Projekttitel: „The Tri World“
(Tri wegen 3 klassen die immer zusammen gehen müssen)

Umsetzung

Src-Baum:
- client
- server
  - account (enthält funktionen zum prüfen von account, erstellen, löschen)
  - network (clients_serve: enthält die funktion, die alle clients bedient, der reihe nach | clients_accept: enthält accept funktion)
  - package (enthält nur den verteiler, der ein packet liest und je nach cmdid in funktion springt)
    - functions (enthält alle funktionen mit Idxxx-name.c)
  - enemy (läd enemys dynamisch in maps | hält ki logic)
  - npc (läd npcs | bewegt npcs)
- share
  - package (enhält send,receive,alloc,free funktionen – keine unterteilung da wenig)
    - io (package **i**nput **o**utput: alle bei dem packages mit daten gefüllt werden oder gelesen werden)

Include-Baum (für client und server):
- be (hält alles was mit be zu tun hat)

namensgebung:
open – close (bei verbindungen)
init – deinit (bei systemen, modulen)
create – remove (bei dateien)
alloc – free (bei speicher)
add – del (bei listen)
start – stop (bei strukturen, die gefüllt werden)

ERRORCODE:

ID00	Alles okay, nichts passiert
ID01	Gewünschte anfrage konnte nicht ausgeführt werden

VOID POINTER LIST
eine liste für alle möglichen strukturen da void pointer.
Bedingung: es muss immer ein pointer geben.

Aufbau:
void pointer liste struktur typedef [vplist]
darin wieder startnode „first“ mit type : vpnode strukt

vpnode hat wiederum pointer auf nächste node.

vplist_vpnode_alloc
vplist_vpnode_free

funtkionen für liste: [share/vplist/vplist_vpnode_add.c], [share/vplist/vplist_vpnode_del.c]

zusätzliche funktion für alle löschen:
vplist_vpnode_delall

Include:
vplist.h (allen in eine header datei – einfacher leichter so)

SOCKET-SET:
wird benötigt, da ready abfragen.
Immer wenn client erstellt wird muss er in set aufgenommen werden.
Immer wenn er del wird muss er auch del werden.

Verbindung:
Server Client prinzip.
ca. 20 Kbyte/s downstream vom server, da t-com nicht mehr bietet (evt. Verbessern duch draufzahlen – einmalig monatlich?).. eigener server zu teuer. Host(dedicated) server zu teuer und monatlich gebühren.

SERVER:

Wartet auf anfragen und bearbeite diese.
Hält keine grafikdaten, nur zahlen (spielerdaten).

LOGGER [src/server/logger]:
erstellt logdatei mit datum und uhrzeit in extra ordner [log]
appendet immer nächte logzeile
logzeile hat zeit und datum + logtext
logdatei erstellt sich immer bei anfang (kann man ausschalten in arguments).
Eine logdatei pro programm start und stop.

Logger.h machen

logger_alloc() erstellt logdatei strukt – darin steht datei pointer

- datum zeit wird geholt
- datei pointer
- malloc von struktur

file wird nach jedem write geflushed fflush!

logger_free() ist klar

logger_write(logger_t *l, char *str, ...) schreibt printf mäsig in loggerdatei. Fügt zeit,datum automatisch dran.

VERBINDUNGEN:
wird in liste gehalten. Sind nicht mit maps verbunden.
Mit SDL realisieren.. ist einfacher und crossplatform.

SPIELDATEN:
es gibt public map und instanz map.
Publicmap memory ist immer da.
In memory wird gehalten:
liste der be's. Wenn brodcast nur zu denen senden die gamer sind (evt. Pointerauslagern).
Map muss wegen weg berechnung von enemy, npc.

INSTANZ:
es gibt platliste wo manche public maps sind und manche instanzen[es gibt eine instanzliste.] Wenn instanz angelegt wird, wird spieldaten erstellt.

LISTE DER CMDs:
alle cmds sind offen (wie wenn man alles steuern könnte). Egal wegen hack.

CLIENT STEUERUNG:
server brodcast [muss verteilt werden, aber nur in map an gamer] = sb)
zu jedem gibt es eine package-write-read-funktion:
anfragen vom client zum server.
Echte speicherverändernde funktionen!

ID	Was macht es	Erklärung	gesendet	src
ID000	Account erstellen	Erstellt einen Account. Schaut nach ob datei vorhanden ist, wenn nicht erstellen und template reinschreiben, also ungefüllter account.	Wunsch account name [gamername], wunsch passwort, klasse, aussehen	account_create(char *name, char *pw, byte class, gamerlook *gl) : bool
ID001	Account löschen	Löscht einen bestehenden account (Datei wird gelöscht)	Account name und passwort	account_remove(char *name, char *pw) : bool
ID002	Login in spiel	Wenn accountname exisiert kann man sich einloggen, gamer wird gefüllt, aktivflag auf true. Brodcast das neuer gamer da ist. (macht packet für client in den alle be's von plat enthalten sind und sendet auch gleich alles => evt. Auslagern [plat_content])	Accountname, passwort	account_login(char *name, char *pw, client *c) : enum(=> name falsch, name richtig pw falsch, schon eingeloggt, erfolgreich)
ID003	Logout aus dem spiel	Wenn aktivflag true – auf false setzen und	-	account_logout(client *c) : void

		gamerdaten löschen.		
ID004	Map wechseln	Wechselt die map – platid wird verändert. (client bekommt clearbe befehl und dann wieder content von plat [plat_content]). Brodcast be entfernen.	Platid	gamer_plat_change(byte platid, client *c) : void

VERBINDUNG CLIENT zu SERVER:

- Verbindung aufbauen (einfache athentifizierung)
- Client belibt verbunden ohne aktiv im spiel zu sein – clientstruktur wird für ihn erstellt die verbinung hält (wird aber torzdem verarbeitet was er will) gamerdaten nicht gefüllt.

ACCEPT:
erstellt clientstruktur mit aktivflag false und gamer ungefüllt.
Danach noch nicht im spiel – aber schon mal verbunden.

MAIN:
liest argument in der console:

ARGUMENTS AUS DER KONSOLE BEIM STARTEN: [in src/server/arguments/arguments_proceed.c] mit arguments_proceed(int argc, char *argv[], mem_t *mem)
port muss angegeben werden, auf was er hören soll.
Opt. Die einstellungsdateien.
Evt. Max verbindungen

variabel machen -p port zb. Und --no-log für logger ausschalten.

bei ausgabe arguments_print_xxx.c (xxx zb = version)

liest die argumente aus und speichert sie in memory.

Es geht nur weiter wenn auch port angegeben ist.

Inizialiseirt alles (netzwerk) [main_init]

While: [main_programm]
verbinungen hinzufügen [accept_proceed] =>benutzt client_accept funktion
mit clienten communizieren [serve_proceed] : Frage – antwort. Liest packet, distributiert, Ki (enemy, npc).

While wird mit variale am laufen gehalten. Kann mit strg+c signal [src/server/signal/]auf false

gestellt werden. Üblicher deinit schluss.

zum schluss
[main_deinit]
alle clienten die sich acceptet haben löschen – closen.
Extra aufwand (wohin damit?):
sockets vom socket set entfernen
sockets schließen
clienten frei machen – vplist clientlist leer machen.

Aufgaben die server übernimmt in ordner spalten.

PACKAGE[NETWORK]IO (bei client und server gleich)

Netwerk io schreiben mit packet art. (flush). Writebyte readbyte usw.

write:
schreibt nicht in verbindung sonderen in paket [package (network/package.h)].
read liest von angekommenden package.
Flush sendet komplettes package rüber.

write – byte, short, int, string(1 byte pro char – sonderzeichen ü,ö,ä per client in ein byte packen und von server interpretieren)
read – byte,short, int, string
send
receive
alloc
free

Methodenliste:

In src/share/package/io/basic_write.c		
WriteByte [byte_write()]	Byte value, package *p	void
WriteShort [usw.]	Short value, package *p	void
writeInt	Int value, package *p	void
writeString	Char *value, package *p	void
In src/share/package/io/basic_read.c		
ReadByte [wie write nur mit read]	Package *p	byte
readShort	Package *p	short
readInt	Package *p	int
readString	Package *p	*char

In src/share/package/package_send.c		
SendPackage [package_send]	Package *p, SDL_Verbindung *socket	Void
In src/share/package/package_receive.c		
ReceivePackage [package_receive]	SDL_Verbindung *socket	*package
In src/share/package/package_alloc.c		
AllocPackage [package_alloc]	(ohne size, weil man nicht weiß wie groß es wird: deshalb sehrgroßes array erstellen zb. 1024 bytes) – trotzdem mit malloc machen!	*package
In src/share/package/package_free.c		
FreePackage [package_free]	*package	void

PACKAGE SEND:
package stream:

Size (2 byte)	Memory (size byte) (byte[0] = cmdid, byte[1] = errorcode)

PACKAGE RECEIVE (receivePackage):
wartet auf 2 bytes.
Wenn 2 bytes da sind werden sie gelesen.
Short wird als size gespeichert.
Es wir auf size bytes gewartet.
Package wird mit memory und size zusammengestellt.
Weiterleitung an switch-case mit funktionspointer.

PACKAGESTRUKT (network/package.h):

pakete werden von dem client zusammengeschnürt, um ein befehl zu senden.
Strings können unterschiedlich groß sein. Deswegen sind pakete unterschiedlich groß. Deswegne muss speicher dynamisch sein.

Da man schon weiß, was man versenden will, wird packetgröße dyn daraus erzeugt.

Memory enthält cmdid.
Errorcode setckt immer in memory drin (wenn es eine antwort ist).
Errorcode muss am anfang stehen.

Memory – hält alles was von den writemethoden reingeschrieben wird – muss dyn erzeugt werden. Durch sizeof bekommt man max größe der memory.	*byte
curpos [auch size] (wird immer hochgesetzt bei writemethoden) es ist wirkliche size, nicht dyn angelegte size.	short

PACKAGE READER [server/package/distributor_proceed.c]
muss so gemacht sein, dass er mehr cmds ausliest.

Algorithmus: (in funktion package_distrubutor_proceed(client *c, package *p, mem *m) : void)
prüfen ob curpos und size, damit abschlussbedinung

Liest byte cmdid. [Curpos immer hochsetzen – (wird von basisfunktionen read schon macht)]
Liest byte errorid.
wenn errorid bestimmter wert nicht weiter lesen => wiederholen

package an funktion fptr[cmdid] geben.

Dort package weiter lesen.
Daten verarbeiten.
Bordcastpackage füllen oder direkt anwort zurücksenden.
Wieder von vorne beginnen.

NETWORK:

DISTRUBUTOR:

NETZWERK-FUNKTIONEN (unterteilen in aufgaben):
funktionen mit client* (von wem es kommt), package * (was er sendet), hauptspeicher * (memory für speicherveränderung)

FUNKTIONSPOINTERARRAY ERSTELLEN.

Bsp:
int (*ptr[])(const char *, ...) = { scanf, printf };

funktionspointer-array:

int (*fptr[])(client *c, package *p, mem *m);

verteilungsfunktion:

nimmt ein package und führt prozess zu richtiger funktion.
In funktion wird memory von package gelesen und verarbeitet.
SendePackage wird dann erstellt (falls nötig – eigendlich immer wegen errorcode).

PACKAGE VERARBEITUNG:

brodcast anfragen – anfragne die viele betreffen
einmal anfragen – zb. Account erstellen (hat kein brodcast)

Alle verbundenen PC auf anfrage durchsuchen in unsortierter liste aber egal.
Danach augenmerk auf enemyki und npc ki richten.

GAMER bevorzugen weil sie die echten spieler darstellen, danach enemys, npcs bearbeiten.

Es gibt die netzwerkfunktionen:
zb.
int ID000_account_create(client *c, package *p, mem *m);

in ihnen wird das package mit basisfunktion auseinandergenommen und die wirkliche funktion wir aufgerufen (account_create(parameter)).

Dateien im stil ID000_account_create.c in ordner src/server/package/functions/*

[(ist name nötig: später dann nur fptr[0] = ID000)]
name schon dranlassen damit man weiß was sie macht.

PACKAGE VERARBEITUNGS INIT: [src/server/distributor/distributor_init.c]

init code:

m->distributor.fptr[0] = ID000_bla;
usw.

MAPBRODCAST:
ein package für alle senden.
Package kann in memory mehr cmds enthalten.
Es können mehrer aktionen geleichzeitig versendet werden...spart http header.

CLIENT IM SERVER:
client in liste.
Accept clients in liste aufnehmen.
Jeder client hat speicher für seinen gamer.
Er kann per befehler speicher ändern (änderung muss gebrodcastet werden).
Um in map zu komen muss er mapid [platid] verändern.
Client hat verbindung zum lesen und schreiben.
Client ist nie direkt mit plat verbunden.

CLIENTSTRUKTUR (network/client.h):

SDL_verbinung [socket]	*SDL_verbindung
Akiv – um zu wissen, ob der spieler im menü auserhalb des spiel ist oder ob er aktiv im spiel ist – bei false ist gamer ungefüllt – gilt zb für accountersteller.	bool
Gamer – alle gamerdaten	Gamer (fester speicher)

(clientliste muss angefertigt werden)

NETZWERK CLIENT FUNKTION: [server/network/clients_serve.c] =>
network_clients_serve(mem *m)
bedient clients:
schaut nach ob sie etwas auf der leitung haben oder ob sie disconnected sind.

=> wenn disconnected:
aktivflag auf false setzen.
accountdaten updaten.
Client von liste entfernen.
Brodcast rausschicken, dass er weg ist.

CLIENT ALLOC FREE
server/network/client/network_client_alloc.c | (void) : client_t *
server/network/client/network_client_free.c | (client_t *) : void

INIT:
init für network ansich

SERVERSOCKET:
open socket für serversocket
close ebenfalls
[network_socket_open]
[network_socket_close]
(kein serversocket weil es ja nur ein socket pro server gibt)

TCPsocket nie pointer machen, da es schon pointer ist!

NETZWERK ACCEPT: basic function: [server/network/client/network_clients_accept.c] =>
network_clients_accept(TCPsocket *socket)
schaut nach ob jemand sich einklinken will.

in extra function:
Authentifizierung.
Fügt dann in m->clientliste hinzu.

NETZWERK BRODCAST: [server/network/plat_brodcast.c] => network_plat_brodcast(mem *m)
plat brodcastpackages müssen gesendet werden.

Algorithmus:

geh alle plats durch. Dort wo package beschrieben wurde, an alle die selbe platid haben senden.

SERVER HAUPTSPEICHER (mem.h):

- platliste (platliste ja nur wegen enemybewegung und brodcast)
- enemyliste (enemykisystem) (enemy selbses system wie bei client)
- npcliste (npckisystem)

- netzwerkdaten [network.h] (nw)
  - port
  - sdl netzwerk stukruren
  - clientliste
- funktionspointer-array [distributor.h] (dtb)

ACCOUNT:

Ein spieler (also gamerobjekt) ist ein account.
Es ist nicht so das ein account mehrere gamer verwaltet.
normales accountsystem. Server hält alle infos von spielern.
Ist in einlesbare dateien angelegt im format: [name des gamers].account .
Speichert folgende werte:

Name des gamers (oder auch accountname)
Passwort (unverschlüsselt)
Letzte besuchte map als id
Letze gewesene location als x y in map (2 byte)
Gamerdaten (name, battledaten, attributdaten, ausrüstung, klassenspezifische daten(zb. Items bei händler))

GAMERDATEN SPEZIALISIERUNG:

Alle attribute (6 + 3) – bei hp beides (cur,max) damit nicht ausloggen und einloggen und geheilt! [battle.h + gamer.h]
Name nicht...ist ja schon in obrigen drin
Ausrüstung (steckt ja in gamer.h)
Klassenspezifische daten

ACCOUNT LESE SYSTEM:

nicht in arbeitsspeicher schreiben.
Dabei wird gesucht. Nur Passwort wird gelesen und verglichen.
Wenn okay dann wird datei vollständig geladen.
Gamerdaten werden für client gefüllt, und gebrodcastet.

Änderung an gamer wird nach bestimmten zyklen (alle 60 sek.) in datei geschrieben.

ACCOUNTSCRIPT:

muss so geschrieben sein, dass es leicht veränderbar ist durch das system.

ACCOUNT SOURCE:

man muss können:

- account erstellen [account_create() schon oben in cmd-liste]
- account löschen [account_remove()]
- account updaten [account_update(client *c) - in regelmäßigem abstand | immer wenn client ausloggt]

PLAT POINTERS:

plat-liste machen da dyn plat erzeugen.
In plat gibt es be-pointer-list für enemy, npc, usw.
client-pointer-list ebenfalls.
Wenn enemy in plat geht, wird enemy zu pointerlist hinzugefügt.

CLIENT:

Wie bei Guildwars: eine *.dat datei, die eine pack-datei ist, die erweitert werden kann. Neben dran die exe datei.
Alle dateien sind einsehbare souredateien. Client kommt nicht an dat ran, da passwortgeschützt.

VERBINDUNG:

Server zu Client:

ID	Was sagt es aus	Erklärung	Server sendet	src
ID000	Account erstellungsstatus	Server sendet, ob der account erstellt wurde.	Nur errorcode.	
ID001	Account lösch status	Server sendet, ob account gelöscht wurde	Nur errorcode	
ID002	Login status	Server sendet, ob login erfolgreich war.	Nur Errorcode	
ID003	Be entfernen	Weißt client an, ein be zu entfernen	Id von be	
ID004	Erstelle gamer	Weißt client an, einen gamer zu erstellen (weil er neu dazu gekommen ist)	Gamerdaten des spielers (aussehen, location)	
ID005	Erstelle enemy	Weißt client an, einen enemy zu erstellen.	Enemy id	
ID006	Erstelle npc	Npc erstellen	Npc id	

ID007	Erstelle platbeobj	Platbeobj erstellen	Platobj id	
ID008	Erstelle material	Material erstellen	materialid und anzahl	
ID009	Erstelle Questgegenstnad	Questgegenstand erstellen	Questgegenstand id	

NETWORK:

funktion zum verbinden und schließen der verbindung (zum server).
network_connect() : TCPsocket – mit string adresse und port (da adresse auch ip sein kann)
network_disconnect() :void

console ist beim clienten zweitranig da wenn er fertig ist nichts über die konsole ausgegeben wird.
Logger sollte doch verhanden sein.

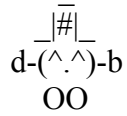
Für jeden befehl muss es eine packet zusammenschnürr funktion geben.
zb. für account erstellen.

account_create(char *name, char *pw, byte class, gamerlook *gl) : bool

Funktion nimmt die parameter und steckt sie in package.
Später wird package versendet.

Distributorsystem wie bei server erstellen. Mit funktionen genau wie bei server.
Das heißt von server lesn.

Zu anfrage pakete zusammensetzen.



ARCHIV:
c source suchen....
tar

Client wird gestarte und verbindung wird schon begonnen.
Wenn keine verbindung hergestellt werden kann, wir kein menu aufgebaut und gesagt, dass server off ist.

FENSTERGRÖßE:

17x14 tiles (tiles 32x32)
(ungerade halten damit scroll immer mittig ist, gleichberechtig nach allen breichen gleiches sichtfeld)

MAIN ANFANG:
alles initialisieren
mit server verbinden (authentifizierung)
cmd 000 mit location, map, gamerinfo
{server hat schon public map}
{add gamer in liste von map ein, (brodcast), gibt zugewiesene id zurück}
client läd map
sync

DATATREE:
ordnerbaum um daten zu halten (immer einzahl):
plat – hält mapscripte
platobj – hält map unabhängige objekte für die map
platgraphic – hält nur ein png für alle map darstellungen und animationen

GRAFIK:
Resourcen durch FTP HTTP server stellen.
tiles in 16x16 lassen und im programm um bestimmte anzahl scallieren.
Ein riesen großes png das alles was map braucht enthält...wie updaten? Alles? → Unnötig doppeltes gesendet. Bei tilegröße 32x32 bei tilemap von 100x100 also 10000 tiles sind es 3200x3200 pixel. Png ist 2,4 mb groß.
Bei 60x60 tiles in 16x16 pixel tile größe sind es 3600 tiles. Logischer als vorher. Dateigröße 230 kb. Vertretbar. Update durch http server → schnelle rate, keine resourcenverschwendung bei gameserver.
Mapgrafik von Spielergrafik trennen. Selbes system wie bei map.

SPIELERGRAFIK:
wie bei map besteht aus tiles.
Nur max vorne, hinten, seitlich (da seitlich spiegeln) [← große frage wie grafik aussehen wird]

- Haare (brauchen max)
- Kopfform (ist nicht so variabel unbedingt, evt. streichen)
- augen (nur von vorne [bei RPGmaker kein unterschied], nur eine auge → später duch programmierung variabel machen)
- körper(alle seiten evt. + extraseitlich wenn rechts anderes wie links also spiegeln nicht möglich)
- hose (wie bei körper)

spieleraussehen verweist auf index des tiles.

ITEMGRAFIK:
wie bei map tileorientiert. Item referenz auf x y (byte groß).

MAP INFO DEFINITON:
pro map eine datei (sonst wird es zu viel für eine datei)
map bekommt eindeutige id (byte langt)
maptyp?

MAP-AUFBAU:

MAP [plat] (nur aussehen der welt, keine spieler):
script, dass erklärt wie die map aussieht, damit kein mapeditor nötig.
Hält information zu start-aufenthalt von monstern. Nur für server gedacht.
Hält info für teleporter zu anderen maps. Da client und server sync sind, kann client dies berechnen.

beinhaltet

OBJEKT [platobj]:
wieder script, dass erklärt wie obj aus tiles zusammengestellt ist, collision usw.

referenziert (wie auch map) mit index x y (byte langt) auf feld

TILEMAP [platgraphic]:
die eigentliche grafik in png

PLAT (nie zwischen client server aktiv herumgeschickt):

daten wird aus platscript erstellt:

PLATSCRIPT:
man zeichnet mit tiles.

- kreise
- rechtecke

zeichnen wie auf 256x256 pixel bild

ALTERNATIVIDEE:
pixeldaten als x,y interpretieren und pixelbild verwenden.
Farb – tile zuordnung in liste.
Die wird geladen.
256x256 bild wird durchlaufen. Unbekannte farbe bleibt schwarz.
Bekannte farbe wird wie tile übernommen.

man setzt platobj an stellen.
Enemys setzen.
npcs setzen.

Script zb verletzt spieler oder teleport zu map...usw.
Schalter scripten events

PLATSTRUKTUR (plat/plat.h):

Fieldarray mit maximaler anzahl (ist ja nur	Field[256 * 256]

speicher – einfacher so).	
Brodcastpackage	Package (echter speicher)

(zuordnung wer in welchem plat ist, steht in platid bei jemdem be! - einfacher so [wie bei partyid])

Beliste durchlaufen und je nach typ etwas machen. Gleichberechtigt.
Enemy – bewegen, angreifen usw (enemy-ki)
npc – bewegen (npc-ki)
gamer – verbindung nachschauen, auf cmd nachschauen

FIELD IN MAP:
enthält referenz auf platgraphic mit x y data (in byte)
enthält collisionsbedingung für alle verschiedenen arten von dingen.
Enthält animationsdaten wenn animation sein soll.
Enthält script logic.. zb. Teleporter usw.

FIELDSTRUKTUR (plat/field.h): (wird nie roh übertragen – script sync zwichen client und server):

Maxframe (bei 0 ist es keine animation sonderen nur ein bild – andere anmationseigenschaften wo anderes speichern oder garnicht)	byte
Curframe (jetziger frame)	byte
Unbestimmte anzahl von frames (dynamisch erzeugen anhand von maxframe)	*fieldframe
collision	Byte (bitorientiert evt. Bit1 für gamer, bit 2 für enemys usw.)

FIELDANIMATION:
absoluter frame switch timer – keine extra timer. Wie bei zelda die blumen bewegen sich immer sync.

FIELD FRAME STRUKTUR (plat/fieldframe.h) für animation:

Layerarray	fieldlayer[2]

Ein farme ist ein tile. Ein tile hat aber immer zwei layer.

FIELD LAYER STRUKTUR (plat/fieldlayer.h)

X	Byte
Y	Byte

X,y von 0 – 255.

Layer:

wenn layer1 und layer2 da und collison an dann immer charakter davor zeichnen.
Wenn layer1 und layer2 aber keine collison dann immer tile über charakter zeichnen.

DINGE IN DER MAP:

ACHTUNG: Unterscheidung zwischen **platobj** (es ist statisch, nicht verschiebbar, aber trotzdem selbe quelle).
Für verschiebbare objekte gibt es **platbeobj.**

Alles was aktiv in map ist, ist ein [be]**.**
kann sein:

- aktiver richtiger spieler [gamer]
- npc [npc]
- monster [enemy]
- platbeobj (→ rückbarer stein, tür die sich öffnet)
- materialie (bei itemgrafik dabei)
- questgegenstand
- questwand (die nur mit schlüssel aufgemacht werden kann)
- schalter, der platbeobj bewegt

BASIS [be]:
besitzt eindeutige id für aktuelle map (byte langt, da max 256 aktive dinge in einer map (spieler, npc, monster) – schon fast viel zu viel).
Bestitzt locale x y für aktuelle map (byte langt auch).
HP wert, cur und max.
name

BASISSTRUKTUR (be/be.h):

Id (id in plat) (wird gesendet)	byte
X (wird gesendet)	Byte
Y (wird gesendet)	Byte
Platid -sagt aus wo be sich gerade befindet (nur info für server)	byte
Name {otional : für gamer (wird gesendet)}	*char
Type (wird gesendet)	Byte (typ welches be es ist: gamer, enemy usw.)
Refid (referenz wenn es kein gamer ist, auf array von bestimmtem typ zb. Enemy oder npc) {optional}	byte
Curhp {optional}	Byte
Count {optional}	byte

(evt. Blickrichtung – naja eigenltich aus bewegung erkennen)

Verworfen!:
[BATTLESTRUKTUR (be/battle.h) [für alle be's die angreifen und verletzt werden können]:]

[Hpcur]	[Byte]
[Hpmax]	[Byte]
[power]	[Byte (angriffstärke)]

[defense]	[Byte (abwehr)]

Power und defense haben enemy schon extra und gamer auch.
Hpcur hat auf jeden fall gamer und enemy, evt. Npc (aber npc kämpft nicht und sollte auch nicht tötbar sein)

AKTIVER SPIELER [gamer]:
besitzt die basis eigenschaften.
Klassennummer – klassenid. (brawler, trader, quester).
Partyid – in welcher party man ist (byte = 0 → keine party).
referenz zu verbindung
Zusammenstellung des aussehens.

GAMERSTRUKTUR (be/gamer.h):

Classid (welche klasse man hat)	Byte
Partyid (in welcher party man ist – innerhalb von plat)	Byte
Verworfen : [Slot-array mit byte als itemid (geht bei keiner itemliste nicht)]	[byte[7]]
Item-Slots die getragen werden (kein brodcast)	Item[7] (itemstuktur steht unten)
Für attribute 9 byte werte (auch power, defense und maxhp!) (kein brodcast)	Byte[9]
Look (brodcast!)	Byte[5] (array vereint alle eigenschaften)

Verworfen!:
[GAMER AUSSEH STRUKTUR (be/gamerlook.h)]
[

hair	byte
Eyeleft	Byte
Eyeright	Byte
Body	byte
Pants	byte

]
Look in gamer mit hinein schreiben.

Verworfen!:
[GAMER AUSRÜSTUNG STRUKTUR (be/gamer/equipment.h)]

Slot-array mit byte als itemids	byte[7]

[GAMER ATTRIBUTDATEN STRUKTUR (be/gamer/attribute.h)]

Für klassenattribute 6 byte werte	byte[6]

Da so klein direkt in gamer.h speichern

MONSTER:

definition von monster in eigner datei (nur einer datei).
Monster wird serverseitig gesteuert.
In Server einbauen: Monster-ki-anwendung [] enthält monsterlogic + ki usw.
Dort wo spieler sind müssen monster hin, ansonsten nicht.

MONSTERDEFINITON:
jedes monster hat monsterid (nur byte 0 – 255).
client server sind sync bevor spiel startet.
Element

ID	Name (zb skellet)	angriff(stärke)	abwehr	aussehen	Element (auch keines möglich)	Max hp	Unipoints belohnung (bei tötung) steht in definition	Welche materialienf allen gelassen werden und wie viel damit random weis zwischen was

Schon in be enthalten.
[
MONSTERSTRUKTUR:

enemyid	Byte verweißt auf id in liste

]

MONSTERSCRIPT (hat client und server sync):
zeilenweiße
script:
(trennzeichn ist leerzeichen = name darf kein leerzeichen haben)
[id = 000] [name] [look] [maxhp] [power] [defense] [element] [unipoints] [marialtyp]:[min drop]-[max drop],[marialtyp]:[min drop]-[max drop]

monster nach routen und dungeons sortieren wo sie auftauchen.
Nach schwierigkeit sortieren.

In map verweist dann auf monsterid oder monstername und setzt es hin sich hinein mit location.

MONSTERGRAFIK:
monster von spielergrafik trennen, da kein spieler ein monster sein kann.
Sollen monster im zusammenstellungsmodus wie spieler sein? Ist das nötig? ..nein..
(evt. Große monster darstellen können).
[Monster sind nur von vorne sichtbar und haben 3 animationen : linkerfuß, rechterfuß, steht (egal auf alle ansicht)]
alle ansichten unstüzen: ist clientarbeit.
Pro monster eine grafikdatei – sonst zu viel und zu unübersichtlich.
Dateiname im stil: enemygraphic(oder graphic/enemy)/005.png für look 5.

rpggrafik exportierer, der zuschneidet.

MONSTERGRAFIK MANAGEMENT:
array mit variabler größe, da monster hinzugefügt werden können (größe durch datei zählen ermitteln oder aus infodatei lesen).
Index des arrays ist dann look des monsters.

MONSTERGRAFIK INITIALISIERUNG:
array erstellen mit variabler größe (siehe oben).
Array hält SDL bilder (immer von einem monster).
Arry[3] muss haben enemygraphic/003.png.

NPC:

NPCDEFINITON:
eingebaut in server. Definition und text für npc in sourcedatei (nur eine datei).
Referenz auf npc mit id.
Questtexte neutrale bemerkungen .. keine aufforderungen.
Evt. Questgegnstand übergabe.
Npc hat npclvl!

ID	Name	Location	Aussehen	npclvl	Text (von liste)	Questbelohnung (nur in unipoints)	Wollender Questgegenstand

Schon in be enthalten
[
NPCSTRUKTUR (be/npc.h):

npcid	Byte (refferenz zur liste mit den ganzen npcs)

]

Text muss nicht in struktur gespeichert werden da referenz mit npcid

NPCSCRIPT:
(zeilenweiße, name ohne leerzeichen, qg = questgegenstand)

[id = 000] [name] [lvl] [unipoints] [qgid-geben],[...] [qgid-wollen],[...]

[Text]

end //beendet npctext und neuer npc fängt an

NPC-LISTE:

ID	**Name**	**lvl**	**Ort**	**Text**	**Questgegnstand geben**	**Questgegenstand wollen**	**unipoints**
ID000							

ITEM: (kann nicht herumliegen in plat)
hat keinen speziellen namen.
Hat typ: ist es für slot2brawler, slot5quester usw.
erklärt welches attribut (6 + 3 = 9) wie verändert wird.
Items können max 3 attribute beeinflussen.
Items können nicht auf den boden geworfen werden.
Quester und brawler können keine items zwischenlagern.
Ausgerüstete items können ersetzt oder weggeworfen werden.

ITEMSTRUKTUR (?/item.h):

type	byte
Itemattribut-array da mehrere gleichzeitig verändert werden können	itemattribut[3]

ITEMATTRIBUT VERÄNDERER (?/itemattribute.h)

attributid	Byte (von 1 – 9) (0 kein attribut verändern)
value	Signed byte (da auch schlechter werden lassen) value wird zu festem attributwert addiert.

MATERIAL:
[materialid referenz zu materialliste] Eher materialtyp.
Anzahl festhalten
grenze max count = 250.
werden von enemy gedropt für trader.

Verworfen! (schon in be enthalten)
[
MATERIALSTRUKTUR:

[materialid] ist schon in be enthalten	[Byte (wie bei item) ist zb. 00=stein, 01=holz usw.]
[Count] in be drin	[Byte, Anzahl der materialien]

]

MATERIAL-Liste:

holz	ID00
stein	ID01
kraut	
wasser	
stahl	

baumwolle	
papier	
feuerdiamant	
wasserdiamant	
Erddiamant	
luftdiamant	
stroh	
leder	
knochen	

MATERIAL-SCRIPT:
[id] [name]

MATERIAL ITEM ERSTELLUNG:
materialien werden gebraucht um rüstungsitems für alle klassen zu schmieden.
Umsomehr materialien eingesetzen werden umso stärker wird die geschmiedete rüstung.
Manche items brauchen nur ein material andere brauchen mehrere.

Schmiede menü pro klasse (nicht pro zu schiedendes objekt zb. Helm oder hose).
Keine hilfe beim erstellen, händler muss selbst herausfinden, was welches attribut hochsetzt.
(Man kann zb. 2 holz und 2 stein nehmen. Bei helm macht das 2 abwehr. Bei schwert macht das 4 angriff.)

händler hat schmiedekunst für klassen. Umsohöher diese ist, umso bessere rüstungen entstehen.

zb. stahl gibt abwehr. Ein abwehrpunkt zur rüstung braucht 30 stahl.
Bei schiedekunst 2 brauch man nur noch 28 stahl.

Händler sammeln materialien. Können nur 5 tragen. Setzten 5 ein um item zu machen mit abwehr 1. vk item und bekommen unipoints. Steigen in schiedekunst auf. Stellen materialtragelast hoch oder schmiedekunst.

QUESTGEGENSTAND:
bekommt man von npc oder wird von monster gedropt oder wird gefunden in dungeon.

Diesmal questgegenstandliste führen wo alle aufgelistet sind mit id und namen und text.
In map wird der questg. Wohin gelegt.
Monster haben in monsterdefinition den questgegnstadn als id drin, wenn sie eien solchen droppen.
Npc haben in definition wie bei monster drin.

Questgegnstnad hat bestimnmte schwere.

Public questgegenstände sind gegenstände in den public maps.
Questgegntand vorrat ist unendlich.
Es wird aufgebhoben aber verschwindet nicht.
Zb brunnen gibt wasserflasche.

Questgegenstand ist zb. Alle arten von schlüssel.
Zb schlüssel der dunkelheit umd die todespforte öffnen zu können usw.

QUESTGEGENSTANDSCRIPT:

[id = 000] [schwere = 0 – 250]
[name (kann aus mehr wörtern bestehen]

[erklärungstext (mehrzeilig)]

end

Verworfen! (steht schon in be)
[
QUESTGEGENSTANDSTUKTUR:

questobjid	Wie bei material, type verweißt auf liste

]

PLATOBJ:

platobj ist mit script definiert.
Eine datei für alle.

PLATOBJ-SCRIPT:
[id = 000]

[platobj-aussehen, collison, usw.]

end

Verworfen!
[
PLATBEOBJ-STRUKTUR:

platobjid	Verweißt auf platobj-liste

]

SPIEL:

entscheidung über klasse:
kämpfer [brawler]
händler [trader] in supporter umbenennen?
quester [quester]

(geschichte muss außenherum sein. Quester kann nur in geschichte weiterführen.)

Attribute durch bezahlen von uni-points hochsetzen.
Nächstes lvl kostet mehr uni-points als zuvor.

Standard attribute jeder Klasse [(ist in be/battle.h enthalten)]:
max hp
angriff
abwehr

Standard Menüs:
ausrüstung-menü:
7 ausrüstungen
abitlity-menü:
3 basis + 6 kassenabhängig: 9 abilitys

Kämpfer:
Hauptaufgabe:
klickt monster an und macht dmg.

Stärke	Dmg wert
Mut	Angriffschnelligkeit
Feuerweißheit	Elementresitenz gegen Feuer
Wasserweißheit	Elementresitenz gegen Wasser
Erdweißheit	Elementresitenz gegen Erde
Luftweißheit	Elementresitenz gegen Luft

Menü:
keine weiteren

Händler:
Hauptaufgabe:
Muss zu spieler sachen verkaufen. Kann erworbene materialien bei npc verkaufen oder bei schmiede zu item machen und an spieler verkaufen.

[traglast]	Anzahl der materialien halten (zum herstellen von rüstung)
heilkunst	heilungsstärke
heilausdauer	Anzahl der heilungen die man machen kann
Händlerrüstung-Schmiedekunst	Verbessert bedingung und stärke für rüstung
Kämpferrüstung-Schmiedekunst	Verbessert bedingung und stärke für rüstung
Questerrüstung-Schmiedekunst	Verbessert bedingung und stärke für rüstung

Menü:
(bei npc : schiedemenü)
item menü – um erstellte items zu halten und dann zu verkaufen
material menü – um aufgesammelte materialien zu halten
verkaufmenü

Quester:
Hauptaufgabe:
klickt npc an und liest deren belangen und informationen. Muss daraus schlussfolgern und in gebiet gehen und etwas holen, auslösen, finden. Schafft das wegen monster nicht allein, deswegen kämpfer. Kann sich nicht selbst heilen, deswegen händler.
Kann items auf npc anwenden um ereignis auszulösen.
Nur quester können in dungeons gehen.

questgegenstand-traglast	Damit schwerere Questgegstände tragen kann
menschenkenntniss	Damit npc mit höhrem lvl ansprechen kann
Dungeonerfahrung	Damit dungeon mit höhrem lvl hineingehen kann
Schlüsselscharfsinn	Damit schlüssel mit höherem lvl einsetzen kann
Hebelscharfsinn	Damit Hebel mit ...
- scharfsinn	

Menü:
Questitem menü
schlüssel menü

Nur HP kein MP nötig.
Universalpunkte [unipoints, uni] sind in accountdata.

Anfangsstartpunkt in stadt: MIJI.
In Gruppen zusammen finden und route ablaufen und in instanzen dort gehen.

Es gibt 3 maparten:
stadt: sind alle ..gibt es nur einmal, keine enemys (zB. Hauptstadt : Miji)
route: gibt es nur einmal wie bei stadt aber + enemys (zB. Route zwischen Miji und Weto)
dungeon: ist eine instanz + enemy, + rätzel (zB. Tempel)

MONSTER:
lassen materialien für handler zum erstellen von richtigen items fallen. Mehr nicht.
Getötetes monster gibt nur brawler unipoints

MATERIALIEN:
nur für handler. Der kann es aufsammeln.
Nur in städten wo schmieden sind kann man items erstellen und dann verkaufen.
Materialien haben immer eine anzahl.
Händler hat also mehrer stücke holz zb.

ITEMTYPEN:

Helm	1
Schulterrüstung	2
armrüstung	3
brustrüstung	4
beinrüstung	5
schuhe	6
schwert	7
hut	1
handschuh	3

werkzeug	2
Tasche	5
rucksack	4
beutel	6
hammer	7
turban	1
schlüsselbund	
wanderschuhe	
kiste	
abzeichen	
lupe	
stab	

ITEMS:

rüstungsitems für brawler (zum verstärken)

- helm
- schultern (ein item für beide schultern)
- arme (ein item für beide arme)
- brust
- beine (ein item für beide beine)
- schuhe (ein item für beide schuhe)
- schwert

trageitems für händler (zum besseren tragen oder mehr tragen)

- Werkzeug
- handschuhe
- hut
- tasche
- Rucksack
- beutel
- waffe

Questeritems

- landkarte
- schlüsselbund
- wanderschuhe
- kiste (für questgegenstände)
- abzeichen
- lupe
- waffe

STÄDTE:

ID01	MIJI	Hauptstadt in der mitte	Kreisgrundfläche, mitte hoher turm, häuser außenherum, stadtmauern mit 4 toren (zu 4 gebieten [norden, süden, westen, osten])
ID02	WESGA	Sumpf-Wald-gebiet	
ID03	MISU	Sumpf-Wald-gebiet	
ID04	WABA	Sumpf-Wald-gebiet	
ID05	DABOK	Berg-höhlen gebiet	
ID06	HOBA	Berg-höhlen gebiet	
ID07	BAGE	Berg-höhlen gebiet	
ID08	SUTOKI	Berg-höhlen gebiet	
ID09	SONU	strandgebiet	
ID10	SHENI	strandgebiet	
ID11	SANDA	strandgebiet	
ID12	AQUI	strandgebiet	
ID13	WETO	wüstengebiet	
ID14	NAKO	wüstengebiet	
ID15	SHIMU	wüstengebiet	

ROUTEN:

20 stück