Seminararbeit

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](#_Toc375527793)

[1.1 Erweiterung der Gameplay Rules 3](#_Toc375527794)

[1.1.1 add a player login with name and password (simple hash is enough) 3](#_Toc375527795)

[1.1.2 add a persistent 'known player management' to the server 3](#_Toc375527796)

[1.1.3 allow up to 16 players login to the game server simultaneously 3](#_Toc375527797)

[1.1.4 only 4 players can play at one time / the rest become spectators 7](#_Toc375527798)

[1.1.5 once one player dies the next spectator becomes an active player 9](#_Toc375527799)

[1.1.6 the dead player becomes a spectator 11](#_Toc375527800)

[1.1.7 add a 'kill' count for each player 12](#_Toc375527801)

[1.1.8 add the 'kill-count' and the name of the active (not spectating) players to the client UI 14](#_Toc375527802)

[1.2 Replication 15](#_Toc375527803)

[1.2.1 use compressed datagrams for transmitted state structures 15](#_Toc375527804)

[1.2.2 do not continuously send projectile updates 17](#_Toc375527805)

[1.2.3 implement client side interpolation 17](#_Toc375527806)

[1.2.4 implement client side prediction for the controlling client 17](#_Toc375527807)

[1.2.5 check if it is possible to implement server side lag compensation 17](#_Toc375527808)

[1.2.6 it might make sense to invent one additional replicated object called 'GameState' 18](#_Toc375527809)

[1.3 Additional functionality (bonus points) 18](#_Toc375527810)

[1.3.1 notify spectatores about how many rounds they have to wait 18](#_Toc375527811)

[1.3.2 add chat functionality to the game 18](#_Toc375527812)

# 

# Aufgaben

## Erweiterung der Gameplay Rules

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

TODO:

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

TODO:

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 eventuelle 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 und die Verbindung aus der Liste 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 in nlTankVsTankGameLogicNode.cpp bei onPeerIsConnected über den ReplicaManager createReplica aufgerufen:

void TankVsTankGameLogicNode::onPeerIsConnected(PeerWrapper\* peerWrapper)

{

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

// creating gameStateReplica

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

}

Wonach sie in nlGameContentReplicaManager schließlich erstellt wird:

if(typeName == nlGameStateReplicaComponent::staticClassName())

{

replicaComponent = nlGameStateReplicaComponent::create();

replica = replicaComponent->getReplica();

}

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

Die Kennzeichnung, ob sich ein Spieler im Spectator-Modus befindet, erfolgt über ein zusätzliches Property “isSpectatorMode” in nlTankPlayerReplicaComponent:

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

Bei der Konstruktion der TankPlayerReplica wird der Spectator-Modus zunächst mit false initialisiert:

TankPlayerReplicaComponent::TankPlayerReplicaComponent() :\_isSpectator(false) { … }

Weiters wurde in nlTankVsTankGameLogicNode.h eine statische Variable mit dem Spieler-Limit von 4 definiert:

static size\_t PLAYER\_LIMIT = 4;

In der Update-Methode in TankVsTankGameLogicNode wird standardmäßig bereits über alle Children des GameplayLayers iteriert, um zerstörte Aktoren ausfindig zu machen. Nun wurde die Update-Methode dahingehend erweitert, dass auch gleich der Spectator-Status der TankReplicas abgefragt wird. Dabei wird mitgezählt, wieviele TankReplicas sich nicht im Spectator-Modus befinden (also aktiv sind). Ab dem 4. aktiven Spieler (Limit spezifiziert durch PLAYER\_LIMIT) werden alle weiteren Spieler in den Spectator-Modus versetzt:

void TankVsTankGameLogicNode::update( float dt )

{

SLBaseClass::update(dt);

SLTimeInterval accumTime(getAccumTime());

GameplayLayer\* gameplayLayer(getGameplayLayer());

if(gameplayLayer != nullptr)

{

Peer\* peer(getPeer());

// iterate over all children and check if Actors are destroyed

size\_t activePlayers = 0; //count how many active players we have

CCArray\* destroyedChildren(CCArray::create());

CCObject\* child = nullptr;

CCARRAY\_FOREACH(gameplayLayer->getChildren(), child)

{

GameActorNode\* actorNode(dynamic\_cast<GameActorNode\*>(child));

if(actorNode != nullptr)

{

//find the TankPlayerReplica component by iterating over all components of the GameActor until we find it

ComponentArray\* components(actorNode->getComponents());

SLSize idx(0);

IComponent\* component(components->componentAt(idx));

while(component != nullptr)

{

TankPlayerReplicaComponent\* replicaComponent(dynamic\_cast<TankPlayerReplicaComponent\*>(component));

if(replicaComponent != nullptr)

{

if (replicaComponent->isSpectatorMode() == false)

++activePlayers; //count the player as active player

if (activePlayers > PLAYER\_LIMIT)

replicaComponent->setSpectatorMode(true); //if we already have <4> active players, set the player to spectator mode

break;

}

++idx;

component = components->componentAt(idx);

}

if(actorNode->isDestroyed()) {

destroyedChildren->addObject(actorNode);

}

}

}

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

Für diese Aufgabe wurde in TankVsTankGameLogicNode eine Liste (präziser: CCArray) eingefügt, welche sich die im Spectator-Mode befindlichen Spieler merkt:

class TankVsTankGameLogicNode : public PeerObserverNode

{

protected:

CCArray \*\_spectators; //this list provides the order of which spectator becomes active next

}

Initialisiert wird diese Liste in der init-Methode in TankVsTankGameLogicNode:

bool TankVsTankGameLogicNode::init()

{

bool initialized(SLBaseClass::init());

\_spectators = new CCArray(); //initialize empty spectators list

return initialized;

}

TankVsTankGameLogicNode::~TankVsTankGameLogicNode()

{

delete \_spectators;

}

Die eigentliche Behandlung der Spectators wurde nun in der update-Methode ergänzt. Dazu werden alle Spieler die sich im Spectator-Modus befinden, in die Spectator-Liste eingefügt sofern sie darin nicht schon vorgemerkt sind. Jene Spieler, die sich bereits am längsten im Spectator-Modus befinden, finden sich an den vorderen Stellen der Liste, genauer: der am längsten wartende ist immer auf Index 0, der am spätestens eingefügte Spieler der gerade erst zum Spectator wurde, befindet sich an letzter Stelle. Werden Plätze frei, so werden jene Spieler, die am längsten warten, aus dem Spectator-Modus genommen und als aktive Spieler gesetzt. Anschließend werden diese Sieler aus der Spectator-Liste gelöscht, wodurch die nächsten wartenden Spectators an die vordere Stelle der Liste rutschen:

CCARRAY\_FOREACH(gameplayLayer->getChildren(), child)

{

GameActorNode\* actorNode(dynamic\_cast<GameActorNode\*>(child));

if(actorNode != nullptr)

{

if(actorNode->isDestroyed())

{

destroyedChildren->addObject(actorNode);

}

else

{

// find the TankPlayerReplica component by iterating over all components of the GameActor until we find it

TankPlayerReplicaComponent\* replicaComponent = getTankPlayerReplicaComponentFromActorNode(actorNode);

if(replicaComponent != nullptr)

{

//if we already have e.g. 4 active players, set the further players to spectator mode

if (activePlayers >= PLAYER\_LIMIT)

replicaComponent->setSpectatorMode(true);

if (replicaComponent->isSpectatorMode() == false)

++activePlayers; //if the player isn't in spectator mode, it's an active player

else

{

//if the player is not playing, add him to the spectator list

if (\_spectators->indexOfObject(replicaComponent) == UINT\_MAX) //check if the spectator isn't already in the list

\_spectators->addObject(replicaComponent); //add the spectator to the list so we can disable spectator mode for it when an active player gets shot

}

}

}

}

}

//calculate how many players can be activated from spectator mode

int activatePlayers = (int)PLAYER\_LIMIT - activePlayers;

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

{

//if a spectator can become active, check if there are enough spectators on the list and if so, active them (disable spectator mode)

if (i < \_spectators->count())

{

TankPlayerReplicaComponent\* playerReplica(dynamic\_cast<TankPlayerReplicaComponent\*>(\_spectators->objectAtIndex(i)));

playerReplica->setSpectatorMode(false); //the player is now active and is able to play

getPeer()->log(ELogType\_Info, "Player with index %i became active", playerReplica->idx());

}

}

//update the spectator list and remove all players that got activated

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

{

if (i < \_spectators->count())

{

//for every player that got activated, we delete it from the list

\_spectators->removeObjectAtIndex(0); //the player that gets activated first is at index 0

}

}

Um einen einfachen Zugriff auf den Spectator-Modus der TankPlayerReplica zu ermöglichen, wurde diese Funktion erstellt, die zu einer ActorNode die TankPlayerReplicaComponent zurückgibt:

TankPlayerReplicaComponent\* TankVsTankGameLogicNode::getTankPlayerReplicaComponentFromActorNode(ActorNode \*actorNode)

{

// find the TankPlayerReplica component by iterating over all components of the GameActor until we find it

ComponentArray\* components(actorNode->getComponents());

SLSize idx(0);

IComponent\* component(components->componentAt(idx));

while(component != nullptr)

{

TankPlayerReplicaComponent\* replicaComponent(dynamic\_cast<TankPlayerReplicaComponent\*>(component));

if(replicaComponent != nullptr)

{

return replicaComponent;

}

++idx;

component = components->componentAt(idx);

}

}

### the dead player becomes a spectator

In TankPlayerReplicaComponent::postUpdate wird zu Beginn geprüft, ob der Actor zerstört wurde und falls ja, wird das Spectator-Flag gesetzt um den Spectator-Modus für diesen Spieler kennzuzeichnen:

void TankPlayerReplicaComponent::postUpdate( float delta )

{

if(isActorNodeDestroyed())

{

setSpectatorMode(true); //the dead player becomes a spectator

}

…

}

Damit der Spieler nicht wieder respawned, wurde auch in der postUpdate Methode von LocalPlayerReplicaComponent auf den Fall, dass der Actor zerstört wurde, eingegangen:

void LocalPlayerReplicaComponent::postUpdate( float delta )

{

// postUpdate will always be called once per frame

if(isActorNodeDestroyed()) {

}

else

{

if (getTopology()==CLIENT) {

}

else if (getTopology()==SERVER)

{

//check if the player has been destroyed

TankPlayerReplicaComponent\* tankPlayerReplicaComponent(getTankPlayerReplicaComponent());

TankReplicaComponent\* tankReplicaComponent(tankPlayerReplicaComponent->getTankReplicaComponent());

GameActorNode\* gameActorNode(tankPlayerReplicaComponent->getTankActorNode());

if(tankReplicaComponent != nullptr) {

if(gameActorNode != nullptr) {

if(gameActorNode->isDestroyed()) //the gameActorNode is destroyed

{

if(tankPlayerReplicaComponent->getActorNode()->isDestroyed() == false) //the playerReplica's ActorNode is not destroyed

{

tankPlayerReplicaComponent->setTankActorNode(nullptr);

tankPlayerReplicaComponent->setTankReplicaComponent(nullptr);

//Player was destroyed -> enable spectator mode

tankPlayerReplicaComponent->setSpectatorMode(true);

SL\_PROCESS\_APP()->log(ELogType\_Info,"Player destroyed & spectator mode enabled");

}

\_ctrlValues.\_controlledReplicaNetworkId = UNASSIGNED\_NETWORK\_ID;

}

}

}

}

}

SLBaseClass::postUpdate(delta);

}

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

Für eine einfache und bequeme Replizierung des Kill-Counts wurde dieser als integraler Bestandteil des Dynamic2DActorDatagram eingefügt (Datentyp: unsigned short, wobei man auch mit einem char auskommen würde. In Bezug auf das Padding wird in der komprimierten Struktur jedoch mit einem short der Platz besser genutzt):

typedef struct TDynamic2DActorDatagram {

float \_x;

float \_y;

float \_fx;

float \_fy;

float \_lvx;

float \_lvy;

float \_avz;

**unsigned short \_killCount;**

SLSize \_updateTick;

} Dynamic2DActorDatagram;

typedef struct TCompressedDynamic2DActorDatagram {

float \_x;

float \_y;

char \_fx;

char \_fy;

unsigned short \_killCount;

float \_lvx;

float \_lvy;

float \_avz;

SLSize \_updateTick;

} Compressed\_Dynamic2DActorDatagram;

Zudem wurde die Basisklasse DynamicActorReplicaComponent um eine Variable \_killCount erweitert, welche als eigentliche Zählvariable für den Kill-Count dient und im Datagram setzt:

nlDynamicActorReplicaComponent.h:

class DynamicActorReplicaComponent : public ReplicaComponent {

protected:

unsigned short \_killCount;

. . .

nlDynamicActorReplicaComponent.cpp:

//initialize kill-count with 0

DynamicActorReplicaComponent::DynamicActorReplicaComponent() : \_killCount(0) { ... }

void DynamicActorReplicaComponent::postUpdate( float delta )

{

// server / authority code

if(getTopology() == SERVER)

{

\_actorDatagram.\_killCount = \_killCount; //set the current kill-count

. . .

Damit der Kill-Count von der Spiel-Logik gesetzt werden kann, wurden in der konkreten Klasse TankReplicaComponent zwei Methoden zur Erhöhung des Kill-Counts implementiert (mehr dazu im nächsten Punkt):

nlTankReplicaComponent.h

class TankReplicaComponent : public DynamicActorReplicaComponent {

public:

void increaseKillCount();

. . .

protected:

void setKillCount(int newKillCount);

. . .

Die Erkennung eines Kills wurde in der update-Methode in TankVsTankGameLogicNode eingebaut. Dazu wurde der bestehende Code zur Kollisionserkennung eines Projektils mit einem (anderen) GameActor dahingehend erweitert, dass über den Instigator des Projektils auf die TankReplica zugegriffen wird bei dieser die Methode zur Erhöhung des Kill-Counts aufgerufen wird:

nlTankVsTankGameLogicNode.cpp (update)

if(distance.length() < (\*tank)->radius()) {

GameActor\* tankActor((GameActor\*)(\*tank)->getUserobject());

GameActor\* projectileActor((GameActor\*)(\*projectile)->getUserobject());

// check the instigator to prevent killing yourself

if(projectileActor->getInstigator() != tankActor)

{

//now we want to find the tank who shot the projectile to reward it

ActorSprite\* instigatorActorSprite(dynamic\_cast<ActorSprite\*>(projectileActor->getInstigator()));

if(instigatorActorSprite != nullptr)

{

//we need the GameActorNode so we can access the TankReplicaComponent which is needed to increase the kill-count

GameActorNode \* gameActorNode = dynamic\_cast<GameActorNode\*>(instigatorActorSprite->getGameActorNode());

if (gameActorNode != nullptr)

{

//get the TankReplica as component of the GameActorNode

TankReplicaComponent \*tankReplica = getComponentFromActorNode<TankReplicaComponent>(gameActorNode);

//I've written a template method to comfortably request a component from a GameActorNdoe

if (tankReplica != nullptr)

{

getPeer()->log(ELogType\_Info, "Tank %i has hit an enemy!", tankReplica->idx());

tankReplica->increaseKillCount(); //reward the player by increasing the kill-count

}

}

}

...

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

Die Darstellung des Namens und der Kill-Count erfolgt in der Klasse TankReplicaComponent. Dazu wurde ein Label vom Typ CCLabelTTF deklariert, welches den in der Basisklasse definierten Kill-Count darstellt:

class TankReplicaComponent : public DynamicActorReplicaComponent {

protected:

CCLabelTTF\* \_labelKillCount;

In nlTankReplicaComponent.cpp wurde dazu wie folgt vorgegangen:

1. Das Label und der Zähler werden mit null bzw. 0 initialisiert:

TankReplicaComponent::TankReplicaComponent() : \_labelKillCount(nullptr)

{..}

1. In der preUpdate-Methode wird geprüft, ob das Label noch nicht initialisiert ist. Falls es erst initialisiert werden muss, wird beim Erstellen des Labels darauf geachtet, dass sich die Information mit dem Actor bewegt. Hierzu wird es dem ActorSprite als Kind hinzugefügt. Des Weiteren wird die Beschriftung unterhalb des Actors platziert und zur besseren Lesbarkeit immer so rotiert, dass sie horizontal ausgerichtet ist:
2. Über die public-Methode increaseKillCount kann der Kill-Count erhöht werden. Diese Methode verwendet die geschützte Methode setKillCount, um die Kill-Count Variable zu erhöhen. In dieser Methode könnten in der detaillierteren Implementierung dann auch etwaige Überprüfungen/Benachrichtigungen bezüglich eines erreichten Kill-Counts eingefügt werden etc.:

void TankReplicaComponent::increaseKillCount()

{

setKillCount(\_killCount + 1);

}

void TankReplicaComponent::setKillCount(int newKillCount)

{

\_killCount = newKillCount;

}

void TankReplicaComponent::preUpdate( float delta )

{

ActorSprite\* actorSprite(getActorSprite());

if(actorSprite != nullptr)

{

if (\_labelInfo == nullptr) //if the kill-count label hasn't been created yet, create it

{

\_labelInfo = dynamic\_cast<CCLabelTTF\*>( ControlUtils::createLabel("New Tank", kCCTextAlignmentLeft) );

\_labelInfo->setAnchorPoint(ccp(0.5f, 0.5f));

CCPoint labelPosition(0, -10); //we don't want the label to directly stick on the tank

\_labelInfo->setPosition(labelPosition);

\_labelInfo->setVisible(true);

setKillCount(0);

actorSprite->addChild(\_labelInfo); //add the label as Child of the actor

}

//Update the kill-count label to displyy the current kill-count

//maintain readability of the kill-count label by keeping it's direction horizontal when rotating the tank

float rotateHorizontally = 360 - actorSprite->getRotation();

\_labelInfo->setRotation(rotateHorizontally);

\_labelInfo->setString(CCString::createWithFormat("Tank: %i, Kills: %i", this->getName(), \_killCount)

->getCString());

## Replication

### use compressed datagrams for transmitted state structures

Oft reicht es, einen kleineren Datentyp für Variablen zu verwenden, bzw. können große Werte mit Verlust von etwas Präzision in kleinere Datentypen komprimiert werden. Des Weiteren muss bei Datenstrukturen das Padding bedacht werden und man kann durch sinnvolle Anordnung verschiedener Datentypen in Structs relativ einfach Platz einsparen.

Anpassungen 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 (Größe mit Padding 32 Bytes): | Komprimiert (Größe mit Padding 16 Bytes): |
| 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; |

Dynamic2DActorDatagram: Der Datentyp für \_fx und \_fy wurde von float auf char angepasst (Einsparung: 2\*3 Bytes = 6 Bytes). Die Anordnung des Kill-Counts wude so gewählt, dass es mit dem Padding aligned ist und nichts verloren geht:

|  |  |
| --- | --- |
| Original (Größe mit Padding 36 Bytes): | Komprimiert (Größe mit Padding 28 Bytes): |
| typedef struct TDynamic2DActorDatagram {  float \_x;  float \_y;  float \_fx;  float \_fy;  float \_lvx;  float \_lvy;  float \_avz;  unsigned short \_killCount;  SLSize \_updateTick;  } Dynamic2DActorDatagram; | typedef struct TCompressedDynamic2DActorDatagram {  float \_x;  float \_y;  char \_fx;  char \_fy;  unsigned short \_killCount;  float \_lvx;  float \_lvy;  float \_avz;  SLSize \_updateTick;  } Compressed\_Dynamic2DActorDatagram; |

Um die Daten von einem größeren Datentyp in einen kleineren zu komprimieren wurde die Methode TCompressedFixpoint aus der Header-Datei "slCompressed.h" verwendet. Am Beispiel PlayerReplicaComponent soll die Verwendung der Komprimierung für die ControllerValues beim Serialisieren und Deserialisieren der Daten gezeigt werden:

Serialisieren (PlayerReplicaComponent::serialize):

* 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));

Deserialisieren (PlayerReplicaComponent::deserialize):

* Anstatt:

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

* Verwendung der De-Komprimierung:

Compressed\_ControllerValues comValues;

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

\_ctrlValues.\_forwardBackward =TCompressedFixpoint<float,char,8>::readInflate

(comValues.\_forwardBackward , -1.0f, 1.0f );

\_ctrlValues.\_leftRight =TCompressedFixpoint<float,char,8>::readInflate

(comValues.\_leftRight , -1.0f, 1.0f );

\_ctrlValues.\_shoot =TCompressedFixpoint<float,char,8>::readInflate

(comValues.\_shoot , -1.0f, 1.0f );

\_ctrlValues.\_updateTick = comValues.\_updateTick;

\_ctrlValues.\_controlledReplicaNetworkId = comValues.\_controlledReplicaNetworkId;

### do not continuously send projectile updates

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

(that's it on the clientside)

### implement client side interpolation

TODO:

### implement client side prediction for the controlling client

TODO:

### 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