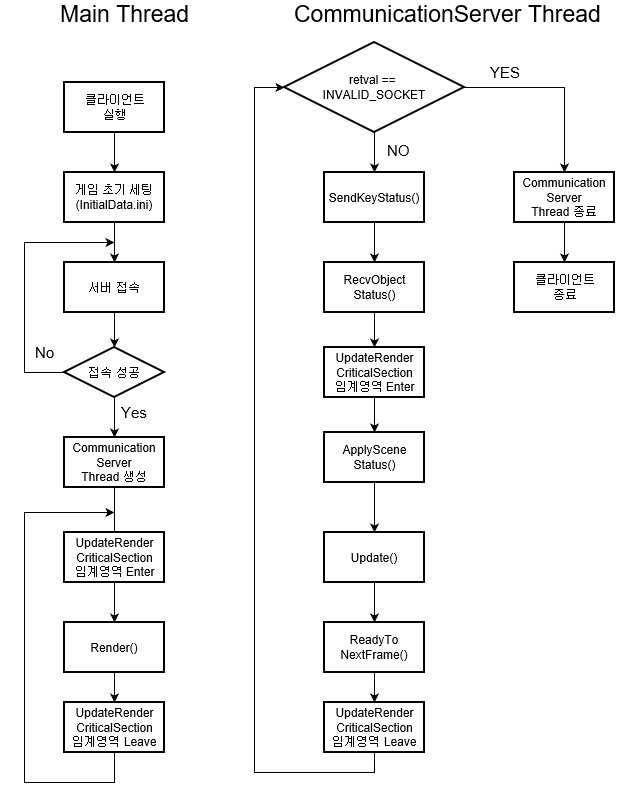
https://www.youtube.com/watch?v=7pFyzC-K\_Vw

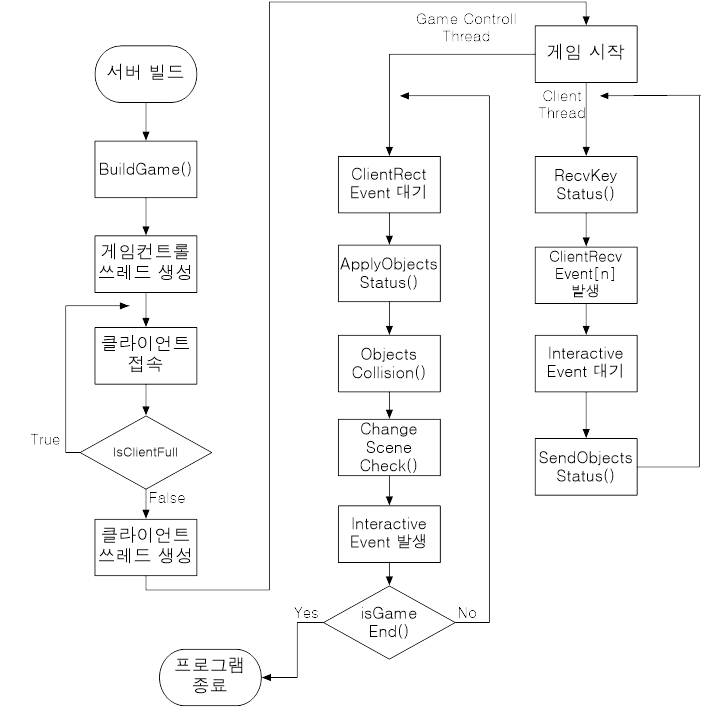
1. **요구사항 분석**

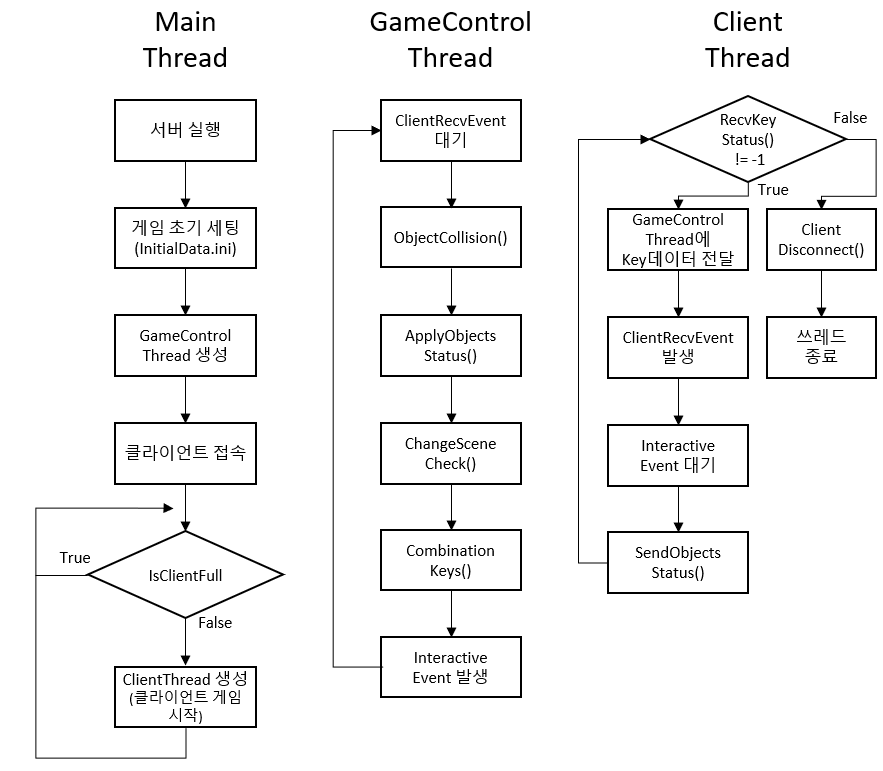
* 기능 요구사항
  + 네트워크 방식은 TCP를 사용한다.
  + 게임 시작을 누르면 서버의 IP와 포트번호를 입력할 수 있다.
  + 서버는 최대 2개의 클라이언트를 수용할 수 있다.
  + 서버에 1명 이상의 클라이언트가 접속되어 있는 한 계속 현재 게임의 상태는 계속 유지된다.  
    ( A플레이어와 B플레이어가 게임에 접속중 이었다. A플레이어가 게임을 종료해도 B플레이어는 혼자서 계속 플레이가 가능하다. 이후 A플레이어가 다시 접속하면 두 플레이어가 다시 함께 플레이가 가능하다.   
    이후 B플레이어가 게임을 종료하면 A플레이어는 혼자서 계속 플레이가 가능하다.   
     ~~하지만 두 플레이어가 모두 게임을 종료한 후 특정한 플레이어가 접속을 하면 다시 스테이지 1부터 시작을 해야 한다.~~  
     서버는 항상 마지막 게임상태를 지니고 있다. 새 게임을 시작하려면 서버의 재시작이 필요함. )
  + 서버가 비정상적으로 종료된 경우 클라이언트도 같이 종료 되어야 한다.
  + 클라이언트는 현재 플레이어가 누른 키 값을 정해진 형식에 맞춰 서버로 전송한다.  
    이 때 플레이어가 어떤 키를 누르지 않아도 정해진 형식에 맞춰 서버로 전송한다.  
    ~~( 즉 클라이언트는 매 프레임마다 서버에 키 데이터를 전송한다. )|~~( 클라이언트는 1초에 30번씩 서버에 키 데이터를 전송한다. )
  + 서버는 클라이언트의 키 데이터를 수신하고, 이를 클라이언트에서 사용될 데이터에 반영한다.  
    ( 6명의 마리오의 위치, 열쇠 위치, 문의 닫힘/열림 여부, 블록의 위치,   
    현재 스테이지 레벨, ~~오브젝트 간의 충돌체크~~ )
  + 멀티쓰레드를 이용한 이벤트 처리로 동기화 과정을 거친다.  
    ( 2개의 클라이언트 전송한 키 데이터를 서버에서 오브젝트 간의 충돌체크 등에 반영할 때 문제 없이 처리)
  + 서버는 처리한 데이터를 클라이언트에 다시 전송하고, 클라이언트는 이를 수신한 후, 각 오브젝트에 데이터를 반영한 후 랜더링을 실시한다.  
    ( 즉, 클라이언트에서 키를 눌러도 실제 연산은 서버에서 처리 한다. 클라이언트는 서버에서 처리한 데이터를 수신받아 이를 다시 클라이언트에 반영하는 것이다. )
  + 스레드 동기화 시 문제점을 해결해야 한다.  
    클라이언트의 오류로 데이터를 송신하지 않을 시 서버에서 무한 대기상태에 빠질 수 있다. 그러면 서버에 접속한 클라이언트 또한 무한 대기상태에 빠질 수 있다.  
    따라서 한 스레드에서 일정 시간동안 데이터를 받아오지 못했을 때 서버에서 어떻게 처리를 해야할 지 고민할 필요가 있다.
  + 서버의 갱신 속도가 일정하지 않으면 문제가 생길 수 있다.  
    서버가 어떤 상황에서는 빠르게 처리하고, 어떤 상황에서는 느리게 처리하고 이를 클라이언트로 전송할 경우 클라이언트에서 게임 객체가 비정상적인 움직임을 보여줄 수 있다.  
    따라서 클라이언트와 서버가 1초에 몇번의 데이터를 주고받을 지 정하고 이를 일치시킬 필요가 있다.
  + 클라이언트는 공유자원에 대한 접근을 처리하지 않으면 문제가 생길 수 있다.  
    클라이언트 내부 오브젝트들에 대한 쓰기 연산( 오브젝트 정보 갱신 )도중  
    읽기 연산( Rendering )이 진행되지 않도록 공유자원 접근에 대한 처리가   
    필요하다.

1. **High Level 디자인**
2. 클라이언트 디자인

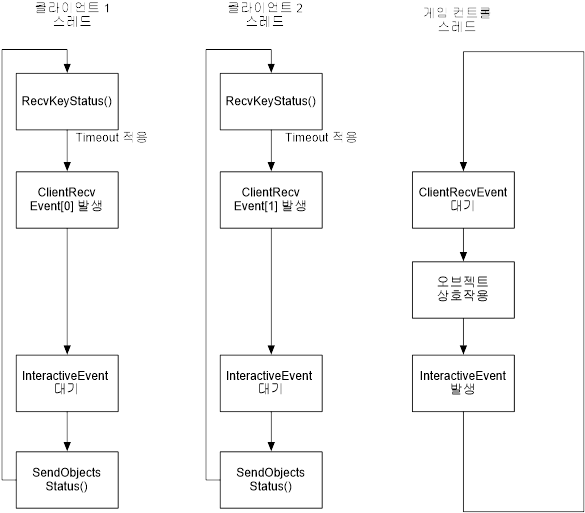
* 클라이언트 디자인



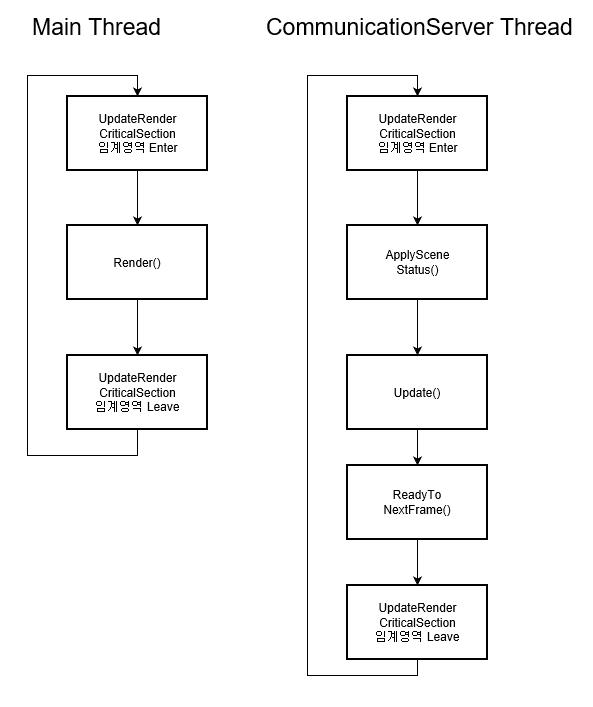
* 서버 디자인
* 서버 디자인



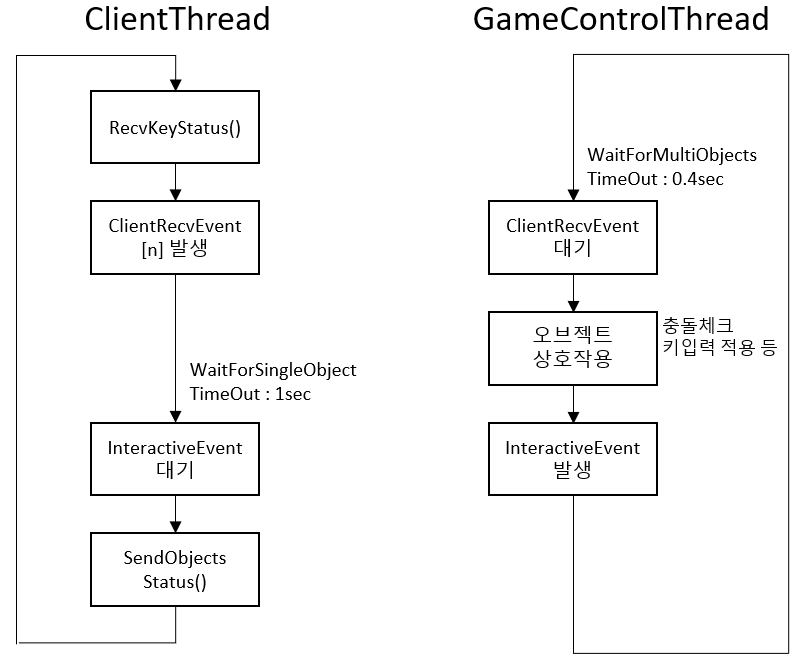
* ~~스레드 동기화 방식~~

~~~~

* + ~~클라이언트는 60프레임을 기준으로 서버로 1초에 60번 데이터를 송신하고,  
    서버 또한 1초에 60번 데이터를 수신한다.~~
  + ~~서버에서 RecvKeyStatus()를 통해 클라이언트의 데이터를 수신받을 때   
    타임아웃을 적용하여 무한대기상태에 빠지는 것을 방지한다.~~
  + ~~타임아웃은 최대 4ms 로 설정~~
* 스레드 동기화 방식 ( 클라이언트 )



* + 클라이언트는 1 초에 60번의 렌더링을 진행하며 1초에 30번 서버와 통신을  
    합니다.
  + CommunicationServer Thread는 서버에게 Key값을 보내고, 오브젝트들의  
    정보를 받아와 Stage에 있는 오브젝트 정보들을 갱신 시켜줍니다.
  + Main Thread는 갱신된 오브젝트들의 정보를 바탕으로 Texture를 입히는  
    Rendering를 진행합니다.
  + 오브젝트들의 정보들을 적용 시키는 것과 Rendering사이에 순서는 중요하지  
    않아 EventHandle이 아닌 임계영역을 사용합니다.
* 스레드 동기화 방식 ( 서버 )



InteractiveEvent   
리셋

* + 클라이언트 60프레임을 기준으로 클라이언트와 서버는 1초에 30번 데이터를 송수신한다.
  + ClientThread는 클라이언트에서 송신한 데이터를 수신하고,  
    GameControlThread는 수신받은 데이터를 기반으로 오브젝트들 간의 상호작용을 처리한 후,  
    처리한 데이터를 ClientThread에서 다시 각 클라이언트로 송신한다.
  + 동기화를 위해서 이벤트 핸들을 사용한다. (ClientRecvEvent, InteractiveEvent)
  + ClientRecvEvent는 자동 리셋 이벤트를 사용한다.
  + InteractiveEvent는 매뉴얼 리셋 이벤트를 사용한다.  
    게임의 상호작용을 처리할 쓰레드는 1개이지만, 클라이언트는 최대 2개까지 접속 할 수 있다. 매뉴얼 리셋을 이용해 기다리는 쓰레드를 모두 깨운다
  + GameControlThread에서 WaitForMultiObjects함수를 통해 ClientRecvEvent 발생을 대기한다. 이 때 접속된 클라이언트의 수에 따라 3번째 인자값을 달리 설정해준다.  
     클라이언트가 1개만 접속했을 때는 false를, 2개가 접속했을 때는 true로 설정해준다.  
     이를 통해 1개만 접속했을 때는 1개의 ClientRecvEvent만 발생해도 오브젝트 상호작용을 처리하고,  
    2개의 클라이언트가 접속했을 때는 모든 ClientRecvEvent (2개)가 발생했을 때 오브젝트 상호작용을 처리한다.
  + ClientThread에서 InteractiveEvent의 발생을 기다릴 때 timeout을 1초로 설정한다.   
     타임아웃 설정을 통해 클라이언트가 대기상태에 빠져 멈춘 것처럼 보이는 것을 방지한다.  
     또한 타임아웃을 초과해 WAIT\_FAILED가 발생한 경우 이전 오브젝트들의 상태를 클라이언트로 전송한다.
  + GameControlThread에서 ClientRecvEvent의 발생을 기다릴 때 timeout을 0.4초로 설정한다.  
     타임아웃 설정을 통해 무한대기상태에 빠지는 것을 방지하고, 혹 이 시간안에 클라이언트로부터 수신을 받지 못한 경우에는 클라이언트로부터 아무 키입력도 받지 않은 상태로 가정하고 오브젝트간 상호작용을 처리한다.  
    (클라이언트는 아무 키입력을 하지 않은 경우에도 매 프레임 데이터를 전송한다.)

1. **Low Leve 디자인**

- 클라이언트 디자인

서버와 클라이언트가 상호작용 하기 위한 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

bool bExit // 마리오가 문밖으로 나갔는지에 대한 정보

~~WORD eSpriteState // 마리오의 Sprite상태~~

}

**struct RecvStageDataFormat** { // 오브젝트의 위치정보를 담을 구조체

WORD wStageNum // 현재 스테이지 레벨에 대한 정보

WORD wKeyXPos // 열쇠 오브젝트의 X좌표

WORD wKeyYPos // 열쇠 오브젝트의 Y좌표

bool IsOpen // 문이 열렸는 지에 대한 정보

}

**Struct RecvStageBlockFormat** { // 블록 오브젝트의 위치정보를 담을 구조체

WORD wxPos // 블록 오브젝트의 X 좌표

WORD wyPos // 블록 오브젝트의 Y 좌표

}

클라이언트 클래스

Class FrameWork : // 게임의 전반적인 Scene과 키 입력, 서버와의 통신 담당

{

Private:

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

**WORD m\_wInputSpecialKey // 플레이어가 입력하고 있는 키보드 입력 값  
 전송할 데이터**

**UINT m\_iStageNum // 현재 스테이지 레벨을 표현하는 변수  
 수신 받을 데이터의 일부**

SOCKET m\_sockServer // 서버와 통신하기 위한 소켓

Char m\_RecvBuf[MAX\_BUF] // 서버로부터 수신 받은 데이터를 저장할  
 버퍼

Char\* m\_pBufptr // m\_RecvBuf에서 참조하고 있는 위치

HANDLE m\_hThreadHandle // CommunicationServer쓰래드의 핸들

Public:

int RecvObjectStatus(char\* buf)

서버로부터 전송된 데이터들을 buf에 저장하고 수신된 데이터의 크기를 반환한다.

void ApplySceneStatus(char\* buf)

서버로부터 전송된 데이터들 중 현재 스테이지 레벨 정보만 사용하고 나머지 데이터는 함수 내부에서 SceneManager의 ApplyObjectsStatus 함수를 통해 넘겨준다.

int SendKeyStatus()

m\_wInputSpecialKey를 Char형으로 캐스팅하고 m\_sockServer를 목적지로 설정 후 운영체제의 송신 버퍼에 저장하고, 송신 버퍼에 저장한 데이터 크기를 반환합니다.

int ConnectServer()

Console환경을 통해 ServerIP와Port번호를 입력 받고 서버와 연결을 시도합니다. 연결에 문제가 있을 경우 ServerIP, Port번호를 재입력 받고 다시 서버와의 연결을 시도합니다.

~~bool IsGameEnd()~~

~~모든 스테이지가 클리어 될 경우, 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.

}

usigned \_\_stdcall CommunicationServer(void\* arg)

arg를 FrameWork\* 로 캐스팅하여  
FrameWork함수 중 서버와 통신하는 함수와 업데이트를 담당할 쓰레드 함수 입니다.

- 서버 디자인

클라이언트와 상호작용 하기 위한 구조체 및 변수

**struct MarioDataFormat** { // 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

bool isExit // 마리오가 문밖으로 나갔는지에 대한 정보

~~WORD eSpriteState // 마리오의 Sprite상태~~

}  
**struct StageDataFormat** { // 오브젝트의 위치정보를 담을 구조체

~~UINT wStageNum // 현재 스테이지 레벨~~

WORD wStageNum // 현재 스테이지 레벨

WORD wKeyXPos // 열쇠 오브젝트의 X좌표

WORD wKeyYPos // 열쇠 오브젝트의 Y좌표

bool IsOpen // 문이 열렸는 지에 대한 정보

}  
**~~struct SendDataFormat~~** ~~{ // 전송할 데이터를 담을 구조체~~

~~MarioDataFormat m\_Mario[6] // 6개의 마리오에 대한 정보~~

~~StageDataFormat m\_Stage // 현재 스테이지에 대한 정보~~

~~}~~**struct SendDataFormat** { // 스테이지에 블록의 위치를 담을 구조체

WORD wxPos // 블록의 X좌표

WORD wyPos // 블록의 Y좌표

}

HANDLE ClientRecvEvent[2] // 클라이언트에게 키 데이터를 전송받았음을  
 알리는 이벤트 핸들 (Auto Reset)

HANDLE InterativeEvent // 모든 오브젝트의 상호작용이 종료되었음을  
 알리는 이벤트 핸들 (Menual Reset)

서버 함수

DWORD WINAPI ClientThread(LPVOID arg)

클라이언트와 연결을 담당할 스레드 함수

DWORD WINAPI GameControllThread(LPVOID arg)

전송받은 데이터를 바탕으로 오브젝트 간의 상호작용을 적용할 스레드 함수

~~Bool IsClientFull()~~

~~클라이언트 접속 요청을 받았을 때 현재 클라이언트가 총 받을 수 있는 클라이언트의  
 개수를 초과했는지 확인. (false를 리턴 시 접속을 허가)~~

~~int RecvKeyStatus(char\* buf)~~

~~클라이언트가 보낸 데이터를 buf에 저장~~

~~void DivideKey(char\* buf)~~

~~buf에 저장된 데이터를 or, xor 연산을 통해 분리, 전송된 키값이 어떤 것인지 분석~~

~~void ApplyObjectsStatus()~~

~~전송된 키 값을 바탕으로 서버에 저장된 오브젝트에 현재 데이터를 적용~~

~~void ObjectsCollision()~~

~~오브젝트 간에 충돌체크를 수행~~

~~void ChangeSceneCheck()~~

~~현재 스테이지가 클리어 되었는지를 확인, 클리어 되었다면 다음 스테이지로 상태를   
 변화시키고 이에 맞춰 모든 오브젝트의 위치, 상태 값을 초기화~~

~~bool isGameEnd()~~

~~모든 스테이지를 클리어 했을 시 클라이언트에 접속 해제를 알리고 프로그램을 종료~~

**기존에 정했던 함수를 모두 클래스의 맴버함수로 재정의함**

Class ServerControl //오브젝트의 상호작용을 위한 클래스

{

public:

void InitGameScene()  
게임 초기화

bool IsClientFull()   
클라이언트 접속 요청을 받았을 때 현재 클라이언트가 총 받을 수 있는 클라이언트의  
 개수를 초과했는지 확인. (false를 리턴 시 접속을 허가)

void ClientDisconnect(int client)  
클라이언트 연결이 종료됐을 경우 해당 클라이언트 쓰레드가 처리하던 값을 해제

void getRecvDatas(int m\_iClientNum, char\* recvData)   
클라이언트 쓰레드에서 전송받은 데이터를 게임 컨트롤 쓰레드로 가져옴

char\* getSendData()  
오브젝트 상호작용이 끝난 결과 데이터를 반환

char\* getBackupData()  
오브젝트 상호작용 하기 전의 데이터를 반환

void ApplyObjectsStatus()   
클라이언트의 입력 키값을 오브젝트에 반영

void ObjectsCollision()  
오브젝트 간 충돌체크

void ChangeSceneCheck()   
현재 스테이지 클리어 여부 확인, 클리어 시 다음 스테이지 상태로 초기화

void CombinationKeys()  
오브젝트의 상호작용 정보를 클라이언트가 받을 수 있는 데이터 포맷으로 변환

}

Class ClientControll //클라이언트와 통신을 위한 클래스

{

Public:

int RecvKeyStatus()  
클라이언트가 송신한 데이터를 수신 후 수신데이터 크기 반환

int SendObjectsStatus(bool backup)  
게임컨트롤 쓰레드에서 처리한 데이터를 가져와서 클라이언트로 송신한다. 이 때 backup이 true일 경우는 처리하기 전의 데이터를 가져와서 송신한다.  
이 후 송신데이터 크기를 반환한다.

}

1. **팀원 역할 분담**

김건우

* 클라이언트 제작 및 클라이언트의 네트워크 기능 구현
* 서버에 키 데이터 전송 및 서버에서 수신 받은 데이터를 클라이언트에 적용

여성우

* 서버에서의 네트워크 기능 구현
* 클라이언트에서 전송한 데이터 처리 및 각 오브젝트에 적용,  
  적용한 오브젝트 데이터를 다시 클라이언트에 전송

1. **개발환경**

* Visual Studio 2017
* OpenGL
* Windows SDK 10.0.15063.0
* GitHub (프로젝트 관리)

1. **개발 일정**

|  |  |  |
| --- | --- | --- |
| 날짜 | 김건우 | 여성우 |
| 11월 8일 | 게임 스테이지 제작 | 게임 프레임워크 코드 분석 |
| 11월 9일 | - | - |
| 11월 10일 | 게임 스테이지 제작 | 게임 프레임워크 코드 분석 |
| 11월 11일 | 게임 버그 수정 | 게임 프레임워크 코드 분석 |
| 11월 12일 | 클라이언트 최종 테스트 및 완성 | |
| 11월 13일 | SendKeyStatus() RecvObjectsStatus() | RecvObjectsStatus() SendObjectsStatus() |
| 11월 14일 | 서버와 클라이언트 간 통신 확인 (데이터를 정상적으로 주고받는지 확인) | |
| 11월 15일 | - | - |
| 11월 16일 | - | - |
| 11월 17일 | ApplySceneStatus() ApplyObjectsStatus() | ApplyObjectsStatus() |
| 11월 18일 | 서버와 클라이언트 간 주고받은 데이터가 정상적으로 오브젝트에  적용되는지 확인 | |
| 11월 19일 | 중간 점검 (버그 확인 및 수정) | ObjectsCollision() |
| 11월 20일 | 다중 클라이언트 접속 확인 | 서버에 멀티쓰레드 적용 ApplyObjectsStatus() 확장 |
| 11월 21일 | - | - |
| 11월 22일 | ChangeScene() | ObjectsCollision() 확장 |
| 11월 23일 | - | - |
| 11월 24일 | - | - |
| 11월 25일 | 다중 클라이언트 접속 확인 및 각 클라이언트의 수신 데이터 확인 | 동기화를 위한 이벤트 제작 (InteractiveEvent, ClientRecvEvent) |
| 11월 26일 | GameControllThread(), 멀티쓰레드 동기화 작업 | |
| 11월 27일 | GameControllThread(), 멀티쓰레드 동기화 작업 | |
| 날짜 | 김건우 | 여성우 |
| 11월 28일 | - | - |
| 11월 29일 | GameControllThread(), 멀티쓰레드 동기화 작업 | |
| 11월 30일 | - | - |
| 12월 1일 | - | - |
| 12월 2일 | 멀티쓰레드 동기화 작업 마무리 | |
| 12월 3일 | 프로토타입 테스트 (예외상황 테스트 및 버그 수집) | |
| 12월 4일 | 예외상황 보완 및 버그 픽스 | |
| 12월 5일 | - | - |
| 12월 6일 | 예외상황 보완 및 버그 픽스 | |
| 12월 7일 | - | - |
| 12월 8일 | - | - |
| 12월 9일 | 게임 테스트 (주변 학우에게 테스트 부탁 및 버그 수집) | |
| 12월 10일 | 버그픽스 | |
| 12월 11일 | 버그픽스 | |
| 12월 12일 | 버그픽스 | |
| 12월 13일 | 제출 | |