Multi-Players in Tetris3D with Networking

I. TCP or UDP?

1. General Comparison

There are many methods to encode the message, which are all through UDP or TCP. TCP transfers the message in a highly reliable way, while UDP can do it much faster though sometimes unreliable. They both have many advantages and disadvantages, check online if you'd like.

2. In the Games

TCP is a better implementation especially when sending files, but not go for the games. TCP might need a lot of time to encrypt/decrypt the messages and also need time to check the accuracy of the messages, which might be able to cause the game to stop suddenly then back to work again. In addition, TCP would combine many messages together, which needs programmers to split the messages in the code.

UDP seems to fit the games much more than TCP. UDP can send the message immediately once the message is generated.

Somehow, UDP has its own shortages:

i. UDP does not ensure the arrival of the messages.
ii. UDP does not ensure the order of the arriving messages.
iii. UDP does not require the connecting status, which means, someone can send the messages to the server externally.
iv. UDP does not have the protection against the hackers.
v. Some other things……

3. However, the UDP can be used more flexible than TCP in the games.

II. RakNet Networking Engine

RakNet is a C++ library basing on UDP protocol, allowing the programmers to transfer the data effectively and efficiently in their own programs, usually for the game programs.

What does RakNet do to overcome the shortages of the UDP?

i. RakNet re-send the missing packets automatically.
ii. RakNet sort the packets efficiently automatically.
iii. RakNet can provide the protection informing the programmer whether the message is modified externally.
iv. RakNet provide a highly efficient, simple connection layer to stop the illegal transferring.
v. Some other features……

Therefore, RakNet seems to be a good solution for the game programming, which can make the programmers concentrate more on the game rather than the networking features.

III. RakNet Operations

1. Create the server:

```
// Creating the server
_peerServer = RakNet::RakPeerInterface::GetInstance();
RakNet::SocketDescriptor sd(SERVER_PORT, 0);
_peerServer->Startup(MAX_CLIENTS, &sd, 1);
_peerServer->SetMaximumIncomingConnections(MAX_CLIENTS);
```

2. Create the client:

```
// Creating the client
_peerClient = RakNet::RakPeerInterface::GetInstance();
```

```
RakNet::SocketDescriptor sd;
_peerClient->Startup(1, &sd, 1);
_peerClient->Connect(gServerIp, SERVER_PORT, 0, 0);
```

3. Send Messages:

```
// Sending the msg: Message
RakNet::BitStream bsOut;
bsOut.Write((RakNet::MessageID)gm); //Message Identifier
bsOut.Write(msg);
peer->Send(&bsOut, HIGH_PRIORITY, RELIABLE_ORDERED, 0,
                RakNet::UNASSIGNED_SYSTEM_ADDRESS, true);
```

4. Receive Messages:

```
while (true) {
    for (packet = peer->Receive();
        packet; peer->DeallocatePacket(packet), packet=peer->Receive()) {
        switch (packet->data[0])
        {
        case ID_REMOTE_DISCONNECTION_NOTIFICATION:
            printf("Another client has disconnected.\n");
            break;
        case ID_REMOTE_CONNECTION_LOST:
            printf("Another client has lost the connection.\n");
            break;
        case ID_REMOTE_NEW_INCOMING_CONNECTION:
            printf("Another client has connected.\n");
            break;
        case ID_CONNECTION_REQUEST_ACCEPTED:
            printf("Our connection request has been accepted.\n");
            break;
        case ID_NEW_INCOMING_CONNECTION:
            printf("A connection is incoming.\n");
            break;
        case ID_NO_FREE_INCOMING_CONNECTIONS:
            printf("The server is full.\n");
            break;
        case ID_DISCONNECTION_NOTIFICATION:
            if (isServer) {
                printf("A client has disconnected.\n");
            } else {
                printf("We have been disconnected.\n");
            }
            break;
        case ID_CONNECTION_LOST:
            if (isServer) {
                printf("A client lost the connection.\n");
            } else {
                printf("Connection lost.\n");
            }
```

```
                break;
            case ID_GAME_MESSAGE_1:
                {
                    // try to receive the message
                    RakNet::RakString rs;
                    RakNet::BitStream bsIn(packet->data, packet->length, false);
                    bsIn.IgnoreBytes(sizeof(RakNet::MessageID));
                    bsIn.Read(rs);
                    printf("%s\n", rs.C_String());
                } break;
            default:
                printf("Message with identifier %i has arrived.\n", packet->data[0]);
                break;
            }
        }
}
```

IV. Implementation

Since to receive the messages needs to be put in an endless loop, we can put the receiving part in the frame-drawing function, for example, frameStarted function.

Here I use a GameMessage to represent the opponent's action, here's how it looks:

```
// Game Message, representing the opponent's action
enum GameMessages {
    ID_GAME_MESSAGE_HELLO = ID_USER_PACKET_ENUM + 1,
    ID_GAME_MOV_X_POSITIVE,
    ID_GAME_MOV_X_NEGATIVE,
    ID_GAME_MOV_Z_POSITIVE,
    ID_GAME_MOV_Z_NEGATIVE,
    ID_GAME_ROT_Y,
    ID_GAME_FALL,
};
```

With these enumerations, we can send message to define the opponent's action.

```
// Keyboard event, moving the blocks on X axis positive
bool CurrentBlockMoveXPositive(GameStatus* gs, RakNet::RakPeerInterface* _peer)
{
    // add the message-sending function, and others remain the same
    SendActionMessage(_peer, ID_GAME_MOV_X_POSITIVE);
    return gs->getCurrentBlock()->MoveX(Blocks::AxisDirection::ePositive);
}
```

And in the frameStarted function, we need another part to receive the message.

```
// In frameStarted function
RakNet::Packet* packet;
for (packet = _peerServer->Receive();
    packet; _peerServer->DeallocatePacket(packet), packet = _peerServer->Receive()) {
    switch (packet->data[0])
    {
    case ID_GAME_MOV_X_POSITIVE:
```

```
            {
                _currentStatus2P->
                    getCurrentBlock()->
                        MoveX(Blocks::AxisDirection::ePositive);
            } break;
        // deal with the other GameMessages
        // ……
        default:
            break;
        }
}
```

And due to that we only transfer the key events, we have to sync the blocks generated randomly. The method I use is not to transfer the random number but to generate a list of number and sync before the game starts, which means, predefined number. Be advised, we can never send the std::vector instance directly, I tried many times but failed, and the reason is that the std::vector contains a size-changeable memory, when transferring the vector, the memories might be crushed.

Here's how I do it:

```
// Generate and send the number list
while (true) {
    for (packet = _peerServer->Receive();
        packet;
        _peerServer->DeallocatePacket(packet), packet = _peerServer->Receive()) {
        switch (packet->data[0])
        {
        case ID_NEW_INCOMING_CONNECTION:
            {
                srand((unsigned)time(0));
                for (int i = 0; i < 100; i++) {
                    _randNumberList1.push_back(rand());
                    _randNumberList2.push_back(rand());
                }
                RakNet::BitStream bsOut;
                bsOut.Write((RakNet::MessageID)ID_GAME_MESSAGE_HELLO);
                // no sending the std::vector directly
                for (int i = 0; i < 100; i++) {
                    bsOut.Write(_randNumberList1[i]);
                    bsOut.Write(_randNumberList2[i]);
                }
                std::cout << "randomizing seed has been sent.\n";
                _peerServer->Send(
                    &bsOut, HIGH_PRIORITY,
                    RELIABLE_ORDERED, 0,
                    RakNet::UNASSIGNED_SYSTEM_ADDRESS, true);
                return true;
            }
            break;

        default:
```

```
                break;
            }
        }
}
```

```
// Receive Random number list
while (true) {
    for (packet = _peerClient->Receive();
        packet; _peerClient->DeallocatePacket(packet),
        packet = _peerClient->Receive()) {
        switch (packet->data[0])
        {
        case ID_GAME_MESSAGE_HELLO:
            {
                RakNet::BitStream bsIn(packet->data, packet->length, false);
                bsIn.IgnoreBytes(sizeof(RakNet::MessageID));
                // no receiving the std::vector directly
                for (int i = 0, tmp; i < 100; i++) {
                    bsIn.Read(tmp);
                    _randNumberList1.push_back(tmp);
                    bsIn.Read(tmp);
                    _randNumberList2.push_back(tmp);
                }
                std::cout << "randomizing seed has been received.\n";
                return true;
            }
            break;
        }
    }
}
```

And in generating the new blocks function, I implement it like this:

```
// old
srand((unsigned)time(0));
int index = rand() % _blockTemplate.size();
// use index to define the next kind of block
// ……
```

```
// new
int rd = _randNumberList[_currentIndex++];
_currentIndex %= _randNumberList.size();
int index = rd % _blockTemplate.size();
// use index to define the next kind of block
// ……
```

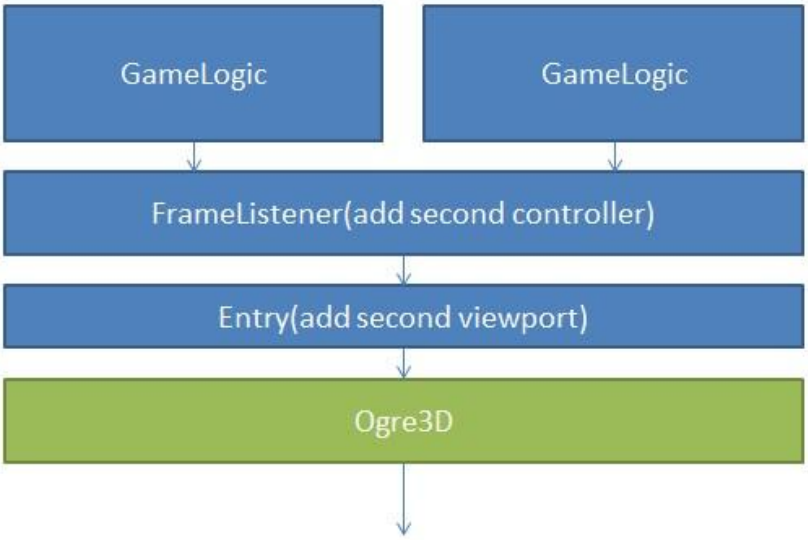
Then, everything is done.

V. Architectural

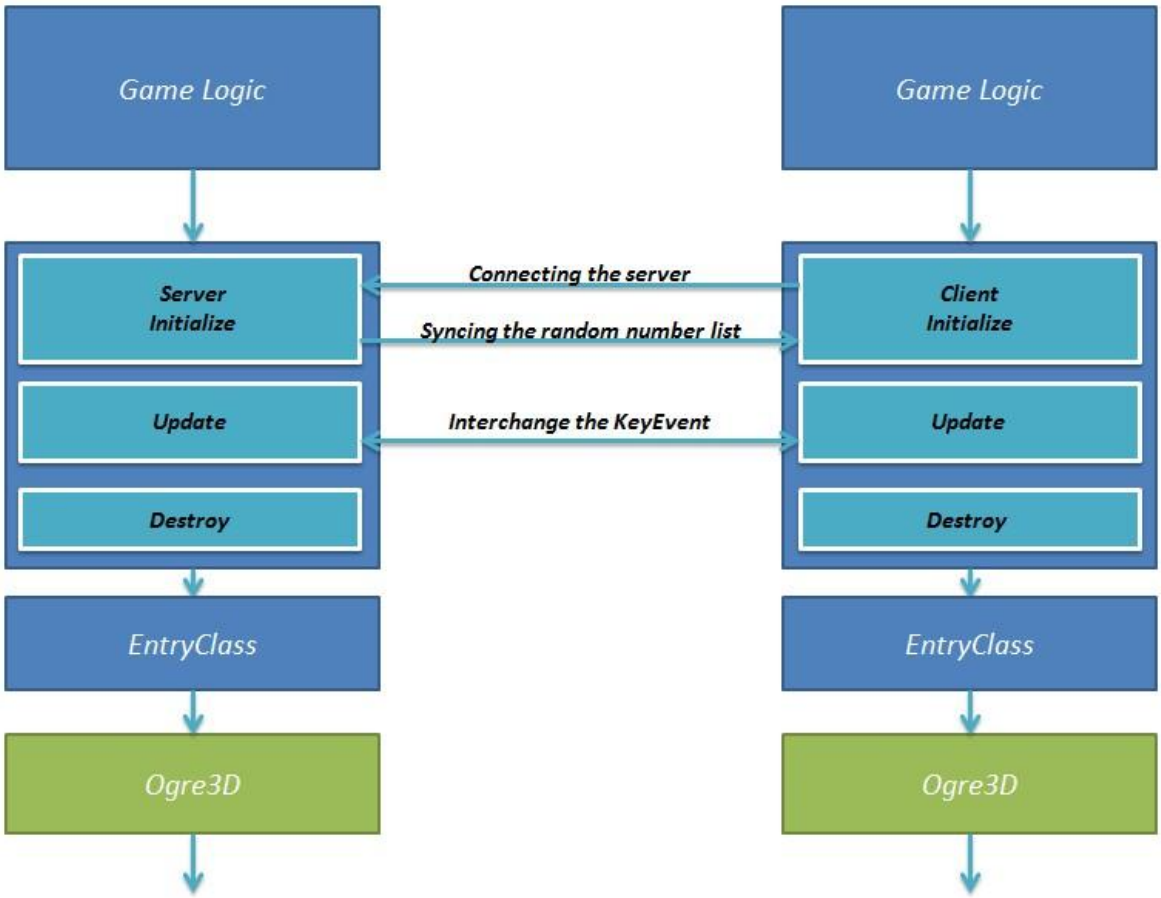
Basing on the last assignment, I did not modify the architecture, but add some features into the control

level.

This is the last assignment's architecture:



And this time, we change it like this:

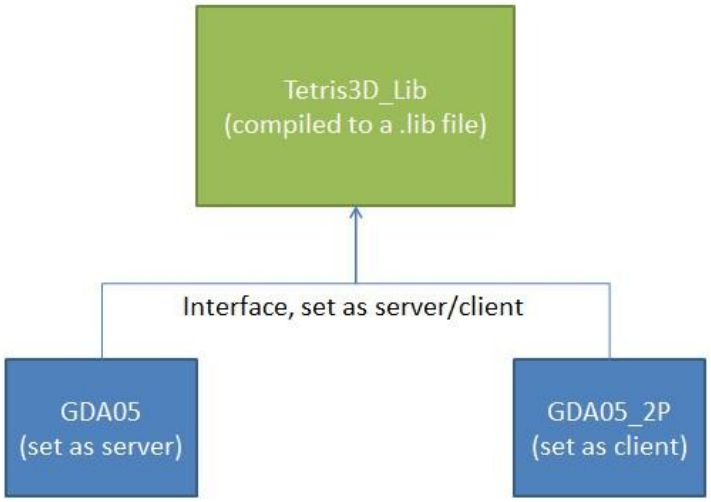


Other parts remain the same.

VI. Hierarchy Diagram

Nothing added, except some networking support functions.

Project Dependency:


Tetris3D_Lib
(compiled to a .lib file)
Interface, set as server/client
GDA05
(set as server)
GDA05_2P
(set as client)