Fordítás

- SDL2 könyvtárat igényli: Link

Felépítés

- type

  > Itt található minden típushoz kötődő dolog.
  > A célja, hogy az objektumok létrehozása és felszabadítása ne ismétlődjön, azok alapértelmezett értékei egységesek legyenek.
  > Előnye az, hogy hibákat is ki lehet iktatni, ha csak egy helyen kell helyesen megírni a dolgokat.

- resource

  > Itt találhatóak a betöltendő képek a típusnak megfelelő mappában 0-tól kezdődő indexeléssel, png formátumban.
  > Bár nem volt feladat az animáció, de ez kihagyható ha csak 0.png nevű képeket adunk meg.

- main

  > Ez tartalmazza a modulok indításának vezérlését.

- client

  > Ez az egység felelős a megjelenítésért, illetve a bemenetek kezeléséért (kivéve a pálya mentése).

- server

  > Ez az egység felelős a pálya létrehozásáért/betöltéséért, illetve annak következő állapotának kiszámításáért.
  > Szintén tartalmaz autentikációt, egy client ellenére, hogy felületesen imitálja a server client modellt.

- SDL

  > Itt találhatóak az SDL-hez kötődő dolgok: inicializálás, textúrák cachelése, ...

- geometry

  > Ide minden a pályához, annak szerkezetéhez kapcsolodó függvények tartoznak pl.: ütközés

- config

  > A nevéből is adódóan itt a játék működését lehet szabályozni.

- network

> Bár nincs valódi hálózat a játékban, de későbbi fejlesztést megkönnyíti egy absztrakt modul. Itt lényegében a client és a server modul közötti konverzió történik.

Adatszerkezetek

- List

  > Olyan helyeken szerepel, ahol a mérete sokszor változik, akár mindkét irányba.

- Array

  > Olyan helyeken szerepel, ahol a méret változik, de csak növekedhet.

Függvény leírás

> Nincs mindenhol említve, hogy fel kell szabadítani a memóriát, mivel a visszaadtott adatszerkezetből egyértelmű ez.
> Ahol nincs megengedve a NULL paraméter ott tilos.

- type
  - *

```
//TypeNew creates a new Type
Type* TypeNew(void)

//TypeDelete frees Type
void TypeDelete(Type* type)
```

  - list

```
//ListInsert inserts data into List by reference
ListItem* ListInsert(List** list, void* data)

//ListInsertItem appends ListItem to List by reference
void ListInsertItem(List** list, ListItem* listItem)

//ListRemoveItem removed ListItem referenced from list
//dataFree is called on ListItem->data if it is not NULL
void ListRemoveItem(List** list, ListItem* listItem, void
(*dataFree)(void*))

//ListFindItemByFunction returns first ListItem where
func(ListItem->data) holds
ListItem* ListFindItemByFunction(List* list, bool (*func)(void*))

//listFindItemByPointerFunction is a helper function of
ListFindItemByPointer
```

```
bool listFindItemByPointerFunction(void* data)

//ListFindItemByPointer returns first ListItem where ListItem-
>data == data
//can not be run in parallel
ListItem* ListFindItemByPointer(List* list, void* data)

//ListNew creates a new List
List* ListNew(void)

//ListDelete frees all ListItem
//dataFree is called on ListItem->data if it is not NULL
void ListDelete(List* list, void (*dataFree)())
```

- array

```
//ArrayNew creates a new Array
Array* ArrayNew(size_t size)

//ArrayInsert insert element into the last position of Array
void ArrayInsert(Array* array, void* value, size_t size)

//ArrayDelete frees Array
void ArrayDelete(Array* array, void(*dataFree)(void*))
```

- geometry

```
//PositionSame determines if a and b have the same coordinate
bool PositionSame(Position a, Position b)
```

- client

```
//Tick draws new frame
static Uint32 Tick(Uint32 interval, void *param)

//EventKey handles movement key events
static int EventKey(void* data, SDL_Event* sdl_event)

//ClientConnect connects to a server
void ClientConnect(void)

//DrawCharacterFind find CharacterYou
static Character* DrawCharacterFind(WorldClient* worldClient)

//DrawCharacter draws gameend screen
static void DrawGameend(WorldClient* worldClient)
```

```
//DrawCharacter draws exit
static void DrawExit(WorldClient* worldClient, Position offset)

//DrawCharacter draws objects
static void DrawObject(WorldClient* worldClient, Position offset)

//DrawCharacter draws characters
static void DrawCharacter(WorldClient* worldClient, Position offset)

//Draw draws to SDLRenderer
static void Draw(WorldClient* worldClient)

//ClientReceive gets updates from server
//worldCopy is not used after return
void ClientReceive(WorldClient* worldCopy)

//ClientStop loads client module
void ClientStart(void)

//ClientStop unloads client module
void ClientStop(void)
```

- server

```
//WorldGenerate generates default map
static void WorldGenerate(int height, int width)

//CharacterFindFunction is a helper function of CharacterFind
static bool CharacterFindFunction(void* data)

//CharacterFind returns Character for UserServer
//can not be used in parallel
static Character* CharacterFind(UserServer* userServer)

//AuthFind returns UserServer with that auth or NULL if does not
exists
static UserServer* AuthFind(char* auth)

//AuthCreate creates a 26 character long auth key
static char* AuthCreate(void)

//TickCalculateDestroyBomb removes bomb and creates fire in its place
//if object->type != ObjectTypeBomb then nothing happens
static void TickCalculateDestroyBomb(Object* object)

//TickCalculateFireDestroy makes fires destroys all ObjectTypeBox and
all Character in collision
static void TickCalculateFireDestroy(void)

//TickCalculateEnemyKillCollisionDetect is a helper function of
TickCalculateEnemyKill
static bool TickCalculateEnemyKillCollisionDetect(void* this,
```

```
Character* that)

//TickCalculateWin checks if any CharacterTypeUser if in a winning
state and removes them if so
static void TickCalculateWin(void)

//TickCalculateEnemyKill checks if any CharacterTypeUser is colliding
with CharacterTypeEnemy and kills them if so
static void TickCalculateEnemyKill(void)

//TickCalculateEnemyMovement randomly creates a new random direction
for CharacterTypeEnemys
static void TickCalculateEnemyMovement(void)

//TickCalculateDestroy removes items where .destroy == tickCount
//destroy hooks also added here
static void TickCalculateDestroy(void)

//TickCalculateAnimate calculates next texture state from current
static void TickCalculateAnimate(void)

//TickCalculate calculates next state from current
static void TickCalculate(void)

//TickSend sends new world to connected clients
static void TickSend(void)

//Tick calculates new frame, notifies users
Uint32 Tick(Uint32 interval, void *param)

//Save saves worldServer and tickCount into world.save
void Save(void)

//Load loads world.save into worldServer
void Load(void)

//EventKey handles WorldServer saving
static int EventKey(void* data, SDL_Event* sdl_event)

//ServerStart generates world, start accepting connections, starts
ticking
void ServerStart(bool load)

//ServerReceive gets updates from users
//userServerUnsafe is not used after return
void ServerReceive(UserServer* userServerUnsafe)

//ServerStop clears server module
void ServerStop(void)

//ServerConnect register new connection user, returns it with auth
//userServerUnsafe is not used after return
void ServerConnect(UserServer* userServerUnsafe)
```

- network

```
//NetworkServerStop disables incoming requests
void NetworkServerStop(void)

//NetworkServerStart enables incoming requests
void NetworkServerStart(void)

//NetworkClientStop disables incoming requests
void NetworkClientStop(void)

//NetworkClientStart enables incoming requests
void NetworkClientStart(void)

//NetworkSendClient send worldServer to client as WorldClient
void NetworkSendClient(WorldServer* worldServer, UserServer*
userServer)

//NetworkSendServer send userClient to server as UserServer
void NetworkSendServer(UserClient* userClient)

//NetworkConnectServer client request to server to create connection
void NetworkConnectServer(UserClient* userClient)
```

- geometry

```
//Collision tells whether there's a collision between objects at
positions
bool Collision(Position position1, Position position2)

//CollisionObjectSGet returns a List with Objects colliding with this
//collisionDecideObjectFunction decides for each object whether it
should be taking into account
//if collisionDecideObjectFunction is NULL then it's treated as always
true
List* CollisionObjectSGet(List* objectS, Position position, void*
this, Collision

//CollisionCharacterSGet returns a List with Characters colliding with
this
//collisionDecideCharacterFunction decides for each object whether it
should be taking into account
//if collisionDecideCharacterFunction is NULL then it's treated as
always true
List* CollisionCharacterSGet(List* characterS, Position position,
void* this, CollisionDecideCharacterFunction
collisionDecideCharacterFunction)

//CollisionLinePositionGet calculates position taking collision into
account in discrete line (from, to)
```

```
//from must not be equal to to
//we can be NULL
//if collisionDecideObjectFunction is NULL then it's treated as always
true
//if collisionDecideCharacterFunction is NULL then it's treated as
always true
Position CollisionLinePositionGet(
WorldServer* worldServer,
    Position from,
    Position to,
    void* we,
    CollisionDecideObjectFunction collisionDecideObjectFunction,
    CollisionDecideCharacterFunction collisionDecideCharacterFunction
)

//CollisionFreeCountObjectGetRecursion is a helper function of
CollisionFreeCountObjectGet
static int CollisionFreeCountObjectGetRecursion(WorldServer*
worldServer, Position positionCompress)

//CollisionFreeCountObjectGet returns how many square sized object-
free area is reachable from (position - position % squaresize)
int CollisionFreeCountObjectGet(WorldServer* worldServer, Position
position)

//SpawnGet returns a position where there's at least 3 free space
reachable without action so player does not die instantly
Position SpawnGet(WorldServer* worldServer, int
collisionFreeCountObjectMin)
```

- key

```
//KeyMovementCollisionDetectObject is a helper function of KeyMovement
static bool KeyMovementCollisionDetectObject(void* this, Object* that)

//KeyMovementCollisionDetectCharacter is a helper function of
KeyMovement
static bool KeyMovementCollisionDetectCharacter(void* this, Character*
that)

//KeyMovement moves character based on it's pressed keys
void KeyMovement(Character* character, WorldServer* worldServer)

//KeyBombPlace places a bomb to the nearest square in the grid
relative to the character
void KeyBombPlace(Character* character, WorldServer* worldServer, long
long tickCount)

//KeyMovementRandom sets randomly one key to be active
void KeyMovementRandom(Character* character)
```

- SDL

```
//SDLResourceListLoadObject loads resources related to objects into
TextureSSObject
static void SDLResourceListLoadObject(void)

//SDLResourceListLoadCharacter loads resources related to characters
TextureSSCharacter
static void SDLResourceListLoadCharacter(void)

//SDLInit loads SDL modules
void SDLInit(void)

//SDLTextureDelete is a helper function of ArrayDelete
static void SDLTextureDelete(void* texture)

//SDLDestroy unloads SDL modules
void SDLDestroy(void)
```