Introduction to Computer Networks – Multi-Player Karting

B05901188 Yen-Shuo, Su

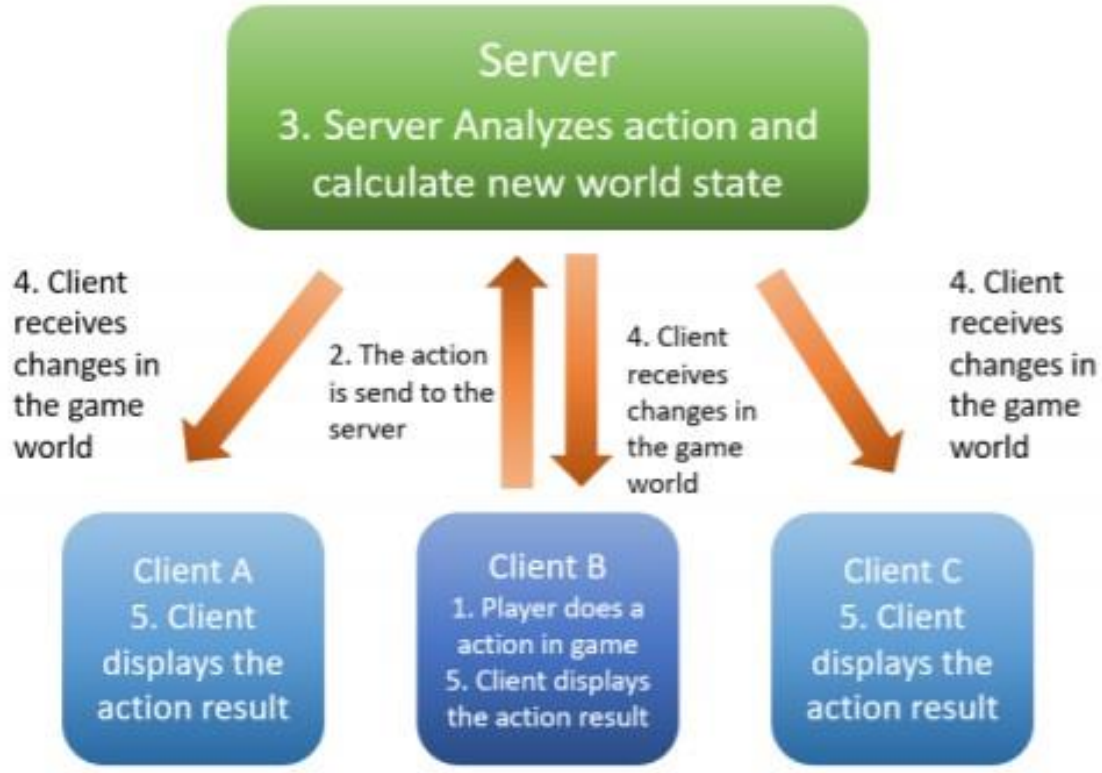
B05201037 Pei-Rong, Li

1. Introduction

The unity Karting Microgame template is used to create local player and other player objects. We used the sample code provided by the teaching assistant to speed up data transfer, improve the movement of game objects, and added some game effects. The maximum players allowed for our multi-player game is ten.

2. Network Model

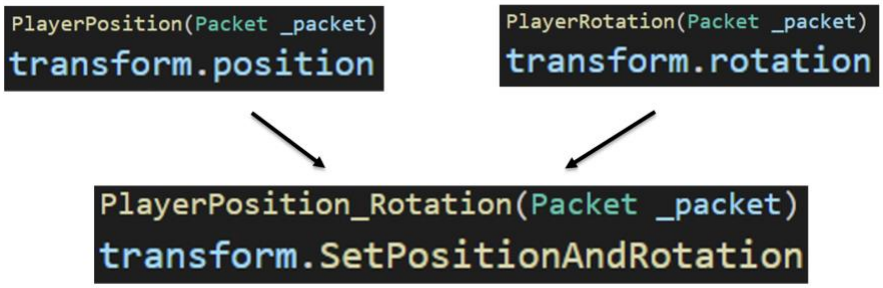
An independent server is used to provide services. When clients receive the inputs from the players, they will provide the information that needs to be updated to the server through packages. The server will then send the information to all the other clients to synchronize the game status of all clients.



3. Quality of Service Improvement

Since the original code suffers from serious lagging when the amount of players increases, various methods are implemented to either speed up the data transferring process or to improve the movement of game objects seen by the client.

3.1 SetPositionAndRotation



In the original sample code, the position and rotation information are sent using two separate methods, thus, two separate packets. However, through the use of the SetPositionAndRotation method, both position and rotation information can be set and sent through one packet at the same time, increasing the speed of data transfer.

3.2 No Packet Transfer Without Client Movement

```
var diff = new_pos-old_pos;
var ang = Quaternion.Angle(old_rot,new_rot);
if (diff.magnitude >= 0.01f || ang !=0f){
    if (_send == 1){
        ClientSend.PlayerMovement(new_pos);
    }
}
```

If the client keeps updating its information through the server even if it is not moving, it will be a waste of resources. Thus, if the client is not moving, it will not transfer packets to the server. At the same time, the server will not update the client's position and rotation status to all the other clients using the UDP protocol. In this way, the number of packets that the server has to deal with decreases. Also, the problem of network congestion can be eased since packets with the same or similar information does not have to be transferred over and over again.

3.3 Object Movement Smoothing through Interpolation

When new position or rotation information of other game objects is received, it will often be the case that the position or rotation status of the game object is fairly far off from its original one. If updated directly, it will cause the position to change abruptly and result in lagging in terms of game experience. To prevent this situation, we propose two different approaches.

Method 1: While Loop Approach

```
var LagDistance = GameManager.players[_id].transform.position - _position;

if(LagDistance.magnitude < 0.1f)
{
    GameManager.players[_id].transform.position = _position;
    LagDistance = Vector3.zero;
}
else
{
    while(LagDistance.magnitude >= 0.1f)
    {
        GameManager.players[_id].transform.position = _position + 0.1f*LagDistance.normalized;
    }
}
```

Using a while loop, the position can be changed through small increments starting from the original position. The advantage is that, game objects will no longer face abrupt changes in movements, since it is now forced to move in small increments. However, if the game object has finished updating its position to the final position before receiving the next packet, it will still face the problem of having to pause for a while before updating its position again.

Method 2: SmoothDamp Method

```
float newPositionX = Mathf.SmoothDamp(GameManager.players[_id].transform.position.x,
                                      _position.x, ref XPositionVelocity, con*Time.deltaTime);
float newPositionY = Mathf.SmoothDamp(GameManager.players[_id].transform.position.y,
                                      _position.y, ref YPositionVelocity, con*Time.deltaTime);
float newPositionZ = Mathf.SmoothDamp(GameManager.players[_id].transform.position.z,
                                      _position.z, ref ZPositionVelocity, con*Time.deltaTime);
GameManager.players[_id].transform.SetPositionAndRotation(new Vector3(newPositionX, newPositionY, newPositionZ),
                                                          _rotation);
```

Another way to smooth object movements is through the use of the SmoothDamp method. Using the method, a time length is specified so that the object moves from its original position to the new one during this time interval. The abrupt change in position problem can also be alleviated by the SmoothDamp method. However, the SmoothDamp method only approaches but never reaches the final position. The looping process has to be stopped somehow before the next packet arrives. Furthermore, objects face inconstant velocity using this method.

3.4 Prevent Different Threads Accessing the Same Game Object

```
if(GameManager.players[_id].moving == 1)
{
    while((GameManager.players[_id].transform.position != _position) && (GameManager.players[_id].moving == 1))
    {
        float newPositionX = Mathf.SmoothDamp(GameManager.players[_id].transform.position.x, _position.x, ref XPositionVelocity, con*Time.deltaTime);
        float newPositionY = Mathf.SmoothDamp(GameManager.players[_id].transform.position.y, _position.y, ref YPositionVelocity, con*Time.deltaTime);
        float newPositionZ = Mathf.SmoothDamp(GameManager.players[_id].transform.position.z, _position.z, ref ZPositionVelocity, con*Time.deltaTime);
        GameManager.players[_id].transform.SetPositionAndRotation(new Vector3(newPositionX, newPositionY, newPositionZ), _rotation);
    }

    GameManager.players[_id].moving -= 1;
}
else
{
    while((GameManager.players[_id].transform.position != _position) && (GameManager.players[_id].moving != 1))
    {
        float newPositionX = Mathf.SmoothDamp(GameManager.players[_id].transform.position.x, _position.x, ref XPositionVelocity, con*Time.deltaTime);
        float newPositionY = Mathf.SmoothDamp(GameManager.players[_id].transform.position.y, _position.y, ref YPositionVelocity, con*Time.deltaTime);
        float newPositionZ = Mathf.SmoothDamp(GameManager.players[_id].transform.position.z, _position.z, ref ZPositionVelocity, con*Time.deltaTime);
        GameManager.players[_id].transform.SetPositionAndRotation(new Vector3(newPositionX, newPositionY, newPositionZ), _rotation);
    }
}
```

A common problem that both methods in the above section faces is that another thread is created upon arrival of the next packet before the previous thread has finished updating the position of the same game object. When multiple threads are changing a game object's position at the same time, the game object will appear to be vibrating back and forth. To fix this problem, the mechanism has to be designed so that if the next thread starts to update an object's position, the previous thread has to be stopped even if it hasn't finished updating the position. In this way, we can solve the back and forth vibration problem of game objects.

3.5 Reduce the Number of Packets as Player Number Increases

```
int cnt = GameManager.players.Count;
if (cnt<=6){
    _send = 1;
}
if (diff.magnitude >= 0.01f || ang !=0f){
    if (_send == 1){
        ClientSend.PlayerMovement(new_pos);
    }
}
if (cnt>6){
    _send*=-1;
}
```

When there are too many players, the server will have to handle a lot of packets, and the lagging problem becomes even more obvious. To solve this problem, the client will cut the number of packets sent by half when the number of players exceeds six. The server will also cut the number of packets sent through UDP by half.

3.6 Other Attempts

```
float packetarrivaltime = Time.time;
float timeinterval = packetarrivaltime - GameManager.players[_id].lastpacketarrivaltime;
GameManager.players[_id].times.Dequeue();
GameManager.players[_id].times.Enqueue(timeinterval);
GameManager.players[_id].lastpacketarrivaltime = packetarrivaltime;
float averagetimeinterval = GameManager.players[_id].times.Average();
```

We also tried to change the data transferred from position and rotation into inputs to reduce the data that needs to be transferred. However, we realized that when the player holds on to an input without letting it go, it is hard to send all inputs through packets, and as the playing time lengthens, the position that a player is actually at and the position calculated by other clients will be more and more off. Thus, this method will probably not be applicable to a racing car game. We also tried to change the time interval in the SmoothDamp method into the average time interval that the client receives packets. However, there is too much calculation involved, especially when the number of players increases, and the program crashes after a running for a while. Thus, we gave up on this approach.

4. Game Effects

We increased the complexity of the racing track and added some game objects along with some effects. Below is a Pokémon that shoots lightning bolts.

5. Group work Division

Yen-Shuo, Su: QoS improvement and game effects

Pei-Rong, Li: QoS improvement

6. References

[1] Scripting API: Mathf.SmoothDamp - Unity

[2] How to use mathf.smoothdamp properly – Unity Answers

[3] Scripting API: Transform.SetPositionAndRotation - Unity