Peragro

(Free)V1.0 Beta

INTRODUCTION

PERAGRO IS A PLUGIN (DLL) ADDITION / EXTENSION FOR GAMESTUDIO TO MAKE

USE OF A (SOURCE ALTERED VERSION) OF THE RECAST/DETOUR NAVIGATIONAL MESH GENERATION AND PATH FINDING SYSTEM

THE PLUGIN ALSO MAKES USE OF IRRLICHT LIBRARY AND MDL SDK FOR SOME MDL FILE FUNCTIONALITY

THREAD WAITING FUNCTIONS… PERAGRO USES WORKER THREADS TO TAKE THE WORKLOAD OFF OF GAMESTUDIO AND TO PREVENT SUDDEN FREEZES , ONLY ONE THREAD IS ALLOWED TO RUN AT ANY GIVEN TIME! YOU MUST WAIT FOR THE THREAD TO COMPLETE BEFORE CONTINUING WITH FURTHER ACTIONS ,IF YOU DON'T IT WILL RESULT IN INCOMPLETE/NON EXISTING DATA ! IT IS VERY IMPORTANT THAT YOU UNDERSTAND THE THREAD WAITING USAGE

FOR ERROR FREE OPERATION , YOU COULD USE WAIT(FRAME AMOUNT) TO WAIT FOR THE THREAD BUT IT IS NOT SAFE AND EVERY PC WILL NOT PERFORM THE SAME SO JUST USE THIS FUNCTION AS INSTRUCTED!!!

```
int peragro_thread (void* thandle);
```

check a thread for completion status , returns 0 for completion and 1 for busy.

thandle: handle of the thread to check on

example:

```
peragro_thr my_thread=peragro_add_all_tiles ();
while(peragro_thread(my_thread)==1)//busy
{
	//display a busy icon here
	wait(1);//wait while busy so as not to freeze;
}
```

Continue safely with other operations

Peragro

PERAGRO OBJECT FUNCTIONS

```
void peragro_create  ();
```

Creates the main object.

This object has to be created to use peragro!!

```
int peragro_valid ();
```

Check if the peragro object is valid and existing

Returns 1 when valid and 0 when invalid

```
void peragro_destroy ();
```

Destroy the peragro object and clean up

```
peragro_thr peragro_init (custombuild* step);
```

initializes the build process from the custom build values and returns handle to the thread used ,and builds the complete

navmesh if buildalltiles were specified as 1 ,or passes building the complete navmesh when buildalltiles were specified as 0,

note that you must atleast have some input geometry in the build list before this function is used ,you may remove the geometry again if its unneeded after waiting for the thread completion all functionality is available , use peragro_thread to wait for the thread

PERAGRO BUILD SETUP FUNCTIONS

Peragro needs to pass filtering/build values into the recast process therefore

A predefined struct along with functions for this was created.

Mostly these input values are based on the idea of an agent/character wich has properties

Of height,radius ,normaly this would be the biggest agent in the game !

Peragro passes these values into recast and recast uses them to filter out non walkable areas

Of the input geometry ,leaving the coder/user with a mesh wich only has walkable areas

Therefore any input geometry passed into the process will be filtered and when pathfinding

Is done on the final output mesh ,the agent should not get stuck or walk into objects ,well

That's the idea…

Peragro

PERAGRO BUILD SETUP FUNCTIONS

```
custombuild* cbuild_create ();
```

Creates a build object ,fills it with default values ,and returns a pointer to the object.

You need to create this object to pass build values into peragro.

The default values are as follows:

```
CellSize        =0.3f;
CellHeight      =0.2f;
AgentHeight     =2.0f;
AgentRadius     =0.6f;
AgentMaxClimb   =0.9f;
AgentMaxSlope   =45.0f;
RegionMinSize   =8;
RegionMergeSize =20;
EdgeMaxLen      =12.0f;
EdgeMaxError    =1.3f;
VertsPerPoly    =6.0f;
DetailSampleDist=6.0f;
monotonePartitioning =0;
DetailSampleMaxError =1.0f;
TileSize =32;
Buildalltiles =1;
```

Please note that the default values are very small and will result

In to much of a high detailed build in gamestudio

Therefore just some are usefull when remained at default while others

Will be changed for speed / needs / recommended values

```
void  cbuild_save   (char* filename , custombuild *step);
```

Saves a build object to file

This allows you to save build values to file so that they can be loaded when needed.

```
custombuild* cbuild_load   (char* filename);
```

Loads the saved file and creates a build object filled with the saved values

```
void cbuild_remove (custombuild* mybuild);
```

Destroy the build object

Peralgro

MIKKO'S EXPLANATION OF BUILD SETUP VALUES

(MIKKO IS THE AUTHOR OF RECAST)

Mikko Mononen memon@inside.org

TileSize

The width/height size of tile's . [Limit: >= 0]

This field is only used when building multi-tile meshes.

CellSize

The cell size to use for fields. [Limit: > 0]

cellsize and cellheight define voxel/grid/cell size.

So their values have significant side effects on all parameters defined in voxel units.

The minimum value for this parameter depends on the platform's floating point accuracy,

with the practical minimum usually around 0.05.

CellHeight

The cell height to use for fields. [Limit: > 0]

cellsize and cellheight define voxel/grid/cell size.

So their values have significant side effects on all parameters defined in voxel units.

The minimum value for this parameter depends on the platform's floating point accuracy,

with the practical minimum usually around 0.05.

AgentHeight

Minimum floor to 'ceiling' height that will still allow the floor area to be considered walkable.

Limit: >= 3]

Permits detection of overhangs in the source geometry that make the geometry below un-walkable.

The value is usually set to the maximum agent height.

Paragiro

MIKKO'S EXPLANATION OF BUILD SETUP VALUES

(MIKKO IS THE AUTHOR OF RECAST)

Mikko Mononen memon@inside.org

AgentRadius

The distance to erode/shrink the walkable area of the heightfield away from obstructions.

Limit: >=0]

In general, this is the closest any part of the final mesh should get to an obstruction in the source geometry.

It is usually set to the maximum agent radius.

While a value of zero is legal, it is not recommended and can result in odd edge case issues.

AgentMaxSlope

The maximum slope that is considered walkable. [Limits: 0 <= value < 90] [Units: Degrees].

The practical upper limit for this parameter is usually around 85 degrees.

AgentMaxClimb

Maximum ledge height that is considered to still be traversable. [Limit: >=0]

Allows the mesh to flow over low lying obstructions such as curbs and up/down stairways.

The value is usually set to how far up/down an agent can step.

RegionMergeSize

Any regions with a span count smaller than this value will, if possible, be merged with larger regions.

Limit: >=0]

EdgeMaxLen

The maximum allowed length for contour edges along the border of the mesh. [Limit: >=0]

Extra vertices will be inserted as needed to keep contour edges below this length.

A value of zero effectively disables this feature.

Peragro

MIKKO'S EXPLANATION OF BUILD SETUP VALUES

(MIKKO IS THE AUTHOR OF RECAST)

Mikko Mononen memon@inside.org

EdgeMaxError

The maximum distance a simplfied contour's border edges should deviate the original raw contour.

Limit: >=0]

VertsPerPoly

The maximum number of vertices allowed for polygons generated during the contour to polygon conversion process.

Limit: >= 3 <=6]

DetailSampleDist

Sets the sampling distance to use when generating the detail mesh.

(For height detail only.) [Limits: 0 or >= 0.9]

monotonePartitioning

Buildalltiles

This value when specified as 1 will build the complete whole navigational mesh when

PERAGRO is initialized.

When specified as 0(zero) the build process will be skipped at initialization which

Is usefull to load saved tiles that has been previously build..

Building can be a slow process therefore when a saved file is rather loaded it is much

Faster and recommended for the game process to rather Pre build and save the files out in development time.

Then load the saved files only in game time

Peragro

RECOMMENDED VALUES ,BASED ON CHARACTER

Pre-define this values

float character_radius

float character_height

(the radius and height of the biggest agent in the game, values might not correspond to engine units..)

then for the build values use the following:

```
CellSize=character_radius/2;

CellHeight=mybuild.CellSize/2;

AgentHeight=peragro_ceiling(character_height/mybuild.CellHeight);

AgentRadius=peragro_ceiling(character_radius/mybuild.CellSize);

AgentMaxClimb=peragro_ceiling((character_height/2)/mybuild.CellHeight);

EdgeMaxLen= AgentRadius*8;(custombuid.agentradius)

Optional -EdgeMaxLen=0;
```

Note that you don't have to strictly follow this ,you may alter values to allow for faster Builds , cellsize has a major impact on speed ,the bigger cellsize the faster but you may Lose detail , if you use one tile only the build may fail if cellsize is to large for The tile ..

Cellheight ,you may want to decrease this height so that the agent can easily step onto objects/terrain mounts..

Note that Some of these values are interweaved so altering one may effect another..

HELPER.C has a predefined build setup function to alter or use as you need

Peragro

INPUT GEOMETRY FUNCTIONS

```
void mesh_from_level_blocks(ENTITY* level , int chunks, int _lod);
```

This function creates mesh objects from level blocks and adds the separate meshes to the build list.

Level:level_entity(the level entity).

Chunks : the amount of blocks to parse starting at 0. Use ent_status(level_ent,16) for all blocks

Lod: level of detail. Use 0 for default.

```
void* mesh_from_level_blocks_chunk (ENTITY* level , int chunks, int _lod);
```

This function creates a mesh object from a single level block and adds the mesh to the build list

And returns the mesh object;

Level:level_entity(the level entity).

Chunks : the index of block to parse starting at 0. Use a loop to iterate for all blocks

Lod: level of detail. Use 0 for default.

```
void trans_mesh_ent(void* mesh , ENTITY* ent);
```

This function transforms a mesh object to the rotation,position,scale of an entity

mesh:the mesh object to transform.

ent: the entity's values will be used to match the mesh to the entity rotation,scale,position

you canot transform the mesh that's already in the list ,for that you have to remove it from the list and transform it and then re enter it into the list and update the tiles

```
void trans_mesh_arg(void* mesh , ANGLE* angle , VECTOR* scale , VECTOR* pos);
```

This function transforms a mesh object to the rotation,position,scale given by the parameter list

mesh:the mesh object to transform.

angle:the rotation angles ,pan,tilt,roll

scale: the scale vector.

pos: the position vector.

```
void* mesh_from_ent(ENTITY* ent, int _lod);
```

This function creates a mesh object from the entity ,transforms the mesh ,adds it to the build list and returns the mesh object.

ent: the source entity.

lod: level of detail ,use 0 for default.

Peregro

INPUT GEOMETRY FUNCTIONS

```
void mesh_from_hmp(ENTITY* ent , int chunks, int _lod);
```

This function creates mesh objects from the terrain entity and adds each chunk as a separate mesh to the build list.

ent: the source terrain entity.

chunks: the amount of chunks to add starting at 0, use ent_status(terrain entity,16) for all chunks.

lod: level of detail ,use 0 for default.

```
void* mesh_from_hmp_chunk (ENTITY* ent , int chunk, int _lod);
```

This function creates a mesh object from the terrain entity chunk and adds the chunk as a single mesh to the build list.

ent: the source terrain entity.

chunks: the chunk's index to add starting at 0, use loop iteration for all chunks.

lod: level of detail ,use 0 for default.

```
void* mesh_from_obj_file (char* filename);
```

This function creates a mesh object from the wavefront obj file and adds the mesh to the build list and returns the mesh object.

filename: the source obj file.

```
void* mesh_from_buffers(D3DVERTEX* vbuffer , short* tbuffer , int verts , int tris);
```

This function creates a mesh object from buffers and adds the mesh to the build list and returns the mesh object.

vbuffer: the source vertex buffer.

tbuffer: the source triangle buffer.

verts: the amount of vertices in the vertex buffer.

tris: the amount of triangles in the triangle buffer.

```
int get_vertcount (void* mesh);
```

This function returns the mesh object's amount of vertices;

mesh: the source mesh object.

Peragro

INPUT GEOMETRY FUNCTIONS

```
int get_tricount  (void* mesh);
```

This function returns the mesh object's amount of triangles;

mesh: the source mesh object.

```
int*   get_tbuffer (void* mesh);
```

This function returns the mesh object's triangle buffer;

The returned triangle buffer follows the following index rules:

```
int *tribuffer=get_tbuffer(mesh);

int vertex1;int vertex2;int vertex3;

vertex1=tribuffer[triangle_index*3]; //(index of vertex used by triangle point 1)

vertex2=tribuffer[triangle_index*3+1]; //(index of vertex used by triangle point 2)

vertex3=tribuffer[triangle_index*3+2]; //(index of vertex used by triangle point 3)
```

mesh: the source mesh object.

```
float* get_vbuffer (void* mesh);
```

This function returns the mesh object's vertice buffer, The returned vertice buffer follows the following index rules:

```
float *vertbuffer=get_vbuffer(mesh);

vertex_x =vertbuffer[vertex_index*3]; //(x value of vertex)

vertex_y =vertbuffer[vertex_index*3+1]; //(y value of vertex)

vertex_z =vertbuffer[vertex_index*3+2]; //(z value of vertex)
```

or

```
int *tribuffer=get_tbuffer(mesh);

int vertex1;int vertex2;int vertex3;

vertex1=tribuffer[triangle_index*3]; //(index of vertex used by triangle point 1)

vertex2=tribuffer[triangle_index*3+1]; //(index of vertex used by triangle point 2)

vertex3=tribuffer[triangle_index*3+2]; //(index of vertex used by triangle point 3)

float *vertbuffer=get_vbuffer(mesh);

vertex_x =vertbuffer[vertex1*3]; //(x value of vertex)

vertex_y =vertbuffer[vertex2*3+1]; //(y value of vertex)

vertex_z =vertbuffer[vertex3*3+2]; //(z value of vertex)
```

mesh: the source mesh object.

Peregrine

INPUT GEOMETRY FUNCTIONS

```
void save_mesh(void* mesh , char* filename);
```

This function saves the mesh object to file,the file can be imported into a modelling application as a wavefront obj file, It saves just pure geometry.

mesh: the source mesh object.

filename: name of the file to save eg. "mymesh.msh","mymesh.obj" there is no restriction on file

Extension/name except normal windows file extension/naming convention rules apply..

```
void* load_mesh(char* filename);
```

This function loads the mesh object from file,it can only load files saved by save_mesh ,it then creates a mesh object and returns that mesh object.

filename: name of the file to load eg. "mymesh.msh","mymesh.obj" there is no restriction on file

Extension/name except normal windows file extension/naming convention rules apply..

Although the extension may be of any kind like .obj please remember this is NOT a wavefront

Obj file loader !!

```
void save_mdl(char* filename , void* mesh , int swapindices);
```

This function saves the mesh object to a mdl file,the file can be imported into a modelling application/gamestudio/wed/med as a mdl file, It saves just pure geometry.

mesh: the source mesh object.

filename: name of the file to save eg. "mymesh.mdl" there is no restriction on file

Extension/name except normal windows file extension/naming convention rules apply..

swapindices: specify 1 to turn triangle facing direction or 0 to do nothing

```
void del_mesh   (void* mesh);
```

This function deletes the mesh object ,usefull for cleaning up.

mesh: the source mesh object to delete.

Peragro

EXPOSED (INTERNAL) BUILD LIST FUNCTIONS

PERAGRO MANAGES AN INTERNAL BUILD LIST WICH IS A COLLECTION OF ALL THE INPUT GEOMETRY

TO BE USED IN THE RECAST BUILD PROCESS, BY ADDING/REMOVING FROM THIS LIST IT IS POSSIBLE

TO SUPPORT DYNAMIC FUNCTIONALITY , FOR INSTANCE YOU MAY WANT TO ADD A BUILDING DURING GAME

TIME AND WANT THE NAVIGATIONAL MESH/PATH FINDING TO INCLUDE THIS BUILDING IN ITS PROCESS ,YOU

THEN ADD THE GEOMETRY BY THE INPUT GEOMETRY FUNCTIONS TO THE LIST AND BUILD THE AREA WHERE

THE GEOMETRY WAS ADDED ,OR YOU REMOVE THE GEOMETRY FROM THE LIST AND UPDATE THE AREA BY RE-ADDING THE TILES OF THE AFFECTED AREA. NOTE PERAGRO SUPPORTS SAVING AND LOADING TILES FOR INSTANT REPLACEMENT OF TILES INSTEAD OF BUILDING WHERE IT WOULD SUFFICE..

```
void clear_meshlist ();
```

This function clears all geometry from the build list ,usefull for cleaning up or removing all geometry.

```
void add_meshlist (void* mesh);
```

manualy add a mesh object to the build list, usefull for adding objects during game time.

Note : adding geometry needs an update of tiles in that area before it becomes apart of

The pathfinding/navigational mesh system.

```
void rem_meshlist (void* mesh);
```

manualy remove a mesh object from the build list, usefull for removing objects during game time.

Note : removing geometry needs an update of tiles in that area before it takes effect of

The pathfinding/navigational mesh system.

```
void  rem_meshlist_ent  (ENTITY* ent , int chunk);
```

manualy remove a mesh object from the build list, usefull for removing objects during game time.

Note : removing geometry needs an update of tiles in that area before it takes affect of

The pathfinding/navigational mesh system.

ent: the entity that was used as the source input.

chunk: the chunk that was used as the source input, -1 removes all mesh objects(chunks) of the source entity in the list ,use 0 for non chunked entites and meshes that were loaded by obj files

Peragro

EXPOSED (INTERNAL) BUILD LIST FUNCTIONS

```
void* get_meshlist_ent (ENTITY* ent , int chunk);
```

retrieve a mesh object from the build list.

Ent: the source entity that was used.

Chunk: the source chunk that was used, if it were an unchunked entity use 0

TILE MANIPULATION FUNCTIONS.. THE RESULTING NAVIGATIONAL MESH IS COMPOSED OF TILES WHICH ARE OF TILESIZE OF THE INPUT BUILD VALUES ,THEY CAN BE REMOVED AND ADDED,SAVED/LOADED

```
void peragro_rem_tile_id (int _tileid);
```

removes a tile from the navmesh by its id;

_tileid: the id of the tile to remove.

Note that if a tile is removed ,that area would be no longer apart of the navmesh therefor no Pathfinding will not occur through it anymore.

```
void peragro_rem_tile_pos(VECTOR* pos);
```

removes a tile from the navmesh from a vector position;

pos:source position.

Note that if a tile is removed ,that area would be no longer apart of the navmesh therefor no Pathfinding will not occur through it anymore.

```
void peragro_rem_tiles_box(VECTOR* bmin , VECTOR* bmax);
```

removes tiles from the navmesh touching a box + tiles surrounding the area to make sure the tiles are removed that had geometry affecting the area .

you can use peragro_calc_box for calculating an unrotated box from an entity.

bmin: the box min vector.

Bmax: the box max vector.

Note that if a tile is removed ,that area would be no longer apart of the navmesh therefor no Pathfinding will not occur through it anymore.

Peragro

TILE MANIPULATION FUNCTIONS

```
void peragro_rem_tiles_ent_box (ENTITY *ent , int chunk);
```

removes tiles from the navmesh touching a box of an entity + tiles surrounding the area to make sure the tiles are removed that had geometry affecting the area .

ent: the entity to use for the box calculation.

chunk: the chunk to use for the box calculation. Use 0 for unchunked entities

Note that if a tile is removed ,that area would be no longer apart of the navmesh therefor no

Pathfinding will not occur through it anymore.

```
peragro_thr peragro_add_all_tiles ();
```

builds all tiles ,and returns the thread handle, you must wait for the thread to complete

before doing anything else ,if you don't you will end up with unfinished/incomplete data.

You may only use a SINGLE thread at any time!!.

To wait for the thread use:`int peragro_thread (void* thandle);`

example :

```
peragro_thr my_build_all=peragro_add_all_tiles();
```

or

```
void* my_build_all=peragro_add_all_tiles();


while(peragro_thread(my_build_all)==1)
{
        //display busy icon
        Wait(1);
}
//safe to continue with normal functionality
```

```
peragro_thr peragro_add_tile(VECTOR* pos);
```

builds tile from a vector ,and returns the thread handle, you must wait for the thread to complete.

pos:the source vector to use .

Peragro

TILE MANIPULATION FUNCTIONS

peragro_thr peragro_add_tiles_box (VECTOR* bmin , VECTOR* bmax);

builds tiles touching a box area+extra for making sure a complete geometry affected area is used ,and returns the thread handle, you must wait for the thread to complete.

use peragro_calc_box to calculate an un rotated box from an entity.

bmin:the box min vector.

bmax: the box max vector.

peragro_thr peragro_add_tiles_ent_box (ENTITY *ent , int chunk);

builds tiles touching a box from an entity ,area+extra tiles are added for making sure a complete geometry affected area is used ,and returns the thread handle, you must wait for the thread to complete.

ent:entity to use for box calculation.

chunk: chunk to use ,use 0 for unchunked entities.

void peragro_save_all_tiles(char* filename);

saves the complete navigational mesh (tiles)in its current state out to a file,very fast!

filename: target file to save to eg “mylevel1.til”.

void peragro_load_all_tiles (char* filename);

loads the complete navigational mesh (tiles) from a file, any existing navmesh tiles/data will be replaced.. very fast!

Use load tiles to quickly load pre build navmesh data instead of building everything at game time

filename: source file to load from eg. “mylevel1.til”.

void peragro_save_tile (char* filename , int tileid);

saves single tile of navmesh in current state out to file by its tileid, very fast!

filename: file to save to eg. “mylevel1building1.til”.

tileid: the id of the tile to save;

Peragro

TILE MANIPULATION FUNCTIONS

```
void peragro_load_tile (char* filename);
```

load single tile from file and replace the tile with the same id in the current navmesh

very fast!

filename: file to load eg. “mylevel1building1.til”.

```
int peragro_tile_id(VECTOR* pos);
```

retrieve the id of a tile traced into with pos vector .

pos: source position vector to find the tile from.

TILE MANIPULATION LIST FUNCTIONS.. PERAGRO MANAGES INTERNAL LISTS FOR USE IN ADDING/REMOVING TILES FROM MULTIPLE LOCATIONS ALL AT ONCE WITH ONE THREAD.

```
void clear_temp_entlist();
```

clears the internal entity box list

```
void clear_temp_pntlist ();
```

clears the internal point(vector positions) list

```
void rem_temp_entlist (ENTITY* ent , int chunk);
void add_temp_entlist (ENTITY* ent , int chunk);
```

adds/removes from/to the internal entity box list

ent: the entity to use for box calculation.

Chunk: the chunk to use ,use 0 for unchunked entities;

```
void rem_temp_pntlist (VECTOR* pnt , int id);
void add_temp_pntlist (VECTOR* pnt , int id);
```

adds/removes from/to the internal point(vector position list)

pnt: vector to use to find tiles affected.

Id: user id to associate with the pnt vector.

Peragro

TILE MANIPULATION LIST FUNCTIONS

```
peragro_thr peragro_add_tiles_box_entlist();
```

adds tiles touching all the boxes calculated from the entities in the internal list and returns the handle to the thread ,you must wait for the thread to complete.

```
peragro_thr peragro_add_tiles_pntlist ();
```

adds tiles from all the vector positions in the internal list and returns the handle to the thread ,you must wait for the thread to complete.

PATHFINDING FUNCTIONS

```
VECTOR* peragro_findpath (VECTOR *from , VECTOR* to , int *num_points);
```

Finds a path from “from” vector to “to” vector and returns a array filled by “num_points” of path points.

Example:

```
VECTOR from;VECTOR to;

vec_set(from,my.x);

vec_set(to,you.x);

int num_points;

VECTOR* path=peragro_findpath (from ,to ,num_points);

//path will be an array of path[num_points], starting at 0
```

Peragro

DEBUG/HELPER FUNCTIONS

```
VECTOR* peragro_rnd_pnts(int points);
VECTOR* peragro_rnd_pnt_radius(VECTOR* from , float radius);
```

Creates random points on the navmesh to use for spawning or pathfinding and returns the array

Of points .

```
VECTOR* peragro_nearest_pnt(VECTOR* from);
```

Finds the nearest point on the navmesh from the "from" vector .returns a single vector from the result.

From: position to find nearest position on navmesh from.

Points: number of points to create ,return will be VECTOR* my_points[points].

Radius: radius from the "from" position to create random points in.

```
float peragro_ceiling(float val);
```

Calculates the ceiling of a value.

```
void  peragro_update_bounds();
```

recalculates the internal mesh bounds,usefull to manualy update bounds if meshes were added

```
void  peragro_calc_box(ENTITY *ent , int chunk , var *_min , var *_max);
```

calculates an unrotated box from the geometry of the entity chunk.

ent:the source entity.

chunk: the source chunk ,use 0 for non chunked entities

_min: the returned box min values min[0]=x,min[1]=y,min[2]=z;

_max: the returned box max values max[0]=x,max[1]=y,max[2]=z;

Usage :

```
var min[3];var max[3];
peragro_calc_box(level_ent,10,min,max);
```

```
void  peragro_world_box(var *_min , var *_max);
```

calculates the navmesh world box

Peragro

DEBUG/HELPER FUNCTIONS

```
int  peragro_tot_tiles  ();
```

total tiles in the current navmesh

```
void peragro_tile_box  (int tileid , var *_min , var *_max);
```

return the bounding box of tile id . min[0]=x,min[1]=y;min[2]=z same for max

usage:

```
var min[3];var max[3];
peragro_tile_box(0 , min max);
//min and max now contain the box values of the tile 0
```

```
int   peragro_in_tmp_list();
void  peragro_init_tiledata (VECTOR *bmin , VECTOR *bmax);
void  peragro_mark_box (VECTOR *bmin , VECTOR *bmax);
void* mesh_from_tmp_list(int index ,var *_min , var* _max , int *tileid);
```

used inside of HELPER.C for debug draw functions.

peragro_init_tiledata stores the tile data within the box area to an internal list

if you made a change to the navmesh you have to re-update the date by re_calling this

function.

peragro_mark_box finds tiles in the internal list that are within the bounds and then stores

Them in another internal list indexed from 0 to the value of peragro_in_tmp_list

Mesh_from_tmp_list returns the data stored in the marked_box list for display