SEMINARARBEIT

im Studiengang XXX

Lehrveranstaltung XXX

Arbeitstitel

Arbeitstitel

Ausgeführt von: Titel Vorname Name Titel  
Personenkennzeichen:

BegutachterIn: Titel Vorname Name Titel

Ort, Datum

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)

hint: a derivate from Dialog centered above the GamplayLayer might do this job

UI in dieser Methode einfügen:

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

Controls aus nlChatSample.cpp verwenden

void ClientChatNetPackageSender::createClientUI( PeerNode\* peerNode, CCControlBase\* parentLayer )

### 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

Antwort auf meine Frage zur Umsetzung: am besten dazu mehrere Instanzen starten mit Visual Studio!

Ansonsten viele GUI changes erforderlich und debugging wird sehr heftig

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

### 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

Padding ist ein wichtiges Thema. Unterschiede ergeben sich durch Anordnung von Variablen in Strukturen oder aber auch durch das Zielsystem und den Compiler (sowie dessen Optimierungseinstellungen.

nlProtocolStructures.h

ControllerValues: 16 Bytes statt 24 Bytes -> 8 Bytes eingespart

|  |  |
| --- | --- |
| Original: 24 Bytes | Komprimiert: 16 Bytes TODO |
| typedef struct TControllerValues  {  float \_leftRight;  float \_forwardBackward;  float \_shoot;  RakNet::NetworkID \_controlledReplicaNetworkId;  SLSize \_updateTick;  } ControllerValues; | typedef struct TControllerValues  {  float \_leftRight;  float \_forwardBackward;  bool \_shoot;  RakNet::NetworkID \_controlledReplicaNetworkId;  SLSize \_updateTick;  } ControllerValues; |

Dynamic2DActorDatagram 24 statt 32 Bytes -> (zuvor 28, jedoch mit Padding 8 Bytes möglich)

|  |  |
| --- | --- |
| Original: 28 Bytes | Komprimiert: 24 Bytes TODO |
| typedef struct TDynamic2DActorDatagram  {  float \_x;  float \_y;  float \_fx;  float \_fy;  float \_lvx;  float \_lvy;  float \_avz;  } Dynamic2DActorDatagram; | typedef struct TDynamic2DActorDatagram  {  short \_x;  short \_y;  float \_lvx;  float \_lvy;  float \_fx;  float \_fy;  float \_avz;  } Dynamic2DActorDatagram; |

Ansonsten könnte man noch in der BitStream.h die Compressed-Methoden für die jeweiligen Datentypen verwenden. Eignet sich z.B. gut für Quaternions.

### 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

in case not : explain why

in case : start implementing it and/or lineout what needs to be done to implement it

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

## Additional functionality (bonus points)

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

### add chat functionality to the game