nlGameStateReplicaSeminararbeit

im Studiengang Master Game Engineering und Simulation

Lehrveranstaltung Fortgeschrittene Netzwerktechnologien

Übungsabgabe

Fortgeschrittene Netzwerktechnologien

Ausgeführt von: Kristian Ljubek und David Pertiller  
Personenkennzeichen: 1210585008 und 1210585012

BegutachterIn: Cyrus Preuss und Stefan Schmidt, MSc

Wien, 23.12.2013

Inhaltsverzeichnis

[1 Aufgaben 3](#_Toc367364234)

[1.1 Erweiterung der Gameplay Rules 3](#_Toc367364235)

[1.1.1 add a player login with name and password (simple hash is enough) 3](#_Toc367364236)

[1.1.2 add a persistent 'known player management' to the server 3](#_Toc367364237)

[1.1.3 allow up to 16 players login to the game server simultaneously 3](#_Toc367364238)

[1.1.4 only 4 players can play at one time / the rest become spectators 3](#_Toc367364239)

[1.1.5 once one player dies the next spectator becomes a active player 3](#_Toc367364240)

[1.1.6 the dead player becomes a spectator 3](#_Toc367364241)

[1.1.7 add a 'kill' count for each player 3](#_Toc367364242)

[1.1.8 add the 'kill-count' and the name of the active (not spectating) players to the client UI 3](#_Toc367364243)

[1.2 Replication 3](#_Toc367364244)

[1.2.1 use compressed datagrams for transmitted state structures 3](#_Toc367364245)

[1.2.2 do not continuously send projectile updates 3](#_Toc367364246)

[1.2.3 implement client side interpolation 4](#_Toc367364247)

[1.2.4 implement client side prediction for the controlling client 4](#_Toc367364248)

[1.2.5 check if it is possible to implement server side lag compensation 4](#_Toc367364249)

[1.2.6 it might make sense to invent one additional replicated object called 'GameState' 4](#_Toc367364250)

[1.3 Additional functionality (bonus points) 4](#_Toc367364251)

[1.3.1 notify spectatores about how many rounds they have to wait 4](#_Toc367364252)

[1.3.2 add chat functionality to the game 4](#_Toc367364253)

# 

# Aufgaben

## Erweiterung der Gameplay Rules

### add a player login with name and password (simple hash is enough)

UI-Controls als Properties in TankVsTankPluginContent definieren:

class TankVsTankPluginContent : public PluginLayerContent {

**CC\_SYNTHESIZE(CCControlEditBox\*, \_ctrlName, CtrlName)**

**CC\_SYNTHESIZE(CCControlEditBox\*, \_ctrlPassword, CtrlPassword)**

**CC\_SYNTHESIZE(CCControlEditBox\*, \_ctrlMessage, CtrlMessage)**

}

Die Login-UI in addGameContentUI hinzufügen:

void TankVsTankPluginContent::addGameContentUI( SLSize idx, PeerNode\* peerNode, TankVsTankGameplayLayer\* parentLayer )

{

**if(peerNode->getPeer()->getTopology() == CLIENT)**

**{**

**\_ctrlName = ControlUtils::createEditBox("Name:", ctrlsPreferredSize);**

**ctrls->addObject(\_ctrlName);**

**\_ctrlPassword = ControlUtils::createEditBox("Password:", ctrlsPreferredSize);**

**ctrls->addObject(\_ctrlPassword);**

**\_ctrlMessage = nl::ControlUtils::createEditBox("Message:", ctrlsPreferredSize);**

**ctrls->addObject(\_ctrlMessage);**

**}**

}

### add a persistent 'known player management' to the server

where player properties are:

name

password

killcount

numberOfGamesPlayed

hint: PersistentDictionary

### allow up to 16 players login to the game server simultaneously

Um die Verwaltung der eingeloggten User sauber zu implementieren, wurde hierzu eine neue Klasse nlGameStateReplicaComponent erstellt, welche den „GameState“ repliziert und hierfür diverse Logiken wie die Beschränkung der Verbindungen implementiert. Für die Reglementierung der eingeloggten User wurde eine statische Variable hinzugefügt, die das Maximum der erlaubten Verbindungen festlegt:

static size\_t CONNECTION\_LIMIT = 16;

Des Weiteren wurde ein Property in dieser Klasse definiert, mit welchem die Anzahl der aktiven Verbindungen verwaltet wird:

SL\_SYNTHESIZE(int,\_connectionCount,ActiveConnectionCount);

In nlGameStateReplicaComponent.cpp wird anschließend im Event onNewIncommingConnectionNotification geprüft, ob das Limit bereits überschritten wurde. Ist dies der Fall, wird die Verbindung über RakNet geschlossen. Andernfalls wird die aktuelle Anzahl der Verbindungen erhöht und die GUID der Verbindung in einer Liste für mögliche spätere Bedarfe gespeichert:

void nlGameStateReplicaComponent::onNewIncommingConnectionNotification(CCObject\* peerWrapperObject)

{

PeerWrapper\* peerWrapper(dynamic\_cast<PeerWrapper\*>(peerWrapperObject));

if(peerWrapper != nullptr)

{

if(getReplica()->getPeer() == peerWrapper->getPeer())

{

if(getActiveConnectionCount() >= CONNECTION\_LIMIT)

{

getReplica()->getPeer()->log(ELogType\_Info, "Connection Limit reached, closing connection");

getReplica()->getPeer()->accessRakNetPeer()->CloseConnection(peerWrapper->getGUID(), true);

}

else

{

\_connectionList.push\_back(peerWrapper->getGUID()); //we maintain this connection ID in a list

\_connectionCount++; //increase the active connection count

}

}

}

}

Um die Anzahl der Verbindungen korrekt zu halten darf nicht nur auf eingehende Verbindungen reagiert, sondern muss auch auf verlorene sowie getrennte Verbindungen reagiert werden. Dazu wird in den Events onConnectionLostNotification und onConnectionDisconnectedNotification Rechnung getragen und die Anzahl der Verbindungen dekrementiert sowie die GUID gelöscht:

void nlGameStateReplicaComponent::onConnectionLostNotification(CCObject\* peerWrapperObject)

{

PeerWrapper\* peerWrapper(dynamic\_cast<PeerWrapper\*>(peerWrapperObject));

if(peerWrapper != nullptr)

{

removeConnection(peerWrapper);

}

}

void nlGameStateReplicaComponent::onConnectionDisconnectedNotification(CCObject\* peerWrapperObject)

{

PeerWrapper\* peerWrapper(dynamic\_cast<PeerWrapper\*>(peerWrapperObject));

if(peerWrapper != nullptr)

{

removeConnection(peerWrapper);

}

}

void nlGameStateReplicaComponent::removeConnection(PeerWrapper\* peerWrapper)

{

if(getReplica()->getPeer() == peerWrapper->getPeer())

{

for(int i = 0; i < getActiveConnectionCount(); ++i)

{

if(\_connectionList[i] == peerWrapper->getGUID())

{

\_connectionList.erase(\_connectionList.begin()+i);

\_connectionCount--;

break;

}

}

}

}

Um neue Verbindungen sowie andere Events allerdings überhaupt erst mitbekommen zu können, wurden hierfür Notifications für die entsprechenden Events definiert und registriert:

class nlGameStateReplicaComponent : public ReplicaComponent

{

private:

Notification \_notificationConnectionLost;

Notification \_notificationConnectionDisconnected;

Notification \_notificationNewIncommingConnection;

};

nlGameStateReplicaComponent::nlGameStateReplicaComponent() : \_connectionCount(0)

,\_notificationConnectionLost(SL\_NOTIFY\_CONNECTION\_LOST)

,\_notificationConnectionDisconnected(SL\_NOTIFY\_DISCONNECTION)

,\_notificationNewIncommingConnection(SL\_NOTIFY\_NEW\_INCOMMING\_CONNECTION)

{

\_replica.setName(nlGameStateReplicaComponent::staticClassName());

ServerAuthorityReplicationRule\* replicationRule(ServerAuthorityReplicationRule::create());

replicationRule->\_replica = getReplica();

\_replica.setReplicationRule(replicationRule);

\_notificationConnectionLost.addObserver(this, callfuncO\_selector(nlGameStateReplicaComponent::onConnectionLostNotification));

\_notificationConnectionDisconnected.addObserver(this, callfuncO\_selector(nlGameStateReplicaComponent::onConnectionDisconnectedNotification));

\_notificationNewIncommingConnection.addObserver(this, callfuncO\_selector(nlGameStateReplicaComponent::onNewIncommingConnectionNotification));

}

nlGameStateReplicaComponent::~nlGameStateReplicaComponent()

{

//unregister observed notifications

\_notificationConnectionLost.removeObserver();

\_notificationConnectionDisconnected.removeObserver();

\_notificationNewIncommingConnection.removeObserver();

}

Damit die GameStateReplica ihre Arbeit erfüllen kann, wird diese in nlTankVsTankGameLogicNode.cpp bei onPeerIsConnected erstellt:

void TankVsTankGameLogicNode::onPeerIsConnected(PeerWrapper\* peerWrapper)

{

getPeer()->log(ELogType\_Info, "%s - received peer is connected", getClassName());

// creating gameStateReplica

getPeer()->getReplicaManager()->createReplica(nlGameStateReplicaComponent::staticClassName(),nullptr);

}

### only 4 players can play at one time / the rest become spectators

Handling über zusätzliches Property “isSpectatorMode” in nlTankPlayerReplicaComponent und Logik in NlGameStateReplicaComponent:

class TankPlayerReplicaComponent : public PlayerReplicaComponent

{

public:

SL\_SYNTHESIZE\_IS(bool,\_isSpectator,SpectatorMode,SpectatorMode);

};

### once one player dies the next spectator becomes a active player

### the dead player becomes a spectator

### add a 'kill' count for each player

### add the 'kill-count' and the name of the active (not spectating) players to the client UI

## Replication

### use compressed datagrams for transmitted state structures

Das Padding durch Anordnung verschiedener Datentypen in Structs sollte bedacht werden. Unterschiede ergeben sich durch Anordnung von Variablen in Strukturen oder aber auch durch das Zielsystem und den Compiler (sowie dessen Optimierungseinstellungen).

Anpassung in nlProtocolStructures.h:

ControllerValues: links/rechts, vorwärts/rückwärts sowie shoot können mit einem char anstatt einem float abgebildet werden (Einsparung: 3\*3 Bytes = 9 Bytes):

|  |  |
| --- | --- |
| Original: | Komprimiert: |
| typedef struct TControllerValues  {  float \_leftRight;  float \_forwardBackward;  float \_shoot;  RakNet::NetworkID \_controlledReplicaNetworkId;  SLSize \_updateTick;  } ControllerValues; | typedef struct TCompressed\_ControllerValues  {  char \_leftRight;  char \_forwardBackward;  char \_shoot;  RakNet::NetworkID \_controlledReplicaNetworkId;  SLSize \_updateTick;  } TCompressed\_ControllerValues; |

Anpassung in Dynamic2DActorDatagram:

Der Datentyp für x und y wurde von float auf short angepasst (Einsparung: 2\*2 Bytes = 4 Bytes):

|  |  |
| --- | --- |
| Original: | Komprimiert: |
| typedef struct TDynamic2DActorDatagram  {  float \_x;  float \_y;  float \_fx;  float \_fy;  float \_lvx;  float \_lvy;  float \_avz;  } Dynamic2DActorDatagram; | typedef struct TCompressed\_Dynamic2DActorDatagram  {  short \_x;  short \_y;  float \_lvx;  float \_lvy;  float \_fx;  float \_fy;  float \_avz;  } TCompressed\_Dynamic2DActorDatagram; |

Dann noch in der BitStream.h die Compressed-Methoden für die jeweiligen Datentypen (eignet sich z.B. besonders gut für Quaternions) verwenden:

Anstatt:

bitStream.WriteAlignedBytes((const unsigned char \*)&\_ctrlValues, sizeof(ControllerValues));

Verwendung der Komprimierung:

Compressed\_ControllerValues comValues;

comValues.\_forwardBackward = TCompressedFixpoint<float,char,8>::writeCompress(\_ctrlValues.\_forwardBackward , -1.0f, 1.0f );

comValues.\_leftRight = TCompressedFixpoint<float,char,8>::writeCompress(\_ctrlValues.\_leftRight , -1.0f, 1.0f );

comValues.\_shoot = TCompressedFixpoint<float,char,8>::writeCompress(\_ctrlValues.\_shoot , -1.0f, 1.0f );

comValues.\_updateTick = \_ctrlValues.\_updateTick;

comValues.\_controlledReplicaNetworkId = \_ctrlValues.\_controlledReplicaNetworkId;

bitStream.WriteAlignedBytes( (const unsigned char \*)&comValues, sizeof(Compressed\_ControllerValues));

### do not continuously send projectile updates

a projectile spawns with an initial position/orientation/velocity and lifetime

(that's it on the clientside)

### implement client side interpolation

### implement client side prediction for the controlling client

### check if it is possible to implement server side lag compensation

One way of compensating server side lag could be to **keep track of the game state on the client** itself too and send its absolute state to the server. The **drawbacks** of this method are that a) the current game state maintenance is not in the single responsibility of the server anymore b) the game state handling needs to be remodeled and b) this method is prone to allow cheating, as clients can send whatever they like (e.g. new positions that vary greater than the maximum speed allows them to move). Therefore, in our opinion, this lag compensation is **impracticable**.

Another way - one that comes without the need of handling the game state at the client - could be **to buffer the game states on the server** and calculate the resulting state based on the game state the client had at the time the action was performed (= game state of (corrent time minus the lag time)). This has the benefit that the **client’s actions correspond to what the player himself sees**, e.g. firing a bullet on the enemy and hitting the enemy leads to the death of the enemy. The drawback of this approach is that within the time of the lag of the client the **enemy might have already moved**, but the player hasn’t received that state yet. Therefore, the enemy would die because he was hit from the client’s perspective and calculations, although actually the enemy could have already moved to another position which is safe (worst case from the dying player’s perspective: he has moved behind a wall or a shield and “out of nowhere” he dies due to the calculations with the old position at the other client). In our opinion, the drawback of this approach clears the benefit, which makes this implementation not lucrative enough.

Considering these techniques, server-side lag compensation is definitely possible, but it doesn’t come without drawbacks, especially if you’re not carefully enough investigating this topic and make it a dedicated project with just the focus on accomplishing an acceptable lag compensation. In our opinion the benefits of an imperfect server-side lag compensation cannot outperform the drawbacks that arise from it, which is why we don’t interpolate or extrapolate or buffer game states and leave server-side lag “as-is”.

### it might make sense to invent one additional replicated object called 'GameState'

Ja es macht durchaus Sinn eine replizierte Klasse GameState zu erstellen, zum Beispiel zur Verwaltung des Spectator-Modus. Diese Klasse wurde bereits im Punkt 1.1.3 sowie bei den jeweiligen Punkten in denen der GameState genutzt wird, beschrieben.

## Additional functionality (bonus points)

### notify spectatores about how many rounds they have to wait

### add chat functionality to the game