GMnet Manual

Release

Parakoopa & GMnet Developers

Sep 23, 2017

GMnet ENGINE Manual

Here you can find the manuals for all GMnet products.

CHAPTER 1

Getting Started

GMnet ENGINE is a multiplayer engine for Game Maker: Studio by YoYoGames. It works by syncing instance variables to other players and has many other features. If you want to get started and learn how to use GMnet ENGINE, start with the *Tutorial*.

Note: GMnet ENGINE is bundled with GMnet PUNCH. All functionality is implemented in GMnet ENGINE. You don't need to read the PUNCH manual. If you want to use GMnet PUNCH with your own multiplayer engine, please continue reading the *GMnet PUNCH manual* instead.

CHAPTER 2

Tutorial

What is GMnet ENGINE?

GMnet ENGINE allows you to make multiplayer games in *Game Maker: Studio* with nearly zero knowledge in networking.

Everything in GMnet ENGINE is done via the key concept of *Game Maker*: **Objects and Instances**.

You simply tell the engine what variables of what object should be synced and how. The engine takes care of sending and recieving the data and the whole client and server architecture.

This means your game's network functions are easy to use and easy to maintain.

So, let's waste no time, and let's get started with making your very first game with **GMnet ENGINE**.

Basic configuration

In this tutorial we want to make a **simple platformer** with some neat little extras to demonstrate what *GMnet ENGINE* can do. Our platformer will have:

- Walls/Players/Physics (not Box2D) (obviously)
- Multiple rooms
- An overlay that shows connected players
- A night and day cycle
- A basic chat system

Adding the engine to a project

Before we can get started, add the asset to an empty project. Depending on what you want to do, you have three options.

- If you want to **follow this tutorial yourself**, you should add **ALL scripts and sprites and the object** `obj_htme` to your project.
- If you want to just **read through this tutorial** and test the demo game, **simply add everything**.
- When starting completely blank or when you want to **add the engine to your own game**, add all folders named **htme** and **udphp** but not the *htme_demo* folders.

Configuration

Great! No matter what option you chose, you propably want to take a look at the initiation and configuration, so let me walk you through.

Before we look at the configuration we check the initiation script. **The init can be found in ‘‘scripts/htme/htme_init‘‘**.

The first thing you'll find under initiation is `randomize();`. That assures that the random operations Game Maker does are truly random, which is important for the engine. This is good to know when you make your game. Thats all for the initiation for now. Let's go to configuration.

The configuration can be found in ‘‘scripts/htme/htme_config‘‘.

First you see **debug level**. Valid options can be found in the enum `htme_debug` and how debugging works is explained in the comment. Leave it on the default option for now.

The next thing is **gmversionpick**. Depending on what version of game maker you are using you may need to change this. If you use above 1.4.1567 set this value to 1. If you use 1.4.1567 or below set this value to 3. If you set it to 3 you need to go to `htme_serverStart` and comment this line:

```
switch (gmversionpick)
{
    // You maybe dont got network_create_socket_ext just add // in front of it
    //case 1: self.socketOrServer = network_create_socket_ext(network_socket_udp,
↪port); break;
    case 2: self.socketOrServer = network_create_socket(network_socket_udp); break;
    case 3: self.socketOrServer = network_create_server(network_socket_udp,port,
↪maxclients); break;
    default: htme_error_message_handler("Go to script: htme_serverStart and decomment␣
↪the one you use!");
}
```

This is because if you use 1.4.1567 or below you dont got the **network_create_socket_ext** function.

Let's skip ahead to **‘‘self.global_timeout‘‘**. That's actually the only important configuration option and even that can stay on default if you want. Timeout specifies after how many steps of inactivity the connection between client and server will die and you will be disconnected. The default is `5*room_speed` which basicly means 5 seconds. When client and server don't communicate to each other for 5 seconds, they are considered to be disconnected.

So yeah that was all there is to configure for now.

Follow the next steps of the configuration. They will explain how GMnet ENGINE (or GMnet PUNCH to be precise) enables you to connect to other players even behind a firewall.

Setup GMnet PUNCH

Note: This topic requires GMnet PUNCH enabled. More information below.

If you want *GMnet ENGINE* can be used to connect players **even if the do not forward ports in their firewall**.

To understand this, you need to know, that whenever you want to play a game with someone there are three options:

- Dedicated servers to play on, like most shooters have
- Forwarding a port on the servers firewall, most common when playing strategy games or hosting an own dedicated server of some game
- "Hole-Punching" - **GMnet PUNCH**, which ships with GMnet ENGINE version, allows UDP Hole Punching. This technique allows two players to connect to each other even when no ports are forwarded at all. This works most of the time, however not always. Especially in mobile networks or buisness networks that might fail. Hole Punching is often used by many games where multiplayer should be easy to do by the user.

That means if you *disable GMnet PUNCH*, your players either can't play together online, only locally, or the server has to open ports.

GMnet PUNCH configuration

In `htme_config` there are several variables you need to change to use *GMnet PUNCH*.

`self.use_udphp`: This is the most obvious one. Set this to **true**, to enable GMnet PUNCH.

`self.udphp_master_ip` and `self.udphp_master_port`: You need to set these to the ip and the port of your mediation server. What a mediation server is, is explained in the next section. You basically need to host a server, to which server and clients connect to share their information.

`self.udphp_rctintv`: How often (in steps) the server should reconnect to the mediation server to make sure it is still connected. It is important that the server is connected to the mediation server, so don't set this too high. The default option should be fine.

If you don't have a server yet you can use the ip 95.85.63.183 and the port 6510 for testing only.

Hosting a master server

Note: This topic requires GMnet PUNCH enabled.

Note: GMnet GATE PUNCH is outdated and will be replaced by a new master server in the next version.

To use the magic of *GMnet PUNCH* you need to host a mediation/master/rendevouz server. The server is a small application written in Java that needs to be run on a server **with a forwarded/open port**.

To explain why you need this, let me explain how the hole punching works:

- The server connects with the master server (GMnet GATE.PUNCH) and its port and ip get registered.
- When the clients want to connect to this server, his port and ip also get registered and sent to the server. And the client gets the ip and port of the server.
- Now both send UDP packets to each other at the same time: The hole is punched.

We need the master server (GMnet GATE.PUNCH) because we need to make the server aware of the fact that a client wants to connect.

The download can be found here:

http://gmnet.parakoopa.de/gatepunch

The server is very basic, feel free to adjust it to your needs.

Currently the server binary listens on **port 6510** and there to change it if you don't want to compile the server yourself, but we will add that option in the future.

To start it, run: `java -jar gmnet_gate_punch.jar` from the folder you saved the server in. Under Windows you might need to replace `java` with the full installation path of java. Under linux I recommend `screen` to run it in the background on a server.

Hosting

For cheap hosting I personally recommend **digitalocean**. Server start at 0.0015 per hour (5 per month) and you can cancel at every time.

Google yourself a promo code and you might get two months for free. There are also tutorial on how to set up java on these servers. They are basically linux servers with complete access.

If you want to use digitalocean, please use this code to register: https://www.digitalocean.com/?refcode=1bff6d6dff1a

Some tutorials

If you want to know how to login to the server: How To Create Your First DigitalOcean Droplet Virtual Server

For some basic commands and stuff: An Introduction to Linux Basics

If you want to install java: How To Install Java on Ubuntu with Apt-Get

For transfering the master server: How To Use Filezilla to Transfer and Manage Files Securely on your VPS

[You can skip the stuff with the keyfile and everything. Simply install filezilla and connect to sftp://yourip with your username and password]

When you are done simply execute `java -jar gmnet_gate_punch.jar`

As I said, I recommend install screen, it allows you to run commands even if you are not logged in, running in the background and you can just switch to them at any time: How to Install and Use Screen on an Ubuntu Cloud Server

You can also create an autostart service for the server and run it on startup.

Starting the engine

Great! Now that we are set up, let's actually make a game.

Before we start, if you want to follow the tutorial make sure again, that you added all sprites, all scripts and the object `obj_htme` to an empty project.

The menu

Let's create a very basic menu.

First create a **new room** called **htme_rom_menu**. For this tutorial we are keeping things very basic and we are not using any views. Set the **background color** to be **black** and the dimensions of the room to be **800x600px**.

Add `obj_htme` to the room. This object should be created in the first room of every game you create. It is persistent and will exist even when you change the rooms.

Now **create** an object called `htme_obj_menu`. This will control our basic menu. Add it to the room we just created.

In the **Draw-Event** put the following code, that will simply display a message on how to proceed when starting the game:

```
draw_text(20,20,"Press N for new server and B to connect.");
```

Add two more events to it. One **B-key** event and a **N-key** event. B will join a server and N create a new one.

Starting the server

In the **N-key** event add the following code:

```
///START SERVER

//Ask player for port
var port = real(get_string("On which port should the server listen?","6510"));

//Setup server, on success start game, on failure end the game.
if (htme_serverStart(port,32)) {
    room_goto(htme_rom_demo);
} else {
    show_message("Could not start server! Check your network configuration!");
    game_end();
}
```

This will first **ask the player** on which **port** the server should run (of course you can also decide to specify a fixed port number).

After that it **starts the server** using `htme_serverStart(port,maxplayers)` on that port and allows a maximum of **32 players**. It checks if the server is running, and if it is, it **goes to the game room** that we will create in a second.

Starting the client

In the **B-key** event add the following code:

```
///CONECT TO A SERVER

//Ask player for ip & port
var ip = get_string("To which server should we connect?","127.0.0.1");
var port = real(get_string("And on which port is the server running?","6510"));

//Setup client, on success go to waiting room, otherwise end game
if (htme_clientStart(ip, port)) {
    room_goto(htme_rom_connecting); //NOTE THAT WE ARE GOING TO ANOTHER ROOM HERE␣
↪THAN THE SERVER ABOVE
} else {
    show_message("Could not start client! Check your network configuration!");
    game_end();
}
```

Now that is slightly more complicated. It **asks for both ip and port** and then tries to connect using `htme_clientStart(ip, port)`.

Instead of going to the game room when we are done, we actually want **the client to go to** `htme_rom_connecting` which is a room we are now going to create. Why? This room will **display a message** saying “Connecting...” and there will be an object in it that **waits for the client to be connected**. After that we want to go to the game room.

The waiting room `htme_rom_connecting`

Create a room called `htme_rom_connecting` that is just like the first one (but without the objects in it). Now **create the object** `htme_obj_waitforclient` and **place it** into the room.

In the **Draw event** put the following code:

```
draw_text(20,20,"Connecting...");
```

This will display the message “Connecting...” after the player decided to start the client.

In the **Step-Event** put the following:

```
///Check if client is connected
if (htme_clientIsConnected()) {
    room_goto(htme_rom_demo);
}
if (htme_clientConnectionFailed()) {
    show_message("Connection with server failed!");
    game_restart();
}
```

The first statement used the `htme_clientIsConnected()` function to check if the client is... well connected. If so, **we can finally go to the game room**.

If the global timeout has been reached, the engine gives up to conect. In this case the second if-Statement is activated which uses `htme_clientConnectionFailed()` to check if the connection failed. In our example it will simply restart the game.

This waiting room is just an example and you can use different techniques to display a waiting... message.

Connection done!

Theoretically, we could now play the game. Of course, there is no game. The game will crash if we try to start a server because our game room `htme_rom_demo` does not exist.

For now, **create an empty room** just like you did before and call it `htme_rom_demo`.

Now, when you **fire up your game twice**, you should be able to **start a server** on one game, which should **lead you to the empty room**.

On the **other game** you should be able to **connect to this server** by connecting to the ip **127.0.0.1**. If it was successfull, the “Connecting...” message should disappear, and you’ll **find yourself in the same empty room as the server**. **Yay!**

Setting up the basic platformer

We now need to create a simple platformer. Since **this is not supposed to be a tutorial on platformers**, and the platformer we are going to use is simple and pretty bad, simply follow the instructions and copy the code.

1. Setup our game room

Give the game room `htme_rom_demo` we created a nice **background color** and the dimensions **800x600**. For this demo, we are not going to use views.

2. Create a wall

Create a solid object, called `htme_obj_wall`. Give it the sprite `htme_spr_wall`. This is our new wall. Place it into the game room, build some walls, a floor and maybe some platforms.

3. Create the player object

Next we are going to create our player.

Create the object `htme_obj_player` and give it the sprite `htme_spr_player`.

Into the **create-Event** put the following code:

```
///Setup basic stuff for the demo platformer.
self.pressed_jump = false;
self.pressed_left = false;
self.pressed_right = false;
self.name = "";

/** Totro generates random names and is not part of the main engine, it's
  * another marketplace asset by me :)
  */
var ttr = totro(5,7,1);
self.name = ttr[0];
/** Gives this player a random color. */
self.image_blend = irandom(16777215);
}
```

The `self.pressed...` variables will store if our player pressed the buttons to move or to jump. We need to store this in a variable as a preperation for later, when we want to add the player to the engine. But you'll see :).

Into the **step event** put the following code:

```
///Perform platforming!
//This is a very basic example of a platformer. You should not program your physics␣
↪like this!

if (place_free(x, y+1)) {
   gravity = 0.2;
} else {
   gravity = 0;
   vspeed = 0;
}

self.pressed_jump = keyboard_check(vk_space);
self.pressed_left = keyboard_check(vk_left);
self.pressed_right = keyboard_check(vk_right);

if (self.pressed_jump && (!vspeed)) {
    vspeed = -12
    image_xscale = 1;
    image_angle = 90;
}
if (self.pressed_right) {
    x+=10;
    image_xscale = 1;
    image_angle = 0;
}
if (self.pressed_left) {
    x-=10;
    image_xscale = -1;
    image_angle = 0;
}
if (vspeed < 0) vspeed=vspeed+gravity;
```

As you can see, we currently just put the button presses into the variables `self.pressed...` and check them. We will change that later.

Now create a **Collision with htme_obj_wall event** and put the following code in it:

```
///Set speed to 0 when hitting a top wall
if (!place_free(x, y-16)) {
    move_contact_solid(90,-1);
    vspeed = 0;
}

if (!place_free(x, y+16)) {
   move_contact_solid(270,-1);
   gravity = 0;
   vspeed = 0;
}
```

Into the **draw event**, we will put some code to draw the name we randomly generated in the create event. First put "**draw self**"" into the event and then this code:

```
///Draw nameplate
draw_set_color(image_blend);
draw_set_halign(fa_center);
draw_text(x,y-sprite_yoffset-5-string_height(self.name),self.name);
draw_set_halign(fa_left);
draw_set_color(c_white);
```

Now, the last thing to do is to place one `htme_obj_player` in the play room `htme_rom_demo`. Every player that joins the game will create its own player object. If you add two player objects in the room. Every player create two player objects.

Basic platformer: Done!

4. Test

Start the game and then start the server. You can now test out how bad the platformer really is. But hey, it does the job. The next tutorial is actually related to the engine again.

The network controller

Before we continue, let's create an object that makes sure, the engine is still running.

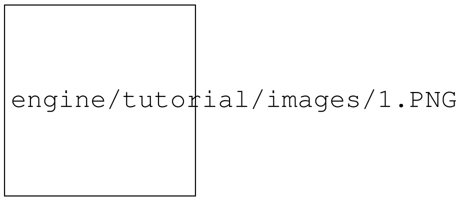

Fig. 2.1: How it should look like

The **engine will stop** running on the client-side, when the **connection** between server and client is **lost**.

Create the object `htme_obj_network_control`. Into the **Step-Event** put the following code:

```
///Check if engine running
if (htme_isLostConnection()) {
    htme_error_message_handler("Game Server or Client died! Go back to menu...");

    // Clean other persistent non synced objects from room
    with htme_obj_chat instance_destroy();
    with htme_obj_playerlist instance_destroy();

    room_goto(htme_rom_menu);
}
```

If `htme_isLostConnection()` **returns true** at any point after your client connected to the server, you can be very certain, that you **lost conection** to the server. Into the if Statement, simply put your own code to handle this scenario. `htme_isLostConnection()` should always return false for the server, unless the server dies due to an errror.

Oh, and don't forget to put the object into the `htme_rom_demo`.

Adding a player

Okay, okay, I know we already added a player in Step 5.

What this title means, is simply that we are now **adding the player to the engine**. After this step the player will be **synchronized between all players you connect to your server**.
So we are basicly done after this step. The rest of the tutorial is some more advanced stuff.

The create event

To set the player up, we only have to make minimal adjustments. **Create a new code block in the create event**, and insert it **before** the other block.

Start the code block with

```
mp_sync();
```

This will **tell the engine to sync this object**. Once this was created on one client, this instance wil also be **sent to all other players**.

If you add one player instance to your room and connect 4 players togther, each player will have four player instances.

Now we need to tell the engine what variables to sync and how:

```
mp_addPosition("Pos",5*room_speed);
```

This will tell the engine to sync the position variables `x` and `y` every 5 seconds. `5*room_speed` means 5 seconds. The first argument of `mp_addPosition` is the name of group. You can choose that however you want.

What we just added is a so called **variable group**. The group is called "Pos", get's synced every 5 seconds and sync the variables `x` and `y`.

Let's set **how the position should be synced**:

```
mp_setType("Pos",mp_type.SMART);
```

This changes the **sync type** to the variable group "Pos". **The default sync type is FAST**. We are changing it to **SMART**. Here is a list of what every sync type does:

- **FAST** (default, you don't have to use `mp_setType` if you want to use that): In our case, if we had chosen FAST, the engine would send the position of our player every 5 seoncds **once**. Since we are using UDP for networking, there is no assurance, that it will actually arrive. That means we don't know if the other players actually get our position every 5 seoncds. The packet could get lost. However, using FAST is very... FAST. You will see when we want to use FAST later in this section.
- **IMPORTANT**: When using IMPORTANT, in our case the position would still be synced every 5 seoncds. However, we are using a special way in *GMnet ENGINE* to make sure the packet arrives. Every 5 seconds, the server/client will flood the other players with the information, until they respond back, that they got it. That's how we assure the information is actually sent. This is however a pretty traffic heavy operation and using that in too short intervals will cause lagg in both connection and framerate.
- **SMART**: This is like IMPORTANT but better. Instead of syncing every 5 seconds, we will only sync every 5 seconds what has changed. If only the x position changed, we are not syncing the y position. If our position has not changed at all, we are not syncing. This is basicly a more traffic-friendly version of IMPORTANT.

We are using **SMART** here and relatively large intervals, because we only sync the position as a backup. As you will see later, we are going to sync the button inputs every single step. **We only sync the position to make sure, the players don't get desynchronized**.

The** button input later will be synced FAST**. That means **some packets could be lost**, and after some time the player could be on two completly different parts of the level, depending on the complexity of your platformer. To make sure that doesn't happen, **we reset the position every 5 seconds**.

Now, the thing is, if we reset it every 5 seconds, it might happen that **the players are just some pixels off**. In that case our players might **slightly "flicker"** whenever the position resets. **That doesn't look nice**.

Let's add a little **tolerance range**. If the position at is **less than 20 pixels away from where it should be**, the other players should **not apply this change** locally, so it **doesn't flicker** that much:

```
mp_tolerance("Pos",20);
```

Great. Position is set up. Now, **just to make sure**, let's also sync **the basic Game Maker physics and drawing variable**s:

```
/**
 * Tell the engine to add the basic drawing variables:
 * image_alpha,image_angle,image_blend,image_index,image_speed,image_xscale
 * image_yscale,visible
 */
mp_addBuiltinBasic("basicDrawing",15*room_speed);
mp_setType("basicDrawing",mp_type.SMART);
```

```
/**
 * Tell the engine to add the builtin GameMaker variables:
 * direction,gravity,gravity_direction,friction,hspeed,vspeed
 */
mp_addBuiltinPhysics("basicPhysics",15*room_speed);
mp_setType("basicPhysics",mp_type.SMART);
```

We don't need to sync them that frequently. If we sync physics to frequently that might look weird and we are only syncing the basic Drawing stuff for the player color (`image_blend`) which doesn't change anyway, and the `image_xscale` and `iamge_angle` which controls how the player faces, which isn't that critical.

Now we also want to **sync the name**:

```
mp_add("playerName","name",buffer_string,60*room_speed);
mp_setType("playerName",mp_type.SMART);
```

This is using `mp_add` to sync our own variables. The syntax is slightly more compelx and we need to do some things to make this work later, but let's just see what we got here: * The first argument is again the name of the group * In the second argument you specify the **names of the variables** you want to sync, **seperated by commas**. We stored the player name in the "name" variable. * The third argument is the **buffer type**. You can find information on which buffer type you need to use on this manual page. **All variables need to have the same buffer type.** `buffer_string` simply means that the variable "name" stores a string. * The last argument is the syncing interval, just as before.

We decide for a 60 seconds interval, because the name will never change. We could have also used 20 years, that wouldn't make a difference. We only need to make sure the engine syncs it on login and some other critical events, and it does that automatically, no matter what interval we choose. We still need to make it SMART because we need to make sure it REALLY arrived at those events.

Next up are the **controls**. Remember how we stored the button input in seperate variables? Well now you might know why:

```
mp_add("controls","pressed_jump,pressed_left,pressed_right",buffer_bool,1);
```

The "buffer type" is `buffer_bool`, because our "pressed_" variables are booleans. 1/0, true/false.

This will **sync the button input to all players every single frame** no matter what. We don't want to have it SMART or IMPORTANT. **FAST is the way to go**, since there is **no point in checking if the data arrives**, because we are syncing the button input every step anyway.

Some things needed when using `mp_add`

Since we are using `mp_add` to sync our own variables, we need to make some code changes. We need to send the variables to the engine at the beginning of the step, and retrieve the data at the end.
The reason fot that is, that in Game Maker there is no way of getting a variable's content by accessing it via a string. We need to store the values to a map first before the engine can read them. This is not needed for `mp_addPosition`, `mp_addBuiltinBasic` and `mp_addBuiltinPhysics`, because we hardcoded them.

If you don't know what any of that means what I just said, don't worry. It's complicated.

The only things you need to know, we will explain them now:

For every object where you use `mp_add` add the following to the **begin step** event:

```
mp_map_syncIn("name",self.name);
mp_map_syncIn("pressed_jump",self.pressed_jump);
mp_map_syncIn("pressed_left",self.pressed_left);
mp_map_syncIn("pressed_right",self.pressed_right);
```

Replace the names with the names of your synced variables. These are the variables that we created above for our tutorial player.

All changes to these variables need to be made **BEFORE** using these functions. That means you either have to change them in begin step, or call `mp_map_syncIn` again after you changed variables. We recommend the first. And that's also what we are going to do in a minute.

In the **end step** event add the following code to retrieve the variables again:

```
self.name = mp_map_syncOut("name", self.name);
self.pressed_jump = mp_map_syncOut("pressed_jump", self.pressed_jump);
self.pressed_left = mp_map_syncOut("pressed_left", self.pressed_left);
self.pressed_right = mp_map_syncOut("pressed_right", self.pressed_right);
```

Setting up controls for synchonization

Last thing we need to do, is to **move this code out of the step event** we created earlier:

```
self.pressed_jump = keyboard_check(vk_space);
self.pressed_left = keyboard_check(vk_left);
self.pressed_right = keyboard_check(vk_right);
```

Simply remove it. In begin step, **add the following code before** the other code:

```
if (htme_isLocal()) {
    self.pressed_jump = keyboard_check(vk_space);
    self.pressed_left = keyboard_check(vk_left);
    self.pressed_right = keyboard_check(vk_right);
}
```

It should now look like this:

```
if (htme_isLocal()) {
    self.pressed_jump = keyboard_check(vk_space);
    self.pressed_left = keyboard_check(vk_left);
    self.pressed_right = keyboard_check(vk_right);
}
mp_map_syncIn("name",self.name);
mp_map_syncIn("pressed_jump",self.pressed_jump);
mp_map_syncIn("pressed_left",self.pressed_left);
mp_map_syncIn("pressed_right",self.pressed_right);
```

All the code we just pasted in begin step does, is **check if this is the instance that was locally created** and then **writes the button input of the players into the variables**.

This way when we have 4 players, we only move the instance we control, the instance we locally created. The `self.pressed_jump...` variables will be changed by the other 3 players for the rest of the three instances.

Look at the following table from the view of player 1:

Instance/Player	Controlled via buttons	Controlled via engine
Ours / Player 1	Yes	No
Player 2' s	No	Yes, buttons from Player 2
Player 3' s	No	Yes, buttons from Player 3
Player 4' s	No	Yes, buttons from Player 4

Test

Fire up two games and create a server / connect to 127.0.0.1.

You should now see both players, **and should see that we now have a multiplayer platformer.**

A second room and doors

Now you know nearly everything needed to make a very simple multiplayer game.

The next sections will cover the following more advanced, but still not very difficult, topics:

- Using the multiplayer engine with mutliple rooms
- Working with the data of the multiplayer engine

We are now going to **create a second room**. Simply** copy** `htme_rom_demo` and restructure the walls a bit. Maybe you could also change the background color. **Remove all other objects from the room, except the walls and ‘‘htme_obj_network_controller‘‘.**

There are two aproaches you could use if you want the player to enter this room:

1. Place a new player instance in `htme_rom_demo2`, make the player object NOT persistent
2. Make the player instance persistent; Persistent objects in Game Maker exist even after you change the room.

For this tutorial we are using the **second approach**. The reason for that, is that we also want to teach you, how persistent objects work with the instance. But if you solve your game using the first method, this should also work in most cases. Please note that the second approach is recommend though.

Make your ‘‘htme_obj_player‘‘ persistent. That's all we need to do in the player object if we want him to move to a second room.

Now **create the object** `htme_obj_door` with the sprite`htme_spr_door`.

Create a **Collsion with ‘‘htme_obj_player‘‘ event** and add this code:

```
///CHECK IF DOOR ENTERED
var isLocal;
with (other) {isLocal = htme_isLocal();}

if (isLocal && keyboard_check_pressed(vk_up)) {
    var dest = htme_rom_demo2;
    if (room == htme_rom_demo2) dest = htme_rom_demo;
    room_goto(dest);
}
```

This checks when a player is standing in front of a door if the other instance, which is an instance of the object `htme_obj_player` is our local player by storing this information in the local variable `isLocal`. When it's our local player ("we") we change the room. The room is changed to `htme_rom_demo2` when in `htme_rom_demo` and viceversa. **Place a door in each room**.

We are done.

Now start a client and server again and try switching rooms. Everything should work as you would expect it. If not, check again if you made the player object persistent and make sure the doors are on the same position in both rooms.

Some technical stuff

When you make persistent objects, they are only persistent locally.

That means when another player switches room, they will cease to exist in your game. When you enter the room they will be resent and recreated to your client.

Here is a nice little table **FOR AN INSTANCE CREATED BY A IN ROOM 1**:

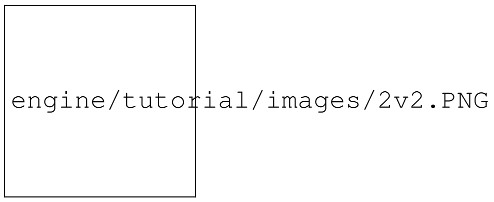

Fig. 2.2: The table

We will extend this table later with a third scenrio that makes the instances visible/existent at all time for all players.

Overlay that shows the name of connected players

We will now create a list that is displayed in the top left corner of the screen and will look something like this:

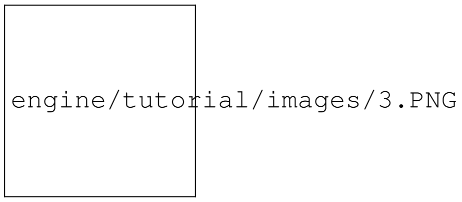

Fig. 2.3: A preview of the player list

Create a new object `htme_obj_playerlist` with no sprite, make it persistent so it exists in both game rooms, and **place it** in `htme_rom_demo`.

This new object is actually not going to be added to the engine. This object will simply look for all existing `htme_obj_player` instances and write their metadata on the screen.

The only thing we'll need is a **Draw-Event**.
Let's start with drawing the word "Players:" on screen:

```
draw_set_color(c_white);
draw_text(40,35,"Players:");
```

Okay, cool, cool. But how on earth do we get all player instances? There are two possibilites:

1. You could use `with htme_obj_player {/*Code*/}` to loop through all player instances
2. The engine provides some tools to get a list of all connected players and a function to get an instance that is controlled by a particular player.

In our case the first solution is the easiest. However we want to teach you what you can do with the engine, and so **we are going to use the second method**. There might be cases where looping through all instances of an object would be simply to much, for example if you have a race game with 2 players and 6 cpus and you only want to get data of the connected players.

```
var playerlist = htme_getPlayers();
```

This will give you a `ds_list` containing the hashes of all players. Each player is identifed by a random 8 character hash.

To loop over this list, add the following for-loop:

```
for(var i = 0;i<ds_list_size(playerlist);i++) {
    var player = ds_list_find_value(playerlist,i);
}
```

`player` will now be the playerhash of a player. We use this hash to get a player instance of this player. **Inside the for loop after ‘‘var player =...‘‘ insert the following:**

```
var instance = htme_findPlayerInstance(htme_obj_player,player);
```

This will either **return an instance or -**1. It will **return -1 when the player is in another room**, because no instance for this player exists. **And that’s a problem**.

Because that means we can’t actually get the name of the player to display it then. That’s a limitation because we store the player name with our player object. After this tutorial is over you should be able to solve this issue, and I challenge you to do so!

Because of this limitation we will simply show “(other room)” in a grey color, when the player instance does not exist:

```
if (instance != -1) {
    var name = (instance).name;
    var _image_blend = (instance).image_blend;
} else {
    var name = "(Other room)";
    var _image_blend = c_gray;
}
```

‘‘name‘‘ now contains the name of the current player and ‘‘_image_blend‘‘ the color.

The following code will draw the sprite and the name:

```
//Draw small player icon
draw_sprite_ext(htme_spr_player,0,50,(i)*20+70,0.5,0.5,0,_image_blend,1);
//Draw player name
draw_set_colour(_image_blend);
draw_text(70,(i)*20+62,name);
```

Remember, all of this must go into the for-loop.

Start the game and test it out, our player list should work just fine.

Adding day and night

Now we want to add time and a day and night cycle. Obviously we want to sync the time between all players.

So let's just create a new object, I guess?

...But wait. If we simply sync a time object using the engine, we will end up having one time object per player. For 4 players that means we would have four time objects fighting over each other.

Of course we could tell all objects to only increase the time when it's locally, but then they would all have their own time counter and that would have the same effect of not syncing them at all.

Oh, if there would only be an easy way to fix this... Oh! Wait! There is!

You can check if the object was **created on the server** with `htme_isServer()`. This way you can** add objects that get controlled by the server and synced to each player**. You have 1 instance on 4 PCs, instead of 4 instances at 4 PCs.

Let me show you how this works.

Create a new object called `htme_obj_time`. Set depth to 1 so it will draw in the background behind the player object. And add the following code in the beginning of the create event:

```
if (!htme_isServer()) {
   instance_destroy();
   exit;
}
```

Let's pretend **I'm a client**:
I create the instance, **see that I'm not the server**, instantly **destroy it** and then **recieve the instance from the server controlled by the server**.

Genius!

After that bit of code we add the code to add our new object to the engine, which is of course never executed when created clientside:

```
mp_sync();
mp_add("time","time",buffer_u16,20*room_speed);
mp_setType("time",mp_type.SMART);
//And at the end we set the time to 1000 - this will be noon
self.time = 1000;
```

This way the** clients recieve the time every 20 seconds**. We are going to program the counter so, that it counts up, even clientside. That means **each player's game counts up on their own**, and the server **corrects the time if neccessary**.

Because we used ‘‘mp_add‘‘ we need to add the following code to **begin step**:

```
mp_map_syncIn("time",self.time);
```

And this to **end step**

```
self.time = mp_map_syncOut("time", self.time);
```

The two things left are counting the time up and actually doing something with it.

Increasing the time

Remember that, because we use `mp_add`, all changes to our `time` variable need to be made before `mp_map_syncIn` is called. Because of that, create a new code block in **position 1 of begin step**:

```
///Increase the time. If time is greater than 2000, set to zero.
self.time++;
if (self.time > 2000) self.time = 0;
```

This will count up our time to 2000 which is midnight, and then reset it to 0, which is also midnight. 1000 is noon.

If you want to perform actions on these server controlled instances only by the server use `if (htme_isLocal()) {}`. Everything inside this statement will only be executed by the server.

Day and night

Paste the following code into the **Draw-Event**. This will change the background color in the first room depending on time and display the time. Room 2 will be interior.

```
///Draw background
if (room == htme_rom_demo) {
    //Draw night/day
    //This is not a good way of doing it, but I'm not in the mood for that :D
    if (self.time == 0) {
       var bgcolor = make_colour_hsv(170,185,0);
    } else if (self.time <= 1000) {
       var bgcolor = make_colour_hsv(170,185,255/100*(self.time)/10);
    } else if (self.time == 1001) {
       var bgcolor = make_colour_hsv(170,185,255);
    } else if (self.time <= 2000) {
       var bgcolor = make_colour_hsv(170,185,255/100*(1000-self.time)/10);
    }
    draw_set_colour(bgcolor);
    draw_rectangle(0,0,room_width,room_height,false);
    //Draw time as debug on screen
    draw_set_colour(c_white);
    draw_text(room_height-70,70,"Time: "+string(self.time));
} else {
  draw_set_colour(c_maroon);
  draw_rectangle(0,0,room_width,room_height,false);
}
draw_set_colour(c_white);
```

That's not right...

Now when testing what we just did, you might realize that **it doesn't work right**. **Even** if you set `htme_obj_time` to be **persistent**, the time object **just vanishes sometimes**. Let's take a look again at our nice table again:

As you can see in this table, **if the server is A and A is in Room 2, the time object will simply disappear on all clients**. Or if the server is in Room 1 and the client(s) in Room 2. And if you don't even set it to be persistent, it even disappears for the server if he is in Room 2. A nightmare!

That's not good! We want our new time object to allways exist, no matter what!

So, first, **set ''htme_obj_time'' to be persistent**. Now,** after the `mp_sync();` in the create event** add this:

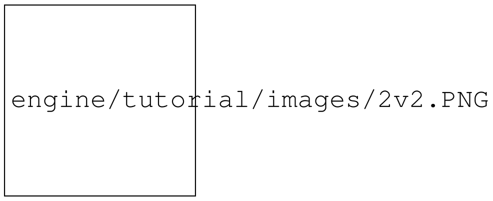

Fig. 2.4: The nice table

```
mp_stayAlive();
```

This tells the engine to **keep this instance alive, no matter what**. This ONLY works if the object is persistent!

When we add this to our table, things look like this:

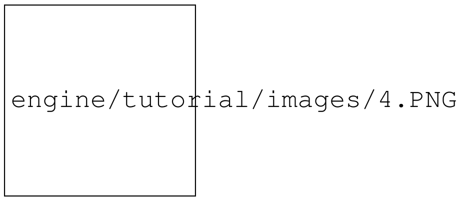

Fig. 2.5: The even nicer table

As you can see, with stayAlive enabled, the instance always exists.

Done!

Time and day is done. Test it out!

We are now ready for the final chapter...: A chat system...

Chat

In this chapter you will learn a new way of syncing information between players. This method was designed for chat messages and similiar things.

For this we use the so called CHAT (Custom Handler for Advanced Traffic) Interface. You will learn how to use it in this chapter and can find more advanced usages of it in chapter Bonus 5 - RPC.

First, let's **create an object ‘‘htme_obj_chat‘‘**.

We are storing messages in the variable **message**. Or at least **the last** message sent by a player. Then we are **syncing it** via the engine, and the **local instance** of that object **will loop through all chat instances**, like we learned when creating the chat overlay. It will then **look for new messages in our "inboxes"**, and **add them to a list that contains all messages**.

Adding a CHAT handler

Create a **Create-Event** and add the following code:

```
mp_syncAsChatHandler("Chat");

//Setup variables
self.output = ds_list_create();
```

‘mp_syncAsChatHandler <functions/chat/mp_syncAsChatHandler>‘__. This function **registers the object to be a sender and reciever for string messages** for the chanel **Chat**. *‘mp_sync <functions/chat/mp_sync>‘*__ is not needed here. We won’t be syncing the object itself, only messages. But you’ll see what that means in a second.

The ds_list **ouput** is used to **store all chat messages**.

Sending a message.

We are going to send a message when the player presses the C-key. Create a **press C-key Event**:

```
self.str_id = get_string_async("Send a chat message:","");
```

This script uses **‘‘get_string_async‘‘** to **ask the player for the chat message**. If you never used that before, things might be a bit complicated, so let me explain:

`get_string_async` displays an input box where the player can input text without locking the game, like `get_string` would do. That’s important, because the client or server will loose connection, if the game is locked. Instead of returning the message, `get_string_async` returns a number, that we need to store. Inside the **Dialog Event, which can be found under Asynchronous** we are retrieving the message:

```
///Process the message the player typed in
var i_d = ds_map_find_value(async_load, "id");
if (i_d == self.str_id) {
   if (ds_map_find_value(async_load, "status")) {
      if (ds_map_find_value(async_load, "result") != "") {
         var message = ds_map_find_value(async_load, "result");
         //Send the message using the CHAT Interface.
         mp_chatSend(message);
      }
   }
}
```

This may seem a bit complicated, the code checks that the message entered was valid, but the important code is the code in the most inner if-clause.

`var message = ds_map_find_value(async_load, "result");` retrieves the message and writes it to the local variable **message**.

Using **‘mp_chatSend(message) <functions/chat/mp_chatSend>‘**__ we are now sending this message to all players. It will also be sent to ourself.

Recieving messages

Messages are stored in a ds_queue, which is filled when new messages arrived. We will be checking every step if new messages arrived.

In the step event, add the following code:

```
if htme_isStarted() {
    var queue = mp_chatGetQueue();
    while (ds_queue_size(queue) > 0) {
        var raw_message = ds_queue_dequeue(queue);
```

The first line uses mp_chatGetQueue to get the queue of messages and then **we loop through it until it is empty**. `ds_queue_dequeue` will **get** the oldest **message** in the queue and remove it from the queue.

This message however is "encoded", so we now use two functions to decode it:

```
var sender = htme_chatGetSender(raw_message);
var message = htme_chatGetMessage(raw_message);
```

htme_chatGetSender and htme_chatGetMessage will, as their name might suggest, return the author of the message (the hash of the player) and the sended message.

So now we have the message. What now?
As you may remember, we gave all our player objects names. **We want to display the sent message together with the name of the player**.
To do that, we first have get the name of the player object using the player hash:

```
var player_instance = htme_findPlayerInstance(htme_obj_player,sender);
 if (player_instance != -1) {
     var name = (player_instance).name;
 } else {
     var name = "(Someboy in another room)";
 }
```

This is the same as in the chapter 9, so check this page of the tutorial for more information.

The last thing we do, is we add the message we recieved to the list we created in the create event, together with who sent it, seperated by a ":".

```
        //Add to list of chat output
        ds_list_add(self.output,name+": "+message);
    }
}
```

Display the chat.

To display the chat, put the following code into the **draw-event**:

```
//Get an offset, so we draw the newest line on bottom
var size = ds_list_size(self.output);
var bottomLine = room_height-50;
var offset = size*20;

for(var i = 0;i<size;i++) {
    var line = ds_list_find_value(self.output,i);
    draw_text(50,bottomLine-offset+i*20,line);
}
```

That draw event is not essential part of the engine, so I'm not explaining it here. It basically loops over the output list and displays each line.

We are done. Have fun testing it out!

Private messages

You can also send private messages to certain players. mp_chatSend supports a second argument, where you can use the hash of a player to only sent your messages to specific players. Try it out!

Conclusion / What's next

You now learned everything there is to know.

Multiplayer with GMnet ENGINE might be a little bit different that what you are used to, but once you got the hang of it, the possibilites are endless.

Try extending the demo project we just created with some features.
Why not start with our challenge? Fix the player overlay and the chat to display the name of the players corectly, no matter what room they are in.

Play around with the engine for a bit, and eventually you will master it!

If you have any questions, find bugs or have feature suggestions, follow the instructions in the Support section.

If you want to learn even more advanced trics, like sending your own custom packets, check out the "Extending the Engine / Advanced Use" section.

Have fun creating awesome multiplayer games! And if you do, don't forget to tell us, we really want to see, what you can come up with.

PS: If you are stuck with our challenge here's a solution (encrypted with Caesar algorithm, shifted 1 letter in the alphabet up):

```
Dsfbuf b ofx pckfdu uibu tupsft uif obnf pg uif qmbzfst, jotufbe pg iunf_pck_qmbzfs.␣
↪Nblf ju qfstjtufou boe tubzBmjwf, tp ju bmxbzt fyjtut. Jo uif qmbzfsmjtu boe uif␣
↪dibu, mppq pwfs ju jotufbe. Bmufsobujwfmz zpv dbo bmtp tupsf uif obnf pg uif␣
↪qmbzfst jo uif dibu pckfdut.
```

Bonus: An ONLINE lobby

Note: This topic requires GMnet PUNCH enabled.

***NOTE:** Version 1.2.0 introduced LAN lobbys. This chapter only covers the online lobby*

The update to version 1.0 introduces a new lobby system for online servers.

That means you can now add an online lobby to your game.

This part of the tutorial will explain everything you'll need to know. Before we start, please **create a new project and load the sample project** (add all files from the GMnet ENGINE asset).

We now have everything we created in this tutorial and more. The lobby is one of these things. Let me take you on a tour on how it works.

GMnet PUNCH needs to be activated. Please follow chapter 2 and 3 for this.

Testing the lobby

When you **press L when starting the game** you will be brought to the demo lobby room. This is a very simple demo room that can only show four servers. Once we are done, you'll be able to create an even better lobby.

To test it, see if a server is online and connect to it via the lobby. Please note that you have to set up a master server before, as explained in chapter 3. If you use the demo master server, please note that there might be "GMnet PUNCH Demo Servers" in the server list. You can not join them, they were created using the UDPHP demo project. Both use the same lobby and the same demo master server.

You can also create a server and fire up a second game and see if you can join it.

The new concepts: Gamenames

In `htme_config` there is a new variable you need to change when creating your own game (keep the default for the demo project though, otherwise you can't join demo servers via the lobby):

```
/**
 *  Shortname of this game
 *  + version
 *  Example: htme_demo100
 **/
self.gamename = "htme_demo100"
```

This string represents your game and the version of your game. If you update your game and it's server is not compatible with older versions, change this string and you can prevent players from joining older servers in the lobby (Please note that this is only for the lobby, if your players are able to connect manually, you need to implement your own way of kicking him out of the game again after he has connected, see *How can my clients get the gamename and data strings after they connected?* below for more info)

You can change it at any time, for example if you have multiple gamemodes you want to mark in the lobby via `htme_setGamename(name)` and get it via `htme_getGamename()`.

Data strings

Aside from the gamename string, there are 7 more strings that can be used to identify your game in the lobby.

Start the demo game and create a server. You will be asked to enter **the name of the server and a description**. These are data strings 2 and 3. The demo lobby uses them for these purposes, but when making your own game, you can use them however you want. 4-8 are unused by the demo project.

Let's open the **N-key** event of **htme_obj_menu** (this event creates the server).

You'll find this part after the server has successfully been created:

```
htme_setData(2,get_string("How should this server be called (for the lobby)?","GMnet␣
↪ENGINE Demo Server"));
htme_setData(3,get_string("Enter a server description (for the lobby)?","A server␣
↪created for the GMnet ENGINE demo project"));
```

As you can see, we set the server name and description right after the server was started using `htme_setData(n, string)`. We use 2 for server name and 3 for description. You can also on the server end retrieve these strings using `htme_getData(n)`.

If you want to update any data string later, after the server connected to the master server, you need to use `htme_commitData()`. This syncs all data strings to the master server. You MUST NOT use it here, because the server hasn't connected to the master server yet!

A note for GMnet PUNCH users: The gamename is data string 1.

Building the lobby

Ah, the lobby. Finally we are ready to build it.

The room of the lobby is the room `udphhtme_lobby`. This room only contains the object `obj_udphphtme_lobby`. This object controls the lobby. Let's dive into it!

The create event

```
//IF YOU USE GMnet PUNCH - it will only let you connect to GMnet PUNCH servers:
if (!script_exists(asset_get_index("htme_init"))) {
   self.game = "udphp_demo120"
}
//IF YOU USE GMnet ENGINE - it will only let you conect to GMnet ENGINE servers
else {
   self.game = "htme_demo121"
}
//IF YOU USE YOUR OWN SERVER - Change self.game!

///Recieve lobby data from the master server
udphp_downloadServerList(4,"date","DESC",self.game);
```

First the variable `game` gets set. We use this to prevent GMnet ENGINE players from joining GMnet PUNCH servers and vice-versa. As said before, this object is used in GMnet ENGINE demo project and GMnet PUNCH standalone demo project, this is why it has seperate code for both.

The important part here is `[udphp_downloadServerList](functions/udphp_downloadServerList)`. This will tell GMnet PUNCH (the part of the engine that controls communication with the master server) to download a list of servers from the master server. The paramters allow for filtering and sorting the result, we use this to only get results that match our game name. More information about what you can filter, can be found on the usage page of udphp_downloadServerList.

You use your own filtering variables later when creating your lobby.

The networking event

```
///Waits for master server response
udphp_downloadNetworking();
```

This code checks if the master server sent the server list and updates it. This is not included in `htme_networking();` and therefor has to be run here.

The draw event

The draw event is split up into different sub-scripts:

'Background', 'Title and Controls', 'Online servers'

Draws some background colors and some text, not important

'Servers (Loop)'

This draws the actual server list.

Let's analyze it:

```
///Servers (Loop)
var l = global.udphp_downloadlist;
for (var i = 0; i<4;i++) {
    draw_text(10,85+80*i,"=("+string(i+1)+")=");
```

First, the list `global.udphp_downloadlist` is stored in the local variable `l` (because it's shorter). **This ds_list contains all the servers we got from the master server**.

Then it begins a loop that loops through the first 4 servers in the list we got by the master server, everything is in this loop and then it draws a nice little number for each server.

```
if (ds_exists(l,ds_type_list)) {
    if (ds_list_size(l)>i) {
```

Now, this is the interesting part.

First we check if the downloadlist was already created (it get's created once the list has been downloaded). After that we check if it has at least as many entrys as the server we want to list. For this example we assume `i` is 1. That means it checks if there is atleast one server in the list. If yes, we have an entry we can now draw.

```
//Get stuff from the downloadlist
var entry = l[| i];
var ip = entry[? "ip"];
var game = entry[? "data1"];
var servername = entry[? "data2"];
var description = entry[? "data3"];
```

Now the entry (a ds_map) for our server is extracted from the list and we get the gamename, which is stored in data1, the ip, which is stored in the key "ip", the name of the server, which we stored in data2, and so on.

```
            draw_text(70,85+80*i,servername+" | "+ip);
            draw_text(70,115+80*i,description);
        }
    }
    draw_line(0,160+80*i,room_width,160+80*i);
}
```

Now we just draw everything.

‘Footer’

Again, just some text, not important.

The press 1-4 key events

Pressing 1-4 on the keyboard will connect to that game. Let’s see how!

```
///LOAD GAME SERVER ON SLOT 1
var l = global.udphp_downloadlist;
if (ds_exists(l,ds_type_list)) {
    if (ds_list_size(l)>0) {
        var entry = l[| 0];
        var ip = entry[? "ip"];
        var game = entry[? "data1"];
```

We again open the downloadlist and check if server 1 is in it, if yes we continue.

```
if (game != self.game) {
   //Not compatible game, exit
   show_message("Game server or version is incompatible!");
   exit;
}
```

Remember the filtering variable we created in the create-event? We use it here to check if the server is a GMnet ENGINE demo game. If not we cancel. Please note, that this is propably not needed here, since we filtered out all, but our game in the create event, when we ran udphp_downloadServerList.

```
        //====GMnet PUNCH DEMO ONLY
        if (!script_exists(asset_get_index("htme_init"))) {
           //This code is irrelevant for GMnet ENGINE and has been removed
        }
        //====GMnet ENGINE DEMO ONLY
        else {
            //Setup client, on success go to waiting room, otherwise end game
            //We don't actually know the port, but that doesn't matter, the master
            //server will tell us the port when we connect
            //
            //THE LINE BELOW is equivalent to:
            //if (htme_clientStart(ip, 0)) {
            if (script_execute(asset_get_index("htme_clientStart"),ip, 0)) {
                //Wait for connection success!
                room_goto(asset_get_index("htme_rom_connecting"));
            } else {
                show_message("Could not start client! Check your network␣
↪configuration!");
            }
        }
    } else {
      //Do nothing - There is no server on this slot
    }
} else {
  //Do nothing - There is no server on this slot
}
```

This is the rest of the script. We once again check if we are running the GMnet ENGINE demo project and then we begin connection.

We call `htme_clientStart` with the ip we got from the list and leave the port at 0. Why? Because we don't know the port. But that doesn't matter, because when connecting to the server, GMnet ENGINE will automatically resolve the port using UDPHP.

Afer that we just go to the waiting room. Done! The rest is the default client connection you know and love.

And this is how you create a lobby! Now go ahead and do it! :)

How can my clients get the gamename and data strings after they connected?

Now, you might want to get the datastrings and the gamename after you connected. For example to display the name of the server in a corner, or to make sure the server is comaptible to the client when connecting manually.

For this you just create a object like the time syncing object. You sync it with the engine, so all other players get it and sync 8 variables that contain the value of the gamename string and the 7 data strings.

If you need help with that, contact me :)

Anything missing?

If this bonus chapter lacks something important, please let me know. If you have any problems feel free to contact me. I know this is quite complicated compared to the rest, so don't be frustrated when you have problems.

Bonus: Global Sync

The update to version 1.1 added a new feature called "Global Sync".

Before you learned how to sync instances between games in this tutorial. However this concept has one downside: **Instances are only created and controlable by one client.** All variables of an instance are **read-only** to all other clients. That's why we needed to create a seperate chat instance for each player that basicaly all jut synced one chat variable. That may be fine for a few variables, and in this case maybe even the best way of doing it, since every player still needs one "chat outbox", however **you can now sync global variables that are read- and writeable by all clients at all times.**

Summary of Global Sync's features: **Change the value on one end and it will update on all other ends as well. All ends can update and access all variables**

How it works

It's super simple. **To store a variable** in the Global Sync pool use this little code:

```
var value = 1+1;
htme_globalSet("name",value,buffer_u8);
```

The third argument is the Buffer type/data type.

This will now sync to all clients immediately (using the SMART syncType.).

You can **retrieve this variable** via this code **on all connected clients and the server**:

```
var value = htme_globalGet("name")
```

Please note that this function may return `undefined` if it wasn't set by any of the clients/server yet, so check if it is defined:

```
if (is_undefined((htme_globalGet("name")) {
    //Nope, that wasn't set yet...
}
```

Bonus: A LAN lobby

Note: This topic requires GMnet PUNCH enabled.

***NOTE:** In addition to LAN lobbys, there are also ONLINE lobbys!

In addition to online lobbys, you can now also create LAN lobbys. In this tutorial we will explain how. We'll assume you haven't read the online lobby tutorial, but if you have: Creating a LAN lobby is similiar, but there are some key differences.

This part of the tutorial will explain everything you'll need to know. Before we start, please **create a new project and load the sample project** (add all files from the GMnet ENGINE asset). ###Testing the lobby

When you **press K when starting the game** you will be brought to the demo lobby room. This is a very simple demo room that can only show four servers. Once we are done, you'll be able to create an even better lobby.

To test it, fire up a demo game on another PC in the network and start a server. You'll see the server in the list after a few seconds.

The new concepts: Gamenames

(**This is the same text as in the online lobby tutorial**)

In `htme_config` there is a new variable you need to change when creating your own game (keep the default for the demo project though, otherwise you can't join demo servers via the lobby):

```
/**
 *  Shortname of this game
 *  + version
 *  Example: htme_demo100
 **/
self.gamename = "htme_demo100"
```

This string represents your game and the version of your game. If you update your game and it's server is not compatible with older versions, change this string and you can prevent players from joining older servers in the lobby (Please note that this is only for the lobby, if your players are able to connect manually, you need to implement your own way of kicking him out of the game again after he has connected, see *How can my clients get the gamename and data strings after they connected?* below for more info)

You can change it at any time, for example if you have multiple gamemodes you want to mark in the lobby via `htme_setGamename(name)` and get it via `htme_getGamename()`.

Data strings

(**This is the same text as in the online lobby tutorial**)

Asside from the gamename string, there are 7 more strings that can be used to identify your game in the lobby.

Start the demo game and create a server. You will be asked to enter **the name of the server and a description**. These are data strings 2 and 3. The demo lobby uses them for these purposes, but when making your own game, you can use them however you want. 4-8 are unused by the demo project.

Let's open the **N-key** event of **htme_obj_menu** (this event creates the server).

You'll find this part after the server has successfully been created:

```
htme_setData(2,get_string("How should this server be called (for the lobby)?","GMnet␣
↪ENGINE Demo Server"));
htme_setData(3,get_string("Enter a server description (for the lobby)?","A server␣
↪created for the GMnet ENGINE demo project"));
```

As you can see, we set the server name and description right after the server was started using `htme_setData(n, string)`. We use 2 for server name and 3 for description. You can also on the server end retrieve these strings using `htme_getData(n)`.

If you want to update any data string later, after the server connected to the master server, you need to use `htme_commitData()`. This syncs all data strings to the master server. You MUST NOT use it here, because the server hasn't connected to the master server yet!

A note for GMnet PUNCH users: The gamename is data string 1.

Setting broadcast settings.

By default the servers will broadcast their information to all PCs in the LAN each 15 seconds. You can change this in `htme_config` by changing the value of `self.lan_interval`

```
/**
 * Interval the servers broadcast data to the LAN, for the LAN lobby
 * @type real
 */
self.lan_interval = 15*room_speed;
```

Building the lobby

Ah, the lobby. Finally we are ready to build it.

The room of the lobby is the room `htme_lanlobby`. This room only contains the object `htme_obj_lanlobbydemo`. This object controls the lobby. Let's dive into it!

The create event

```
self.game = "htme_demo121"
//IF YOU USE YOUR OWN SERVER - Change self.game!

///Recieve lobby data from the master server
htme_startLANsearch(real(get_string("On which port should we search for servers?",
↪"6510")),self.game);
```

This will start searching the LAN for games with the game id `self.game` on the port that is asked to the player. More information on this function can be found in the usage page of htme_startLANsearch.

The Room end event

```
///STOP LAN SEARCH
htme_stopLANsearch();
```

This will stop searching for LAN servers. You should run this if your player leaves the lobby

The networking event

```
///LOOKING FOR INCOMING SERVER BROADCASTS
//htme_step doesn't do that btw!
htme_networking_searchForBroadcasts();
```

This runs the code that actually searches for LAN servers. It waits for incoming server broadcasts.

The draw event

The draw event is split up into different sub-scripts:

'Background', 'Title and Controls', 'Online servers'

Draws some background colors and some text, not important

'Servers (Loop)'

This draws the actual server list.

Let's analyze it:

```
///Servers (Loop)
var l = htme_getLANServers();
for (var i = 0; i<4;i++) {
    draw_text(10,85+80*i,"=("+string(i+1)+")=");
```

First, the list `htme_getLANServers()` is stored in the local variable `l` (because it's shorter). **This ds_list contains all the servers we found**. It will be filled over time with all servers in the LAN.

Then it begins a loop that loops through the first 4 servers in the list we got by the master server, everything is in this loop and then it draws a nice little number for each server.

```
if (ds_exists(l,ds_type_list)) {
    if (ds_list_size(l)>i) {
```

Now, this is the interesting part.

Ee check if it has at least as many entrys as the server we want to list. For this example we assume `i` is 1. That means it checks if there is atleast one server in the list. If yes, we have an entry we can now draw.

```
//Get stuff from the downloadlist
var entry = l[| i];
var ip = entry[? "ip"];
var port = entry[? "port"];
var game = entry[? "data1"];
var servername = entry[? "data2"];
var description = entry[? "data3"];
```

Now the entry (a ds_map) for our server is extracted from the list and we get the gamename, which is stored in data1, the ip, which is stored in the key "ip", the name of the server, which we stored in data2, and so on.

```
            draw_text(70,85+80*i,servername+" | "+ip+":"+string(port));
            draw_text(70,115+80*i,description);
        }
    }
    draw_line(0,160+80*i,room_width,160+80*i);
}
```

Now we just draw everything.

'Footer'

Again, just some text, not important.

The press 1-4 key events

Pressing 1-4 on the keyboard will connect to that game. Let's see how!

```
///LOAD GAME SERVER ON SLOT 1
var l = htme_getLANServers();
if (ds_exists(l,ds_type_list)) {
    if (ds_list_size(l)>0) {
        var entry = l[| 0];
        var ip = entry[? "ip"];
        var game = entry[? "data1"];
```

We again open the downloadlist and check if server 1 is in it, if yes we continue.

```
if (game != self.game) {
   //Not compatible game, exit
   show_message("Game server or version is incompatible!");
   exit;
}
```

Remember the filtering variable we created in the create-event? We use it here to check if the server is a GMnet ENGINE demo game. If not we cancel. Although this is not needed if you filtered out other games with htme_startLANsearch like we did in the create event.

```
        if (htme_clientStart(ip, port)) {
            //Wait for connection success!
            room_goto(htme_rom_connecting);
        } else {
            show_message("Could not start client! Check your network configuration!");
        }
    } else {
      //Do nothing - There is no server on this slot
    }
} else {
  //Do nothing - There is no server on this slot
}
```

This is the rest of the script. We call `htme_clientStart` with the ip we got from the list the port.

Afer that we just go to the waiting room. Done! The rest is the default client connection you know and love.

And this is how you create a lobby! Now go ahead and do it! :)

Anything missing?

If this bonus chapter lacks something important, please let me know. If you have any problems feel free to contact me. I know this is quite complicated compared to the rest, so don't be frustrated when you have problems.

Bonus: Event Handlers for Connecting/Disconnecting

Sometimes you may want to exceute code when players connect or disconnect. Or you want to prevent players from joining after the game has started.
In this chapter, we will teach you how you can do that. It is very easy.

Basic example

Let's say you have this script called `scr_welcome`:

```
show_message("Cool! Someone connected!");
return true; //- We will come to that later
```

You want to execute this script, if someone connects. To do that simply use this code somewhere after the engine was started (obj_htme is created):

```
htme_serverEventHandlerConnecting( scr_welcome );
```

That's all you have to do. For a script to be run at disconnection, simply call `htme_serverEventHandlerDisconnecting` instead.

What is the return value for?

As you saw in the example above, the script `scr_welcome` returns `true`.

If you are using a script as an event handler for **connecting** players, you need to return either true or false (you don't have to return anything for the Disconnect-Event).

When your script returns **true**, the connection is **accepted** and the player is **connected**.
When your script returns **false**, the connection is **refused** and the player will be **kicked** before he even really connected.

Getting more information about the player

Both, the disconnect and connect event, provide an argument for your script.

The **connect event** gives your script an argument0 containing a **ds_map** with the keys **port** and **ip**.

The **disconnect event** also contains this ds_map, but **additionally** has a key **hash** containing the **player hash** that you can also use.

Here is an example how you can use this information. The server will refuse all connections from the local computer when using this script:

```
///somewhere in your code
htme_serverEventHandlerConnecting( scr_no_local_clients );
htme_serverEventHandlerDisconnecting( scr_goodbye );
```

```
///scr_no_local_clients(player_map);
var player_map = argument0;

if (player_map[? "ip"] == "127.0.0.1") return false;
else {
    show_message("Cool! "+player_map[? "ip"]+":"+string(player_map[? "port"])+"␣
↪connected!");
    return true;
}
```

```
///scr_goodbye(player_map);
var player_map = argument0;

show_message(":( ! "+player_map[? "ip"]+":"+string(player_map[? "port"])+" with the␣
↪hash "+player_map[? "hash"]+" left!");
```

Bonus: RPC

RPCs (Remote procedure calls) are used to execute scripts/code/functions remotely.

For the engine this means RPC enables you to execute scripts on other clients / servers. After you finished this part of the tutorial you will have a fully functional way of calling scripts via the network.

This tutorial will also show you how to send the value returned by the RPC-called script back to the game of the player who called it.

WARNING: Please note that this tutorial does not cover any secruity mechanisms. You propably want to restrict which scripts can be executed and who and when clients can execute them and on which targets.

To make RPCs with GMnet ENGINE we will be using the CHAT Interface. Basic knowledge on how to use that is required.

**This section is more advanced and good coding skills are required [don't fear to try it out though :)] **

Setup

First we need an object to listen for RPCs and to send RPCs through.

Create an object **obj_htme_rpc** with the events below, make it **persistent** and make sure it is created when clients have connected or a server was started, and get destroyed when the engine shuts down.

Create Event

```
mp_syncAsChatHandler("RPC");
```

Step Event

```
var queue = mp_chatGetQueue();
while (ds_queue_size(queue) > 0) {
    htmerpc_recieve(ds_queue_dequeue(queue));
}
```

And create the two scripts ‘‘htmerpc_recieve‘‘ and ‘‘htmerpc_send‘‘. The will be used to send and recieve RPC calls. And also create the script ‘‘htmerpc_callback‘‘. This script will later be used to store the values that the RPC-called scripts return.

The RPC protocol

We will now create a protocol to send and recieve the RPCs.

The message

Okay, so let’s think about how we can “package” our RPCs.

Each RPC needs to contain a script name and it’s argument. And each RPC has one recipient that should execute the command.

Let’s use a ds_map for this. GameMaker allows you to convert ds_maps into JSON which can be sent over the network and then be decoded again.

Our message will be a ds_map, turned into a json string with the following contents:

```
ds_map =>
   [id] => string
   [script] => string
   [argument_count] => real
   [argument0] => string or real (optional)
   [argument1] => string or real (optional)
   ...
   [argument12*] => string or real (optional)
```

`id` is used to identify the RPC calls. You will see why this is important later.

When we later convert these ds_maps into JSON, the message will look like this:

```
{"id":"foo","script":"scr_myscript","argument_count":2,"argument0":12.3,"argument1":
↪"Hello World"}
```

* Only 13 arguments can be used as the script `htmerpc_send` we will create now can only take in 16 arguments, and two of them are used for other things

Creating a script to send messages

We are now going to write `htmerpc_send`. This script should be used anywhere to send RPC commands.

This script will need: * The id of this RPC call. This is used to get the value that the script that was executed using this call later. * The script to execute * A target player (where the script should be executed) * The arguments

So let's fill the script with this:

```
///htmerpc_send(id,script,to,[argument0...13])
/* Sends RPC calls to another player */

/*
 * Turn script into a string. You call this function with (htmerpc(my_script,...);
 * my_script will be turned into a number by Game Maker that identifies this script,
 * we turn this into a string to send it over the network, because if you use
 * different versions of your game, this id might not be the same for the same script.
 */
var rid = argument[0];
var script = script_get_name(argument[1]);
var to = argument[2]; //Hash of the player to send this to
var rpc_argument;
if (argument_count > 3) rpc_argument[0] = argument[3];
if (argument_count > 4) rpc_argument[1] = argument[4];
if (argument_count > 5) rpc_argument[2] = argument[5];
if (argument_count > 6) rpc_argument[3] = argument[6];
if (argument_count > 7) rpc_argument[4] = argument[7];
if (argument_count > 8) rpc_argument[5] = argument[8];
if (argument_count > 9) rpc_argument[6] = argument[9];
if (argument_count > 10) rpc_argument[7] = argument[10];
if (argument_count > 11) rpc_argument[8] = argument[11];
if (argument_count > 12) rpc_argument[9] = argument[12];
if (argument_count > 13) rpc_argument[10] = argument[13];
if (argument_count > 14) rpc_argument[11] = argument[14];
if (argument_count > 15) rpc_argument[12] = argument[15];
```

This "simply" processes the arguments for the script. All arguments for the RPC script get written in the rpc_argument array. You can also do this using a loop by the way, which is far more elegant. We will use argument_count later again to see how many RPC arguments we have.

We actually don't have that much to do now. We create the ds_map...

```
var rpc_command = ds_map_create();
```

...fill it...

```
rpc_command[? "id"] = rid;
rpc_command[? "script"] = script;

//If argument_count is 4, we have 4 arguments, which means 1 rpc argument etc.
var rpc_argument_count = argument_count-3;
rpc_command[? "argument_count"] = rpc_argument_count;

//Now we loop through all arguments and add them to the list:
for (var i = 0; i < rpc_argument_count; i++) {
    rpc_command[? "argument"+string(i)] = rpc_argument[i];
}
```

...and convert it to json:

```
var message = json_encode(rpc_command);
//After that we don't need the map anymore
ds_map_destroy(rpc_command);
```

That's all. We can now send the message. mp_chatSend allows a second argument called `to`. This is the hash of the player that should recieve the message. Exactly what we need! We use a with-Block to call the `mp_chatSend` with our RPC object

```
with (obj_htme_rpc) {
    mp_chatSend(message,to);
}
```

The message is sent! Let's process it!

Recieving RPCs

What's the point of sending RPCs if we can't recieve them, right?

In the step event we created earlier we already added a call to `htmerpc_recieve` with a message as argument. Let's create the script.

```
///htmerpc_recieve(message)
/* Processes RPC calls*/
var message = htme_chatGetMessage(argument0);
var from = htme_chatGetSender(argument0);
```

Using htme_chatGetMessage we take the raw message and decode it to get the actual message. We also store the hash of the player that sent the RPC, because at the end of this script we want to send a RPC back containing the returned value of the script. But we'll come to that in a bit.

Let's decode the ds_map:

```
var rpc_command = json_decode(message);
```

`rpc_command` will now contain the ds_map we created earlier. Magic!

Let us waste no time and execute the command. This can be done by combining `asset_get_index` and `script_execute`. `asset_get_index` turns the string that contains the script name back into an id and `script_execute` executes the command using this id.

Because we also want to process the arguments, this may look a bit ridiculous now, but it works:

```
var rid = rpc_command[? "id"];
var rpc_argument_count = rpc_command[? "argument_count"];
var result;

if (rpc_argument_count == 0) {
    result = script_execute(asset_get_index(rpc_command[? "script"]));
}

if (rpc_argument_count == 1) {
    result = script_execute(asset_get_index(rpc_command[? "script"]),rpc_command[?
↪"argument0"]);
}

if (rpc_argument_count == 2) {
    result = script_execute(asset_get_index(rpc_command[? "script"]),rpc_command[?
↪"argument0"],rpc_command[? "argument1"]);
}

/** CONTINUE THIS UNTIL 14 **/

if (rpc_argument_count == 13) {
    result = script_execute(asset_get_index(rpc_command[? "script"]),rpc_command[?
↪"argument0"],rpc_command[? "argument1"],rpc_command[? "argument2"],rpc_command[?
↪"argument3"],rpc_command[? "argument4"],rpc_command[? "argument5"],rpc_command[?
↪"argument6"],rpc_command[? "argument7"],rpc_command[? "argument8"],rpc_command[?
↪"argument9"],rpc_command[? "argument10"],rpc_command[? "argument11"],rpc_command[?
↪"argument12"]);
}
```

And well... that's it. But wait - now we need to send the value that this function returned back.

Returning the value and sending it back

Everything we did so far was fine, but now what if we want to know what the script returned?

Remember the script `htmerpc_callback` we created earlier? When we are done executing the RPC script we send the returned value we stored in `result` above back to the sender of the RPC. This script will then tell the author of the original RPC the returned value.

That means:

1. Player A sends RPC to Player B to run scr_myscript which returns "Hi!".
2. Player B recieves the RPC, runs the script and stores the value in `result`.
3. Player B sends RPC to Player A to run `htmerpc_callback` with the arguments being the `id` of the original call and the `result`.
4. Player A recieves the RPC, runs the script, notices that it calls `htmerpc_callback` and therefor stops (otherwise this would be an endless loop).

Step 2 and 4 in script (add to the end of `htmerpc_recieve`):

```
//Only send returned value if this RPC isn't already about a returned value␣
↪(otherwise this would result in an endless loop)
if (rpc_command[? "script"] != "htmerpc_callback") {
    //Send returned value back via RPC
    //The id is not relevant for this because we don't track this RPC - we leave it␣
↪empty.
    htmerpc_send("",htmerpc_callback,from,rid,result);
}

//Destroy the map, we don't need it
ds_map_destroy(rpc_command);
```

The recieve-Script is now done. We now need to create `htmerpc_callback` and add a way to actually get the values.

Storing and retrieving the returned values

But before we do that, we need somewhere to store those values. Let's use our `obj_htme_rpc` for that and a ds_map. It will have the ids as keys and the returned values as values.

Add it to the create event:

```
self.returnedValues = ds_map_create();
```

Okay, so now let's create `htmerpc_callback`, this script is actually incredibly simple:

```
///htmerpc_callback(id,returnedValue)
/* Reciveves the returned value of RPCs via RPC */

ds_map_add(obj_htme_rpc.returnedValues,argument0,argument1);
```

That is all. Whenever a RPC call is sent, it will now add the returned value to the map. You can get it anywhere by using code like this:

```
///Create event of some object - You can use htme_hash() to generate random ids
self.rpc_id = htme_hash();
htmerpc_send(self.rpc_id,my_cool_script,some_player_hash,0,"Test",3+2);
```

///Step event

```
var returnedValue = ds_map_find_value(obj_htme_rpc.returnedValues,self.rpc_id);

//ds_map_find_value returns undefined if the key was not found -> if the␣
↪returnedValue has not been recieved
//Please make sure the function actually returns something and returns something␣
↪other than undefined, otherwise this code will never run.
if (!is_undefined(returnedValue)) {

    show_message(returnedValue);
    //After that make sure to delete the key if you don't need it anymore
    ds_map_delete(obj_htme_rpc.returnedValues,self.rpc_id);

}
```

That's all!
Here's a basic RPC protocol for you to improve on. Have any ideas? Be sue to leave them in our new forums!

When using this for your game, be sure to include error handlers and a secuity mechanism as explained at the top of the page.

CHAPTER 3

Concepts

Buffer types

When using mp_add you need to specify a buffer type that specifys as which datatype the information is sent. This is set for all variables in the Variable Groups.

The buffer types are the same as the ones used by Game Maker.

Strings

`buffer_string`

Do not use `buffer_text`. It's not supported.

Booleans (Reals with 0/1; false/true)

`buffer_bool`

Reals (Numbers)

From the Game Maker manual:

Type	Desciption
buffer_u8	An unsigned, 8bit integer. This is a positive value from 0 to 255.
buffer_s8	A signed, 8bit integer. This can be a positive or negative value from -128 to 127 (0 is classed as positive).
buffer_u16	An unsigned, 16bit integer. This is a positive value from 0 - 65,535.
buffer_s16	A signed, 16bit integer. This can be a positive or negative value from -32,768 to 32,767 (0 is classed as positive).
buffer_u32	An unsigned, 32bit integer. This is a positive value from 0 to 4,294,967,29 5.
buffer_s32	A signed, 32bit integer. This is a positive value from 0 to 264 - 1.
buffer_u64	An unsigned 64bit integer. This can be a positive or negative value from -(263) to 263 - 1.
buffer_f32	A 32bit float. This can be a positive or negative value within the same range as a 32 bit signed integer.
buffer_f64	A 64bit float. This can be a positive or negative value from -(263) to 263 - 1.

Floats are numbers with commas (3,2), Integers are numbers without (3). When syncing a number with commas as an Integer, the value will be rounded.

CHAT Interface

The CHAT (Custom Handler for Advanced Traffic) Interface is a new way of syncing information between players.

Instead of using objects and instances, this uses a more classic approach. Using the CHAT Interface you can send and retrieve string messages, with the support for multiple channel.

CHAT messages are sent using the **IMPORTANT** sync type.

Tutorial

- Creating a chat system with it
- Call scripts over the network using the CHAT Interface (RPC)

Commands

- mp_syncAsChatHandler for registering an object to recieve and send messages on a channel.
- mp_chatGetQueue to recieve new messages.
- mp_chatSend to send messages.
- htme_chatGetMessage to get the message out of a message.
- htme_chatGetSender to get the author of a message.

The debug overlay

The debug overlay is a new feature that allows you to do better debugging. Enable it by setting `self.debugoverlay` to true in **htme_config**.

F12: Toggle overlay

F12 will show and hide the overlay when it’s enabled.

F1: All instances

Shows a list of all instances and the variable groups for local instances.

For the first 10 instances in the list, press SHIFT + the shown number to access instance details.

Instance details

Variable	Descripti on
Hash	The id of the instance as stored by the engine
isVisible	Whether or not this is a visible instance
isNotCach ed	True if this instance is in the same room (server only; clients will only show instances in the same room)
Persisten t	See Instance scrope and rooms
stayAlive	See Instance scrope and rooms
Instance ID	The local id of the instance, assgined by GameMaker . If cached the instance will not actually exist, and therefor the instance will have no id
Object	Name of the object of this instance
Player	Number and hash of the player this instance belongs to
Saved variables	All variables as synced by the engine
Vargroups	Details of all ‘variable groups <c oncepts/v argroups> ‘__

F2: Visible instances

Same as F1 but filters by showing only visible instances

F3: Players

Shows a list of all players, including their player number and their hash. Also shows who is the local player and who runs the server.

Press SHIFT + number shown to show all instances of one player. If you are the server, press STRG + number to kick a player.

F4: Invisible instances

Same as F1 but filters by showing only invisible instances

F5: Instances in cache

Same as F1 but showing only cached instances (instances in same room; This list is always empty for clients).

F6: Global sync

Shows all variables in the global sync pool.

F7: CHAT Interface Channels

Lists all CHAT Interface channels and how many messages haven't been processed (read) and what the content of the most recent unread message is.

F8: Signed packets sent

Shows details about all signed packets that are not yet recieved by the clients/server.

F9: Signed packets inbox

Shows details about all signed packets that have been sent to the clients/server.

F10: Maps and Lists

Shows a count of all **ds_lists** and **ds_maps**. If the numbers just increase it may be a memory leak somewhere.

F11: Disconnect - Client only

Disconnect from the server. This will tell the server to kick us and then kill the engine.

Check if overlay is enabled

`htme_debugOverlayEnabled` will return true when the overlay is enabled. Use this to draw your own debug stuff when the overlay is active.

Delta time and room_speed

The engine can use `room_speed` (default) or delta time.

room_speed

The engine is set to use `room_speed` as the default timer count. You use it when you sync objects:

```
mp_addPosition("Pos",5*room_speed);
mp_add("playerName","name",buffer_string,60*room_speed);
```

In htme_config:

```
self.udphp_rctintv = 3*60*room_speed;
self.global_timeout = 5*room_speed;
self.punch_stage_timeout=1*room_speed;
self.lan_interval = 15*room_speed;
```

Room speed is simple to use, understand and works most of the time.

room_speed limits

Game Lag

If your game experience lag the room halt and so do the room speed count. Lets say that you are connected to a server and your game lag for 3 seconds. Note that the `global_timeout` is set to `5*room_speed` = 5 seconds. You and the server got each a counter running. And these counters is not synchronized. Server counter can be on 3 and yours on 1. This means that your game will disconnect if the server don't respond within 5 seconds. But it also means that the server will kick you if you don't send anything within 5 seconds. The server got his eyes on his watch and if your lag is on his 3 second count you will be kicked. Because 3+3 lag=6 seconds.

Different room speeds

Another issue is that your game must have the same room speed in all rooms. Because if you are in a room with speed 30. And the server is in a room with speed 60. Every second for you is 2 seconds for the server. So you will be kicked. Because 3*2=6. So after 3 seconds on your counter the server will be on 6 and kick you.

Delta time

Delta time can compensate for lag and different room speed. Because it counts real seconds. If your game lag for 3 seconds. The counter is still counting.

Delta time setup

You need to setup the engine to use delta time. This means that all timers must be changed to real seconds. When using room speed you set:

```
mp_addPosition("Pos",5*room_speed);
```

But you need to change it to:

```
mp_addPosition("Pos",5);
```

This set the time to 5 seconds. **Note: Sometimes you set the time just to 3 to update the position every 3 steps. But if you activate delta time this means 3 seconds now. So say your room speed is 30 you must change it to 0.1. Because 3 steps in a room with room_speed 30 is 0.1 seconds.**

But we need to set this up. First go to the script ‘‘htme_config‘‘and set:

```
use_delta_time=true;
```

You also need to remove all the `room_speed` in `htme_config`.

```
self.udphp_rctintv = 3*60;
self.global_timeout = 5;
self.punch_stage_timeout=1;
self.lan_interval = 15;
```

And go throught all your synced object and remove the `room_speed`.

```
mp_addPosition("Pos",5);
mp_add("playerName","name",buffer_string,60);
```

GMnet GATE.TESTER - The master server testing tool

GMnet GATE.TESTER allows you to test your master servers.

It's currently basicly an online lobby browser for debugging. Run your master server with the `--testing` parameter to enable debugging using this tool.

You can find it at: http://gmnet.parakoopa.de/gatetester

Local and remote Instances

When you **place an instance of an object into a room**, that is set up to be be **synced via GMnet ENGINE** (mp_sync(); called in the create-Event), you will have **one local instance of this object in your room**, **and one remote instance for each other player you are connected to**.

That means,** if you place one instance of object A in your room**, you will have** 4 instance if 4 players are playing**. Each player has **one local instance he controls**, and the **three other instances that the other players control**.

You can find out if a instance is local by using htme_isLocal(); by using this if-Statement:

```
if (htme_isLocal()) {
 //Bla
}
```

everything in the statement will only be executed once for every player, by the instance that he controls.

If you place 2 instances of an object into the room, you will have 8 instances with 4 players, of which 2 are local and the above code will be executed by these two instances.

Remember that all code outside the `htme_isLocal` will run on the remote computer. This is because when the `htme_obj_player` sync to the other computer his game will create the same object. So the create event, step event all events will run on his computer too. So you must make sure that you only run what you want on the remote computer. Ex in the `htme_obj_player` you have this code and this will run when you start the game

```
// This code will run on your computer
// *************************
self.pressed_jump = false;
self.pressed_left = false;
self.pressed_right = false;
self.name = "";
// *************************

// This code will run on your computer
// *************************
if (htme_isLocal()) {
    var ttr = totro(5,7,1);
    self.name = ttr[0];
    self.image_blend = irandom(16777215);
}
// ************************
```

The `htme_obj_player` is synced to all other player and will be created with `instance_create(x,y, htme_obj_player)`. And you know that when you create an object all events will run. So this will run on the remote computer:

```
// This code will run on the remote computer
// ************************
self.pressed_jump = false;
self.pressed_left = false;
self.pressed_right = false;
self.name = "";
// ************************

// This code will not run because htme_isLocal()=false with a remote object on a␣
↪remote computer
// ************************
if (htme_isLocal()) {
    var ttr = totro(5,7,1);
    self.name = ttr[0];
    self.image_blend = irandom(16777215);
}
// ************************
```

The other players will also sync their `htme_obj_player` to you so this will run on your computer:

```
// This code will run on your computer
// ************************
self.pressed_jump = false;
self.pressed_left = false;
self.pressed_right = false;
self.name = "";
// ************************

// This code will not run because htme_isLocal()=false on your computer with a remote␣
↪object
// Your computer is a remote computer in the other players perspective
// ************************
if (htme_isLocal()) {
    var ttr = totro(5,7,1);
    self.name = ttr[0];
    self.image_blend = irandom(16777215);
}
// ************************
```

As you can see there is a perspective. In your case all other players object is a remote object and you are local. But in all other players perspective you are the remote and they are the local.

mp_map_syncIn and mp_map_syncOut

Whenever you use mp_add, you need to do 2 things for each variable you sync with it:

1. Add the following to the end step event: `gml self.VARIABLENAME = mp_map_syncOut("VARIABLENAME", self.VARIABLENAME);` Where you replace "VARIABLENAME" with the name of your variable.
2. Run the following code whenever you change that variable, we recommend doing this in the begin step event: `gml mp_map_syncIn("VARIABLENAME",self.VARIABLENAME);` This also needs to be done at least once after you set up the engine if you don't put it in begin step.

THIS DOES NOT TO BE DONE FOR ANY OTHER FUNCTION STARTING WITH mp_add. ONLY WITH mp_add ITSELF!

Technical explanation

The engine needs to know what variables it needs to sync. So if you execute this code:

```
mp_add("message","message",buffer_string,5);
```

Then we know you want to sync the variable message. The problem/limitation in Game Maker is, that we can't get the value of the message variable of this instance now. Why?

Well let me have an example. This works:

```
var instance; //Some valid instance
var test = (instance).x;
//Test will now have the x position of the instance as value
```

Now in that case we are directly accessing the x-Position of the instance. That's what all other mp_add* functions besides mp_add itself do, because they can be hardcoded. When writing the code we already know what to sync and in what order.

We don't know how your variables are called while coding, so we have to get them dynamically. Let's take the following exmaple:

```
var instance; //Some valid instance
var variable = "x";
//What now???
var test = (instance).variable; //? Doesn't work!
var test = (instance).(variable); //? Doesn't work!
var test = (instance)[variable]; //? Doesn't work!
```

There's no way in Game Maker Studio of doing that. That's why we use a ds_map to cache your instances variables. Because with a ds_map, that does work!

```
var map; //Some valid ds_map that belongs to your instance; That's what you change␣
↪when using the syncIn and syncOut command
var variable = "x";

var test = ds_map_find_value(map,variable); //Yay! Works!
```

PLUS - Master server (GMnet GATE.PUNCH)

See this page of the tutorial:

PLUS - Hosting a master server

A detailed description of how GMnet PUNCH works, can be found in the manual of GMnet PUNCH.

Players and Playerhashes

Players are identified by hash strings.

You can loop through all players using this loop over the list returned by htme_getPlayers:

```
var playerlist = htme_getPlayers();
for(var i = 0;i<ds_list_size(playerlist);i++) {
    var player = ds_list_find_value(playerlist,i);
}
```

`player` will contain the hash of the player.
This hash can be used by htme_findPlayerInstance.

You can get the local player (your player) hash by using the variable `global.htme_object.playerhash`.

You can get a player hash from a synced object by using the objects variable `obj_name.htme_mp_player`.

Instance scope and rooms

Non-persistent instances will disappear if they leave the room. It doesn't matter if a local instance or a remote instance leaves the room, it will always disappear.

Persistent instances will follow the local player into the next room. That means persistent remote instance will be destroyed, persistent local instances not.

stayAlive instances (objects where mp_stayAlive(); was used) that are also persistent will always follow the local player into a new room, no matter if it's a local or remote instance.

Here's a table that illustrates that:

INSTANCE WAS CREATED BY PLAYER A IN ROOM A `engine/concepts/images/4.PNG`

Signed packets

Signed packets are a special type of packets that are guaranteed to arrive.

Normally when using UDP multiplayer, packets are sent very fast on the cost of reliability.

Signed packets give GMnet ENGINE the option to have packets that are sent with the reliability of TCP. These are used with the sync types SMART and IMPORTANT and some other internal operations.

You can also use them. Either by using the SMART and IMPORTANT sync types or by using them with your own packets. For how see "Extending the Engine / Advanced Use".

States of the engine

The engine can be in different states.

1. Is the engine running (=Server or Client started)? -> htme_isStarted returns true
2. Are we running the server? -> htme_isServer returns true
3. (Only client) -> Is the client connected to a server? -> htme_clientIsConnected returns true
4. (Only client) -> Did a connection to the server fail? -> htme_clientConnectionFailed returns true

VarGroup SyncTypes (mp_type)

Each Variable Group can be assigned one syncType using mp_setType that controls how the varGroup will be synchonized. They are enum values of the enum `mp_type`. FAST actually means `mp_type.FAST`! The default value, if you don't use mp_setType, is `mp_type.FAST`.

These are your options:

- **FAST**: The engine sends the variable **once** in the interval you set. Since we are using UDP for networking, there is no assurance, that it will actually arrive. That means we don't know if the other players actually get this change. The packet could get lost. However, using FAST is very fast.
- **IMPORTANT**: When using IMPORTANT, the variable will still be synced in the set interval. However, we are using a special way in *GMnet ENGINE* to make sure the packet arrives. Every X seconds, the server/client will flood the other players with the information, until they respond back, that they got it. That's how we assure the information is actually sent. This is however a pretty traffic heavy operation and using that in too short intervals will cause lagg in both connection and framerate.
- **SMART**: This is like IMPORTANT but better. Instead of syncing every X seconds, we will only sync every X seconds what has changed. This is basicly a more traffic-friendly version of IMPORTANT.

Tolerance

Using mp_tolerance you can give each variable that is a real (a number) a tolerance range in that it will not be chanegd locally. This is mainly used to make smooth movement and prevent players from jumping arround a few pixels.

Take as an example the x position:

If the actual x-Position of an instance should be 12 but it's 15 locally, the change will not be applied locally if the tolerance is greater than or equal 3.

PLUS - GMnet PUNCH

See this page of the tutorial:

PLUS - Setup GMnet PUNCH

A detailed description of how udphp works, can be found in the manual of GMnet PUNCH.

UPnP

UPnP stands for Universal Plug and Play.

GMnet ENGINE use UDP hole punching to make the NAT/router allow online connections.

UPnP automates the port forwarding on the users router or NAT. It sends a command to the router to forward a port.

But not all NATs allow UDP punch. The user must then manually port forward the server port (ex 6510) or use other methods such as UDP Hole-Punching (as done by GMnet PUNCH) that can also allow online play without UPnP or port forwarding.

By enabling UPnP you allow users who don't forward their port and that are in networks that don't support UDP Hole-Punching to play anyway.

When your users start the server, UPnP will try to port forward the server port for them. You can enable UPnP in `htme_config`.

Note: UPnP won't work 100% for everybody (even if UPnP is supported) when used with the GMNet PUNCH master server until version 1.4.0!

But there is no harm in turning it on anyway.

Variable Groups

Variable Groups or **VarGroups** describe a group of variables of an instance that are synced via the engine.

Create a new VarGroup:

- **mp_add**(groupname,variables,datatype,interval);
- **mp_addPosition**(groupname,interval);
- **mp_addBuiltinBasic**(groupname,interval);
- **mp_addBuiltinPhysics**(groupname,interval);

Change the SyncType of the group:

- **mp_setType**(groupname,syncType);

Give the group a Tolerance:

- **mp_tolerance**(groupname,tolerance);

CHAPTER 4

Functions

Sync Functions

mp_add(groupname,variables,datatype,interval)

Description

Adds a new group of variables to be synced to this instance.

Please read this manual page when using mp_add: mp_map_syncIn and mp_map_syncOut

Example

```
///Create Event
mp_sync();
mp_add("playerName","name",buffer_string,10*room_speed);;
```

Arguments

Name	type	description
group name	strin g	The name of the group, this is only used locally to identify this group, for example if you want to use mp_setType
varia bles	strin g	A list of local variables of the instance seperated with commas
datat ype	real, buffe r_*	A value of a "buffer_" constant to specify the data type of all variables in this group. See manual for a list of datatypes and their meanings. All datatypes from enum mp_buffer_ types are also allowed but should not be used by you as a user!
inter val	real	The interval in which the variable group get's synced with the other players

Returns

Nothing

mp_addBuiltinBasic(groupname,interval)

Description

Adds a new group of variables to be synced to this instance.

The variables are: * image_alpha,image_angle,image_blend,image_index,image_speed, image_xscale,image_yscale,visible

Example

```
///Create Event
mp_sync();
mp_addBuiltinBasic("drawing",10*room_speed);;
```

Arguments

Name	type	description
group name	strin g	The name of the group, this is only used locally to identify this group, for example if you want to use mp_setType
inter val	real	The interval in which the variable group get's synced with the other players

Returns

Nothing

mp_addBuiltinPhysics(groupname,interval)

Description

Adds a new group of variables to be synced to this instance.

The variables are: * direction,gravity,gravity_direction,friction,hspeed,vspeed

Example

```
///Create Event
mp_sync();
mp_addBuiltinPhysics("physics",10*room_speed);
```

Arguments

Name	type	description
group name	strin g	The name of the group, this is only used locally to identify this group, for example if you want to use mp_setType
inter val	real	The interval in which the variable group get's synced with the other players

Returns

Nothing

mp_addPosition(groupname,interval)

Description

Adds a new group of variables to be synced to this instance.

The variables are: * x,y

Example

```
///Create Event
mp_sync();
mp_addPosition("pos",10*room_speed);;
```

Arguments

Name	type	description
group name	strin g	The name of the group, this is only used locally to identify this group, for example if you want to use mp_setType
inter val	real	The interval in which the variable group get's synced with the other players

Returns

Nothing

mp_map_syncIn(varName, variable)

Description

More information: mp_map_syncIn and mp_map_syncOut

Arguments

Name	type	description
varName	string	Name of the variable to store
variable	any	Value of that variable

Returns

Nothing

mp_map_syncOut(varName, variable)

Description

More information: mp_map_syncIn and mp_map_syncOut

Arguments

Name	type	description
varNa me	strin g	Name of the variable to get
varia ble	any	Value of the variable that the instance currently has. Will later be used to compare them to values in the engine. Currently this is returned if the engine has no valid data for this varName.

Returns

Value of varName in the variable map of this instance

mp_setType(group,type)

Description

Changes the type with which a group will be synced to the other clients.

More information: VarGroup SyncTypes (mp_type)

Example

```
///Create Event
mp_sync();
mp_addBuiltinPhysics("physics",10*room_speed);
mp_setType("physics",mp_type.SMART);
```

Arguments

Name	type	description
group name	strin g	The name of the group of which you want to change the type of
inter val	real	The values of the enum mp_type can be seen in htme_config or the manual.

Returns

Nothing

mp_stayAlive()

Description

Configure this instance to be in stayAlive mode and be room independent. Instance NEEDS to be persistent!

More information: Instance scope and rooms.

Example

```
///Create Event
mp_sync();
mp_stayAlive();
```

Arguments

None

Returns

Nothing

mp_sync()

Description

Configures this instance to be synced via the GMnet ENGINE. This must be called before all other mp commands.

Example

```
///Create Event
mp_sync();
mp_addPosition("Pos",5*room_speed);
```

Arguments

None

Returns

Nothing

mp_tolerance(groupname,tolerance)

Description

More information: Tolerance.

Example

```
///Create Event
mp_sync();
mp_addPosition("pos",1);
mp_tolerance("pos",10);
```

Arguments

Name	type	description
groupname	string	The name of the group
interval	real	The tolerance to apply

Returns

Nothing

`mp_unsync()`

Description

Removes an instance from the engine and * if not [local]concepts/instances): destroys it [DO NOT USE IT ON REMOTE INSTANCES!] * if local: informs server which then informs all players.

Does NOT destroy local instances! You have to do that yourself when calling this manually!

Example

```
///Step
if (somethinghappend && htme_isLocal()) {
    mp_unsync();
    instance_destroy();
}
```

Arguments

None

Returns

Nothing

Tools

`htme_clientConnectionFailed()`

Description

Check if the client did not manage to connect in time. Only call this after creating a client, otherwise the result will be invalid.

Arguments

None

Returns

True if the engine is operating in client mode and gave up to connect.

`htme_clientDisconnect()`

Description

Disconnects from the server.

Arguments

None.

Returns

Nothing.

`htme_clientIsConnected()`

Description

Check if the client is connected.

Arguments

None

Returns

True if the engine is running in client mode and if the client successfully connected to the server.

`htme_disconnectNow()`

Description

This will close and disconnect a server or client. This will also clean the engine and you can go to the main room if the script return true. Use this when you want to exit the game and go back to the main menu.

```
if htme_disconnectNow()
{
    room_goto(rm_main);
}
```

Arguments

None

Returns

True if ok to disconnect.

`htme_findPlayerInstance()`

Description

Finds an instance created by a player. Only returns the first instance that was created by a player.

More information and examples: Players and Playerhashes

Arguments

Name	type	description
player	string	Hash identifier of the player

Returns

An instance or -1 if no instance was found.

`htme_getPlayerNumber(playerhash)`

Description

Extracts the number of the player from the playerhash The number starts at 1 for the server, 2 is the first connected client, 3 the third and so on. If 2 disconnects, the next player connecting will occupy place 2 again.

Arguments

Name	type	description
playe rhash	strin g	A playerhash as stored in the list that you can get with htme_getPla yers.

Returns

The player number

htme_getPlayers()

Description

Get a ds_list of players.

More information and examples: Players and Playerhashes

Arguments

None

Returns

a ds_list containing all player hashes.

htme_isLocal()

Description

Returns true if the instance was created locally, false if it wasn't or if it wasn't even synced with the engine.

Arguments

None

Returns

True if calling instance was created locally and is synced, false otherwise.

htme_isLostConnection()

Description

Check if the connection is lost. This will also clean the engine and you can go to the main room if the script return true.

```
if htme_isLostConnection()
{
    room_goto(rm_main);
}
```

Arguments

None

Returns

True if the connection is lost.

`htme_isServer()`

Description

Check if this engine is running in server mode.

NOTE: This will always return true when called by non local instances, this can't be used to identify if a remote instance belongs to the server!

Arguments

None

Returns

True if running in server mode, except when called by a remote instance, in that case it will always return true.

`htme_isStarted()`

Description

Check if a server or client was started

Arguments

None

Returns

True if the engine is started.

`htme_serverDisconnect(player)`

Description

Asks a client to execute htme_clientDisconnect to disconnect. If he doesn't within the timeout set in htme_config, he will be kicked by the server.

Arguments

Name	type	description
player	string	Hash identifier of the player

Returns

Nothing.

`htme_serverShutdown(player)`

Description

Shuts the server down and tells all player about it.

Arguments

None.

Returns

Nothing.

`htme_syncGroupNow()`

Description

Force syncs all or specific group names one time in the next step. If you don't provide any arguments, all groups will be force synced. Good if you have a group that syncs every 60 seconds but want to sync it earlier because you changed the value.

Arguments

None or up to 16 group names (string)

Returns

Nothing

Chat

`htme_chatGetMessage(chat_queue_entry)`

Description

Get the message of a CHAT Interface message. See mp_chatGetQueue for details.

Example

Read the tutorial about creating a chat system for more information.

Arguments

Name	type	description
chat_queue_entry	string	An entry of the queue returned by mp_chatGetQueue.

Returns

Message that was sent

`htme_chatGetSender(chat_queue_entry)`

Description

Get the sender of a CHAT Interface message. See mp_chatGetQueue for details.

Example

Read the tutorial about creating a chat system for more information.

Arguments

Name	type	description
chat_queue_entry	string	An entry of the queue returned by mp_chatGetQueue.

Returns

Playerhash of the player that sent this message

`mp_chatGetQueue()`

Description

Object has to be set up with mp_syncAsChatHandler first, otherwise an error will occur.

Get a ds_queue containing all not processed string that recieved via this channel. All entries in this queue can be decoded using these commands: * htme_chatGetSender - To get the playerhash of the player that sent this message * htme_chatGetMessage - To get the chat message.

Example

Read the tutorial about creating a chat system for more information.

Arguments

None

Returns

A ds_queue, it's entries can be decoded using the functions above.

`mp_chatSend(message, [to])`

Description

Object has to be set up with mp_syncAsChatHandler first, otherwise an error will occur.

Send message via the CHAT Interface over this chanel. The message has to be a string. You can also experiment with json to send more complicated data.

This message will by default be sent to all clients and the server [also the local game!] and can be retrieved via mp_chatGetQueue.

Use the additional 'to' argument to only send this message to one player.

Example

Read the tutorial about creating a chat system for more information.

Arguments

Name	type	description
messa ge	strin g	The message to send
[to]	strin g	(optional; Will send to all by default) hash of the player that the message should be sent to.

Returns

Nothing

`mp_syncAsChatHandler(channel)`

Description

Add this object as a handler for traffic via the CHAT Interface.

This object can then send and recieve traffic on the specified channel using mp_chatGetQueue and mp_chatSend.

This is independent from mp_sync and it's mp commands. You don't have to use mp_sync.

Example

Read the tutorial about creating a chat system for more information.

Arguments

Name	type	description
channel	string	The name of the channel to assign to this object

Returns

Nothing

Online Lobby

`udphp_downloadServerList(...)`

Description

Detailed information in 'Tutorial - Bonus 1 - An ONLINE lobby <tutorial/13_lobby>'__.

Download the list of servers from the master server.

global.udphp_downloadlist_refreshing will be set true. It will be set false again if download is finished and **global.udphp_downloadlist** will contain a list of online servers then. If download fails, global.udphp_downloadlist_refreshing will never reset. It will fail, if the master server has the lobby disabled or is not reachable.

Use the (optional) arguments to sort and filter the list. Default sorting can be seen below under arguments.

Format of **global.udphp_downloadlist**:

```
ds_list:
 [0...] => ds_map:
           [ip]    => string
           [data1] => string
           [data2] => string
           [data3] => string
           [data4] => string
           [data5] => string
           [data6] => string
           [data7] => string
           [data8] => string
           [createdTime] => string -> can be converted to real
                            unix timestamp, time the server was created.
```

Arguments

Name	type	description
[limi t]	strin g/rea l	The number of servers to return or EMPTY STRING if ALL servers should be returned (optional)
[sort by]	strin g	Field to sort the result by, this can be: date (filter by time created; DEFAULT), data1, data2, data3, data4, data5, data6, data7, data8 (optional)
[sort by_d ir]	strin g	Sort ascending (“ASC”) or descending (“DESC”; DEFAULT) (optional)
[filt er_d ata1]	strin g	Only list servers that match this excact string for their first data string (the game name). Can be EMPTY STRING if you don’t want to filter (optional)
[filt er_d ata2]	strin g	See above; but for second data string (optional)
[filt er_d ata3]	strin g	See above... (optional)
[filt er_d ata4]	strin g	See above... (optional)
[filt er_d ata5]	strin g	See above... (optional)
[filt er_d ata6]	strin g	See above... (optional)
[filt er_d ata7]	strin g	See above... (optional)
[filt er_d ata8]	strin g	See above... (optional)

Returns

Nothing

LAN Lobby

`tme_startLANsearch(port, [gamefilter])`

Description

Detailed information in ‘Tutorial - Bonus 3 - A LAN lobby <tutorial/15_lanlobby>‘__.

Search in the LAN for servers on the port specified. You can now use **htme_getLANServers()** to get a list of LAN servers. This list will be filled with servers over time. Run this comamnd again to empty it and resend the broadcast (to refresh the list).

```
Format of the list returned by **htme_getLANServers()**:
ds_list:
 [0...] => ds_map:
           [ip]    => string
           [port]  => real
           [data1] => string
           [data2] => string
           [data3] => string
           [data4] => string
           [data5] => string
```

```
            [data6] => string
            [data7] => string
            [data8] => string
```

Arguments

Name	type	description
limit	real	The port to scan on
[game filte r]	strin g	Only list servers that match this excact string for their first data string (the gamename). Can be EMPTY STRING if you don't want to filter

Returns

Nothing

Global Sync

htme_globalGet (name)

Description

Returns a variable from the global sync list. See htme_globalSet for information on what these variables are. Will return undefined if variables was not stored previously by any client or the server. Make sure the player is connected or a server is running!

More information: Bonus 2 - Global Sync (Sync a pool of variables editable by all)

Example

```
var value = htme_globalGet("name")

if (is_undefined(value) {
    //Nope, that wasn't set yet...
}
```

Arguments

Name	type	description
name	string	The name of the variable

Returns

The saved variable (real or string) or undefined

`htme_globalSet(name,value,datatype)`

Description

Stores a real or string value in the global sync list. Use buffer_type to define the type of the variable. The global sync list is a list of global variables that can be retrived via htme_globalGet.

They get synced between all clients and servers and be read and written by any (unlike instance variables which are read-only for all but the creator).

Make sure the player is connected or a server is running!

NOTE: Using this command will sync this variable immediately via SMART

More information: Bonus 2 - Global Sync (Sync a pool of variables editable by all)

Example

```
var value = 1+1;
htme_globalSet("name",value,buffer_u8);
```

Arguments

Name	type	description
name	string	The name of the variable
value	real/string	The (new) value of the variable you want to store
datatype	real	See Buffer type

Returns

Nothing

`htme_globalSetFast(name,value,datatype)`

Description

See htme_globalSet.

The only difference is that this script uses the FAST sync type. This means in rare cases when using this function the global Sync cache may be desynced, therefor only use this function in very specific cases.

More information: Bonus 2 - Global Sync (Sync a pool of variables editable by all)

Example

```
var value = 1+1;
htme_globalSetFast("name",value,buffer_u8);
```

Arguments

Name	type	description
name	string	The name of the variable
value	real/string	The (new) value of the variable you want to store
datatype	real	See Buffer type

Returns

Nothing

Events

`htme_error_message_handler(message)`

Description

You don't call this script. You edit this script to show all the error messages from the engine. You can use the existing or create your own. The script will receive the error message as an argument.

Arguments

Name	type	description
message	string	The error message

Returns

Nothing

`htme_serverEventHandlerConnecting(script)`

Description

Call a script if a new player connected. This script has to meet the following specifications:

Callback Arguments:

Name	type	description
playe r_ma p	ds_m ap	a ds_map with the keys ip and port, which contain ip and port of the player

Callback Returns:

TRUE if the connection is accepted. The engine will then register the player
FALSE if the connection should be refused, the server will abort connection

Example

See BONUS 4 - Event Handlers for Connecting/Disconnecting.

Arguments

Name	type	description
scrip t	resso urce id of a scrip t	The script to call when a player connected.

Returns

Nothing

`htme_serverEventHandlerDisconnecting(script)`

Description

Call a script if a player disconnected. This script has to meet the following specifications:

Callback Arguments:

Name	type	description
player_map	ds_map	a ds_map with the keys ip and port and hash

Callback Returns:

Nothing

Example

See BONUS 4 - Event Handlers for Connecting/Disconnecting.

Arguments

Name	type	description
scrip t	resso urce id of a scrip t	The script to call when a player disconnected .

Returns

Nothing

Config

`htme_config()`

Description

Contains the configuration and setups variables.

For more informatio see:

1. Basic configuration
2. **PLUS** - Setup GMnet PUNCH

CHAPTER 5

Extending the Engine / Advanced Use

Interpolation

You will notice that sending keystrokes are not always optimal. Your player object will do some jittering when move. And you teleport from place to place sometimes. Here is a video to show why you need interpolation:

In short interpolation means that you travel from point 1 to point 2. But not by using keystrokes. Instead we use positions and interpolate (travel smootly) from point 1 to point 2. But it can be any 2 points. It can be one x position to another x position. It can be one image_angle to another image_angle. You can interpolate any values to make a smooth look.

To code this we use the demo project as template. In this example we only interpolate the x and y position. Start in the htme_obj_player create event:

```
mp_addPosition("Pos",600*room_speed);
```

We only need to update the position once we enter the room. The interpolation will fix the rest. We are still in the create event and add this code after the `mp_sync()`:

```
self.xpos=x;
self.ypos=y;
stepsToWaitUntilSendNextPos=6;
mp_add("interpolation","xpos,ypos",buffer_s16,stepsToWaitUntilSendNextPos);
```

This will sync the position to the other clients. We also need some variables to hold some values:

```
/// Interpolation setup
// use when check if new pos received
last_received_xpos=-1;
last_received_ypos=-1;
// interpolate values each step
travel_every_step_x=-1;
travel_every_step_y=-1;
// steps we interpolate
travel_every_step_counter=-1;
```

```
// save new pos if the other interpolation is not done yet
new_queue=ds_queue_create();
```

These will keep track of the last received position. So we know if we moved. And in interpolation we move some in each step so we use varialbes to hold the travel speed in. We also want to stack the positions if receive a new position before we are done.

Now brace yourself and add this in the Step begin event:

```
if (htme_isLocal()) {
    // ====================
    // Run on THIS computer (LOCAL)
    // ====================
    // This script here only run if the instance is local (our player on this␣
↪computer)
    // The engine will create our player on the other players screens
    // So we now got our own player on this computer and the other players computers
    // We must control what we want to send to our player on the other players␣
↪computer
    // Here you should add controls over your player
    // if we got this inside if (htme_isLocal()) {
    // This will only run on your computer and not on the other players computers
    // We add some info to send to our player on the other computers
    self.xpos=x;
    self.ypos=y;
    // We will set this on every step but the engine will only send it every
    // mp_add("interpolation","xpos,ypos",buffer_s16,stepsToWaitUntilSendNextPos);
    //                                                ^
    //                                                6
    // Every 6 steps we send this info to our player on the other computers
} else {
    // ==============================
    // Run on OTHER players computers (NON LOCAL)
    // ==============================
    // This script here only run on the other players screens
    // Here you should add what will happen on other players screens
    // in the end step we use mp_map_syncOut() this is used to receive
    // information from mp_map_syncIn()
    // We now want to use the information we got from our player on our computer
    // on this player computer
    // So in the above we set
    // self.xpos=x;
    // self.ypos=y;
    // The multiplayer engine sent it to this object on the other computer
    // The engine also create a new instance of this object. And now the engine
    // wait until you send information to it from your computer. Like you xpos and␣
↪ypos
    // So what should happen on the other players screen with our xpos and ypos
    // we got from your computer?
    // We want to interpolate.
    // This will run every step but we dont get new positions every step do we
    // So we must check when new information is received.
    if self.xpos!=last_received_xpos or self.ypos!=last_received_ypos {
        // We got a different xpos or ypos value from our computer
        // Let us save this new pos in a queue
        // We always save the new pos in a queue
        // Because we might get a new pos before we are done with the first one
        ds_queue_enqueue(new_queue,self.xpos,self.ypos);
        // In a queue the values get out as they came in so if we add
        // Its like puting cards one each other and draw them from below
        // What comes in first draw first
        // Ex you enqueue number 3,8,5,2,7,4 and then dequeue 4 of them
        // you get 3 first then 8,5 and 2
        // next time you dequeue you get 7 and 4
        // Now lets save this pos as the last
        last_received_xpos=self.xpos;
        last_received_ypos=self.ypos;
    }

    // Now we check the queue if we got a new pos in it
    if ds_queue_size(new_queue)>0 {
        // First we check if we allready interpolating
        // If we do then we must wait until it's done and then we can do the new pos
        if travel_every_step_counter<1 {
            show_debug_message("x:" + string(x) + " y:" + string(y))
            // We now want to smootly move our player object on this player computer
            // to that new x pos or y pos
            // get the new x and y pos from the queue
            // we saved x first and the y so
            var newx=ds_queue_dequeue(new_queue);
            // and then get y
            var newy=ds_queue_dequeue(new_queue);
            show_debug_message("new x:" + string(newx) + " new y:" + string(newy))
            // Ok now we must move it some pixels at a time every step
            // And we know how long it took to travel this new pos
            // We sent this new info every 6 step so from x to new x it took 3 steps

            // Now we check if we got many new pos saved
            // If we got to many our player on the other computer will
            // fall back some and desync
            // So we check if we got 3 new pos then we can just skip one and travel
            // to the next
            if ds_queue_size(new_queue)>2 {
                // We got to many new pos lets take one more out
                var newx=ds_queue_dequeue(new_queue);
                var newy=ds_queue_dequeue(new_queue);
                // But now we took 2 new values
                // And we know it took 6 steps to travel to one new pos
                // so if we take out 2 new pos it must have taken 12 steps
                // to travel there. But to avoid that we fall back again
                // lets only do it in 11 steps
                // We set the stepsToWaitUntilSendNextPos in the create event
                var StepsWeTravel=(stepsToWaitUntilSendNextPos*2)-1;
            } else {
                // Set time it took to travel to this new pos
                // We only took one new pos so it took 6 steps to travel there
                var StepsWeTravel=stepsToWaitUntilSendNextPos;
            }
            // First we check the distance from current pos on this computer (x,y)
            // To your new position we sent to this copmputer (xpos, ypos)
            var distance_to_move_x=newx-x;
            var distance_to_move_y=newy-y;
            // Then we calc how much we must move to reach the new pos in the steps␣
↪we want
            travel_every_step_x=distance_to_move_x/StepsWeTravel;
            travel_every_step_y=distance_to_move_y/StepsWeTravel;
            // We want to only move our player on other computer in 6 steps (or more␣
↪if we took 2 new pos) so lets count it
            travel_every_step_counter=StepsWeTravel;
        }
    }
    // We now must handle the movement
    // First we check if we got any counts left
    if travel_every_step_counter>0 {
        // Ok we must travel some
        // Let add it to x and y on this computer
        // Remember this script will never run on your computer where you control the␣
↪player
        // Only on the other players computers
        show_debug_message("travel x:" + string(travel_every_step_x) + " travel y:" +␣
↪string(travel_every_step_y));
        x+=travel_every_step_x;
        y+=travel_every_step_y;
        show_debug_message("new x:" + string(x) + " new y:" + string(y));
        // Now we remove one count
        travel_every_step_counter-=1; // same as travel_every_step_counter=travel_
↪every_step_counter-1;
        show_debug_message("steps left:" + string(travel_every_step_counter))
        // When it reach 0 we will check if we got a new pos we can interpolate to
    }
}

// COMMON
// This will run on this computer and the other computers
// The mp_map_syncIn only run on the local. If we are on the remote it will just␣
↪ignore this
mp_map_syncIn("xpos",self.xpos);
mp_map_syncIn("ypos",self.ypos);
```

I hope you got all.

The last one is in the step end event:

```
self.xpos = mp_map_syncOut("xpos", self.xpos);
self.ypos = mp_map_syncOut("ypos", self.ypos);
```

This will now interpolate the position and make your player move smootly from one position to the next. You can also download a demo here: https://drive.google.com/file/d/0BxE4k4xEiNO2azAzNkZObUV0NnM/view?usp=sharing

CHAPTER 6

Troubleshooting

Check if client is connected

Sometimes you want to check if another player is conencted.

Use htme_getPlayers()

`htme_getPlayers()` will return a ds list with all the connected players. You can check if above 1 and then you know a client is connected.

```
var playerlist = htme_getPlayers();
var total_players=ds_list_size(playerlist);
if total_players>1 {
    show_debug_message("Another player is connected!");
}
```

Why do i control two instances

When you start two instances of your game you may experience that you control P1 and P2. This may be because of missing htme_isLocal or windows may send the controls to both games at the same time. Here are some thins you can try.

Use htme_isLocal()

You must make sure that all your keyboard and mouse inputs are inside a `htme_isLocal()` statement like this:

```
if (htme_isLocal()) {
    self.pressed_jump = keyboard_check(vk_space);
    self.pressed_left = keyboard_check(vk_left);
    self.pressed_right = keyboard_check(vk_right);
}
```

This will make sure that only the local player run the control code.

Use Dual window extension

With this extension game maker will start 2 games when you test them in game maker. When you click one window the other may not receive the controls. Import this extension: https://drive.google.com/file/d/0BxE4k4xEiNO2dEY5bUZ0dmo1b1E/view?usp=sharing And add this object in the first room: https://drive.google.com/file/d/0BxE4k4xEiNO2NW81Q1FnOTdsUUU/view?usp=sharing

Use Windows exe and game maker player export

You can also try export the game to windows exe and then run the game in game maker with game maker player export selected. This may stop windows to send the input to both games at the same time.

Error on connect

Sometimes you may experience errors when you try connect. Here are some error messages and the solutions.

Port is a real

```
ERROR in
action number 1
of Step Event0
for object obj_htme:

Illegal argument type
at gml_Script_htme_clientConnect (line 29) - network_send_udp( self.socketOrServer,␣
↪self.server_ip, self.server_port, self.buffer, buffer_tell(self.buffer) );
#######################################################################################
------------------------------------------------------------------------------------
↪------
stack frame is
gml_Script_htme_clientConnect (line 29)
called from - gml_Script_htme_step (line 59) - htme_clientConnect();
called from - gml_Object_obj_htme_StepNormalEvent_1 (line 2) - htme_step();
```

Make sure you provide a real and not a string as port argument in `htme_clientStart`.

```
htme_clientStart(ip,real(port))
```

Fire input not received on client

Sometimes you may experience that some keytrokes is missing on the client side. Ex say you use this code:

Create event:

```
self.firenow=false;
mp_add("firebutton","firenow",buffer_bool,1);
```

Step Begin event:

```
if (htme_isLocal())
{
    self.firenow= keyboard_check_pressed(vk_space);
}

mp_map_syncIn("firenow",self.firenow);
```

Step End event:

```
self.firenow=mp_map_syncOut("firenow",self.firenow);
```

This code send a network message to the other players every step. But sometime the client wont receive the message. This is because of network lag.

You send 30 network messages per sec. These messages is sent to the client. But every message take about 10-600 milliseconds to reach the other player. This means that some of the messages may be bundled and be received at the same time. Say you press fire 2 times within 1 sec. Under 1 sec the messages look like this. Every 0 is a no fire and 1 is a fire. Ever space is a step in gm. You send a perfect array of messages. 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 But the player who receives the messages may receive them like this 000 0000001 00 0000 000100 00000000 The other player received several messages in the same step. 000 is ok. 0000001 is ok because the 1 was the last message received in this step and `self.firenow` will be true when the step event run in the player. 00 and 0000 ok. But here it comes. 000100 here the other player receive 6 messages in the same step. Even if 1 fire was in there the last message received set `self.firenow` to false. So no fire this step.

Send only when needed

You can fix this by increasing the `mp_add("firebutton","firenow",buffer_bool, room_speed*100);` This will stop to send messages every step. But when you fire you need to send it. You can do this by calling `htme_syncGroupNow("firebutton")`.

if htme_isLocal() {

> If keyboard_check_pressed(ord("f")) {
>
> > firenow=true mp_map_syncIn("firenow",self.firenow); htme_syncGroupNow("firebutton");
>
> } Else if keyboard_check_released(ord("f")) {
>
> > firenow=false mp_map_syncIn("firenow",self.firenow); htme_syncGroupNow("firebutton");
>
> }

}

Player numbers

It can be tricky to get the player number. But here is how to do it.

Get current player

Use this code to get the local player number. If you are the server your number is 1. If another player connects he is number 2. If you have 3 players and p2 disconnect nr 2 is free. When someone joins he will get the free nr 2.

```
htme_getPlayerNumber(global.htme_object.playerhash);
```

Get remote player number from object

You can also get the player number from any synced object. Use this code to check what player a synced object belongs to.

```
htme_getPlayerNumber(obj_synced_player.htme_mp_player);
```

How to quit/disconnect

When you start a server or connect a client to a server and want to quit while you are in the game.

Use htme_disconnectNow()

After you call `htme_disconnectNow()` the gmnet engine will disconnect and you can go to the room you want. Make sure you remove all other network related objects after.

```
if htme_disconnectNow() {
    // If disconnect is ok then do some cleaning
    // Remove persistent but not synced objects
    with htme_obj_chat instance_destroy();
    with htme_obj_playerlist instance_destroy();
    // Go back to menu room
    room_goto(htme_rom_menu);
}
```

CHAPTER 7

How to update

When updating, check the changelog and this page to see which files have changed. When you made changes to these files, make a backup and update the asset.

After you are done updating, check what has changed (you might find detailed information in the changelog) and merge the new version with your changes.

Recent updates

1.2.2 - CHAT Interface, Event Handlers, RPCs, MIT License

- LICENSE CHANGED TO MIT Due to confusion about the GPLv3 license, I decided to change the license to the MIT license. This is way easier to understand for everybody. BY UPDATING YOU AGREE TO THIS NEW LICENSE. THIS LICENSE REPLACES THE OLD ONE.
- Support for custom Event Handler for Connection/Disconnection on Server Also adds an easy way to reject players when they conect
- Fixed a bug that prevented the game to be run with the YYC (thanks to Imtnt @gmc)
- Server Shutdown Function added and automatic shutdown/disconnect if the htme object is destroyed / game is ended
- PLUS/UDPHP : UDPHP 1.2.1 - Fixed an issue where servers and client could not connect to each other when the master server was not reachable.
- Added CHAT Interface: This is a new set of functions that allow you to send and recieve string messages in a more clasic way. The manual was updated so the Chat Tutorial now uses this new way of syncing (an RPC tutorial was also added to the manual; see below).
- Updated debug overlay to include debug information about the CHAT Interface
- Fixed a bug, where in rare cases clients could start syncing before they got their player hash which lead to invalid instances being synced and mayor desync
- MANUAL (http://htme.parakoopa.de/manual) :

- Added manual pages for the event handler scripts
- Added Tutorial Chapter Bonus 4 - Event Handlers for Connecting/Disconnecting
- Added manual page for htme_shutdown
- Added manual page for htme_globalSyncFAST
- Added manual page for the CHAT Interface and all it's functions
- Rewrote Tutorial 11 - Chat (it is now using the CHAT Interface)
- Added Tutorial Chapter Bonus 5 - RPC
- Updated debug overlay manual
- Master Server is now on Version 1.2.3, you might want to update, some critical bugfixes were made.
- **We now have a forum: http://htme.parakoopa.de/forum - Check it out - This is now the main place for support! **

1.2.1 - Introducing a better online lobby and a LAN lobby for all!

This update improves the ONLINE lobby by adding fltering and sorting mechanisms. Check out the updated tutorial! Also we added a seperate lobby for LAN Servers, the master server was HUGELY updated and we are happy to introduce a testing tool for master servers

- Added a LAN lobby.
- UDPHP/PLUS: Added a new lobby system that supports filtering
- UDPHP/PLUS: Added support for new registration and sending version number
- UDPHP/PLUS: Changed it so the server only reconnects to the master server if it lost connection
- **You need to update the master/mediation server if you are a GMnet ENGINE!!**

Master server changelog

Master server Version 1.2.0

- Added the possibility to name the server
- Added –version parameter
- Added –testing paramter to be tested via HTMT.
- Added multiple testing commands
- Added a filter for a minimum required udphp version
- Changed registration so it requires the udphp version of the server that wants to register
- Fixed bugs in server destruction
- Added a createdTime field to the servers
- Added filter and sorting methods to the lobby

CHANGED FILES IN 1.2.1:

```
scripts/htme_init.gml
objects/obj_udphphtme_lobby.object.gmx
scripts/udphp_config.gml
scripts/udphp_downloadServerList.gml
scripts/udphp_serverCommitData.gml
scripts/udphp_Punch.gml
scripts/htme_getDataServer.gml
scripts/htme_getLANServers.gml
rooms/htme_lanlobby.room.gmx
scripts/htme_networking_searchForBroadcasts.gml
objects/htme_obj_lanlobbydemo.object.gmx
objects/htme_obj_menu.object.gmx
scripts/htme_serverBroadcast.gml
scripts/htme_serverStart.gml
scripts/htme_startLANsearch.gml
scripts/htme_stopLANsearch.gml
scripts/htme_step.gml
```

1.2.0 - Debug overlay and graceful discon

I recommend to reinstall GMnet ENGINE, meaning removing the old asset and adding the new one. Make sure to backup your config.

- Added htme_serverDisconnect and htme_clientDisconnect to gracefully kick players or disconnect from the server.
- Added a debug overlay. All information about that here.
- Changed visuals of demo project
- UDPHP (PLUS only): Fixed bug that resulted in not closing old TCP connections on the servr when reconnecting.
- Added globalSetFAST. Same as globalSet but with the fast sync type.

CHANGED FILES IN 1.2.0:

```
objects/htme_obj_playerlist.object.gmx
objects/obj_htme.object.gmx
rooms/htme_rom_demo.room.gmx
rooms/htme_rom_demo2.room.gmx
scripts/htme_clientDisconnect.gml
scripts/htme_clientNetworking.gml
scripts/htme_clientShutdown.gml
scripts/htme_debugOverlayEnabled.gml
scripts/htme_debugoverlay.gml
scripts/htme_doDrawInstanceTable.gml
scripts/htme_doGlobalSync.gml
scripts/htme_doInstAll.gml
scripts/htme_doInstCached.gml
scripts/htme_doInstInvisible.gml
scripts/htme_doInstVisible.gml
scripts/htme_doMain.gml
scripts/htme_doMain_new.gml
scripts/htme_doOff.gml
scripts/htme_doPlayers.gml
scripts/htme_doSignedPackets.gml
scripts/htme_doStateInstAll.gml
```

```
scripts/htme_doStateInstCached.gml
scripts/htme_doStateInstInvisible.gml
scripts/htme_doStateInstVisible.gml
scripts/htme_doStateMain.gml
scripts/htme_doStateOff.gml
scripts/htme_do_createMicro.gml
scripts/htme_dotbd.gml
scripts/htme_findPlayerInstance.gml
scripts/htme_globalSetFast.gml
scripts/htme_globalSet_new.gml
scripts/htme_init.gml
scripts/htme_sendGSFast.gml
scripts/htme_sendGS_new.gml
scripts/htme_serverDisconnect.gml
scripts/htme_serverKickClient.gml
scripts/htme_serverNetworking.gml
scripts/htme_serverProcessKicks.gml
scripts/htme_serverStart.gml
scripts/htme_step.gml
scripts/mp_add.gml
scripts/udphp_serverPunch.gml
sprites/htme_spr_door.sprite.gmx
sprites/htme_spr_player.sprite.gmx
sprites/images/htme_spr_door_0.png
sprites/images/htme_spr_player_0.png
sprites/images/htme_spr_wall_0.png
```

1.1.0 - Introducing: Global Sync

- Added Global Sync to sync global variables that are read- and writeable at any time by all clients and the server.

CHANGED FILES IN 1.1.0:

```
htme_clientNetworking.gml
htme_clientStart.gml
htme_init.gml
htme_serverEventPlayerConnected.gml
htme_serverNetworking.gml
htme_serverStart.gml
htme_recieveGS.gml (added)
htme_sendGS.gml (added)
htme_globalGet.gml (added)
htme_globalSet.gml (added)
```

1.0.0 - The lobby update

Please see this page for more information

0.6.0 - The performance update

The great performance update! - ADDED A NEW LICENSE. BY UPDATING YOU AGREE TO THIS NEW LICENSE. – The license is still GPLv3, but it comes with an additional permission to create games in Game Maker without having you to provide the source code of your game when using the Mutliplayer Engine - Made isLocal faster

- Improved behaviour and performance of signed packets. Reliable data will no longer by sent if new data is available - Replaced most of the maps with lists, resulting in a huge performance boost! - Fixed crashes and desyncs on certain events - Fixed a mayor memory leak and improved memory managment - COMES WITH udpph 1.1.2. - No mayor changes to the local manual! - Online manual was updated with changelog.

CHANGED FILES IN 0.6.0:

```
htme_cleanUpInstance.gml
htme_clientBroadcastUnsync.gml
htme_clientStart.gml
htme_createSignedPacket.gml
htme_createSingleSignedPacket.gml
htme_forceSyncLocalInstances.gml
htme_init.gml
htme_isLocal.gml
htme_recieveSignedPackets.gml
htme_recieveVarGroup.gml
htme_removeSignedPacket.gml
htme_removeSignedPacketsByCatFilter.gml
htme_roomstart.gml
htme_sendSignedPacket.gml
htme_sendSignedPackets.gml
htme_serverBroadcastUnsync.gml
htme_serverCreateSPForAllCheckRoom.gml
htme_serverEventPlayerConnected.gml
htme_serverEventPlayerDisconnected.gml
htme_serverKickClient.gml
htme_serverNetworking.gml
htme_serverRecreateInstancesLocal.gml
htme_serverRemoveBackup.gml
htme_serverSendAllInstances.gml
htme_serverStart.gml
htme_serverSyncPlayersUDPHP.gml
htme_syncInstances.gml
htme_syncSingleVarGroup.gml
mp_add.gml
mp_unsync.gml
udphp_stopServer.gml
```

udphp CHANGELOG:

udphp 1.1.2: * Fixed stopServer not working anymore

udphp 1.1.1: * Fixed some bugs that were created with the changes in 1.1.0. ...

0.5.0

Initial release

CHAPTER 8

Getting Started

Always wanted to make a multiplayer game? But you don't want to host your own dedicated servers or want the players to open ports in their firewall? Do you just want them to simply press a button to play wih their friends?

GMnet PUNCH allows you to do the impossible! With simply calling a few scripts in your game you can get your game to the next level of multiplayer experience.

What is UDP hole punching?

UDP hole punching is a technique that allows UDP packets to be send, that normally a NAT would block. Normally if you want to host a game, you have to open the ports, because otherwise your NAT would block the UDP (or TCP) packets that are required for network communication. UDP hole punching works because in the process the client does not simply "connect" to the server. The server also sends packets back, which means both try to connect at the same time to each other. Then both NATs will accept the connection.

The problem is that at least the server doesn't know when a client connects to it. That's why we need a (globally reachable= master server. The server registers at the master server and opens a TCP connection so the master server can send the client informations*. When a client tries to connect, it also registers it's UDP port at the master server and tells it to which server it wants to connect. If the server is found, the ports will be send backd via TCP. The client get's the server port/ip and the server the client port/ip. Then both send the packets and the hole is punched. Both players are connected to each other.

(* TCP allows the master server to send packets back to the server. That's because TCP works with connections and if the server opens a connection to the master server, both can send packets to each other, even if the server is behind a NAT.)

(Source: http://en.wikipedia.org/wiki/UDP_hole_punching)

Requirements

Besides a valid **Game Maker: Studio** license you will also need to host a master server somewhere. "Somewhere" means it has to be reachable from the internet! [Ports must be opened] **Also please note that this only works with**

UDP game servers! - Setting up UDP game servers in Game Maker is very differnt from TCP game servers since UDP is connectionless. If you need help, contact me!

Wait, I need to host a "master server"?! - Where do I get that?

The server **GMnet GATE.PUNCH** that is needed to connect clients and servers is free software and the source code can be found on it's product page.

It is written in java and you can either download the binarys or the well-documented source code. Adjust the mediation server to fit your needs or write your own (you can find the communication "protocol" below) It comes with everything that is needed for TCP and UDP communication with Game Maker: Studio and can also be used to create a dedicated multiplayer server or something elese.

You need to host this server yourself. If you can't right now, contact me, I can host a server for you for testing purposes. For the demo you can also connect to the default IP and port (please note that this demo server might be down).

If you need more help setting up the server, check out the tutorial page of GMnet ENGINE on this topic.

UPnP

UPnP stands for Universal Plug and Play.

GMnet PUNCH use UDP hole punching to make the NAT/router allow online connections.

UPnP automates the port forwarding on the users router or NAT. It sends a command to the router to forward a port.

But not all NATs allow UDP punch. The user must then manually port forward the server port (ex 6510) or use other methods such as UDP Hole-Punching (as done by GMnet PUNCH) that can also allow online play without UPnP or port forwarding.

By enabling UPnP you allow users who don't forward their port and that are in networks that don't support UDP Hole-Punching to play anyway.

When your users start the server, UPnP will try to port forward the server port for them. You can enable UPnP in `udphp_config`.

Note: UPnP won't work 100% for everybody (even if UPnP is supported) when used with the GMNet PUNCH master server until version 1.4.0!

But there is no harm in turning it on anyway.

Features

- **Multiplayer without port forwarding** - Your players can enjoy gaming with their friends without having to deal with anoying firewalls
- **Builtin server and client structure that simply connects with the one you already have** - Just follow the instructions and add it to your game.
- **Easily extendable to support Peer-to-Peer gaming!**
- **Open source (GPLv3)** - You have the source code, you can modify it in any way you want and also use it as much as you want

- **Simple to use with tutorials and sample project** - You download the sample project with all required *GMnet PUNCH* scripts. You can just press play and try it out.
- **Well documented** - Not only do you get a sample project! *GMnet PUNCH* is well documented and easy to read. And since it's open source and you bought it, you can easily change it to your needs.
- **Easy to use** - For both you and the players!
- **Using a master/mediation/rendevouz server…** - You don't need to host a whole game server, you only need a small lightweight server that connects the clients to your server. More information can be found on the page Requirements.
- **Fallback mode when UDP hole punching doesn't work** - In local networks *GMnet PUNCH* will automatically try to connect directly to the server.
- **Infinite clients** - Supports infinite clients in a single game.

CHAPTER 9

Tutorial

Implementing GMnet PUNCH in your game

To implement GMnet PUNCH in your game copy all **scripts** and the behaviour of all the objects to your game and adjust them to your needs. When copying the behaviour, you can simply use your existing server/client objects and add the codes at the top of your events.

obj_control

Let's actually talk about this "behaviour": I will not go over the debug objects in the sample project, I will only guide you through the neccassry objects/code. Let's start with object **obj_control**. The only important thing in *obj_control* is the **create event**. In the create event you can find, that **udphp_config** is called. This is the **first** you have to call when you want to use GMnet PUNCH.

How to use udphp_config (arguments):

- **master_ip(stringl)** -> The IP of the master server that both server and client will use to connect to each other. Confused? See the section *Wait, I need to host a "master server"?! - Where do I get that?*.
- **master_port(real)** -> The UDP and TCP port the master server listens on.
- **reconnect intverval(real)** -> After this amount of steps the server will reconnect to the master server. This has to be done, to ensure the server is always connected to the master server. Future versions will auto-reconnect on loss of connection, but we are still testing that out.
- **connection timeout(real)** -> After this amount of time the server and client will give up to connect t eachother. The client will display a connection failed message. This timeout will also be used if you set the client to directly connect to the server.
- **debug(boolean [0/false or 1/true])** -> Show more debug messages. Not recommended for production use.
- **silent(boolean)** -> This will turn ALL debug messages off. Recommended for production use.

Copy the call of this script to your game startup or somewhere else where it get's called before the other GMnet PUNCH scripts. Make sure no global variables starting with udphp_ are used. These are reserved for *GMnet PUNCH*.

Server

Okay so we are done in obj_control. Let's go to **obj_server** and **obj_client**. Both have the same structure. *GMnet PUNCH* uses the **create event**, the **step event** and the **networking event** of both. The following scripts can jsut be inserted as [the first] entries in these events. The GMnet PUNCH scripts will only run when needed and they won't interupt your networking. If they do, please contact the support!

Server create event

obj_server's create event get's called on server creation (obviously). You need to create a **player list (ds_list)** there. This list will be used by GMnet PUNCH to store all players in. How to use them in your game will be explained later, but this is your only way to tell which player communicates with the server! You will also need a **buffer** that will be used for communication by the server. It should either be large or a growing buffer. The last thing you'll need is a **UDP server created by network_create_server(network_socket_udp,{port},{maxplayers}**. All these will be fed into the **udphp_createServer** script.

How to use udphp_createServer (arguments):

- **server** - A UDP server
- **buffer** - A buffer
- **player-list** - A ds_list that will store all players (more information below)

udphp_createServer will return either true or false. If the returned value is false, the server creation failed.

```
player_list = ds_list_create();
buffer = buffer_create(256, buffer_grow, 1);
server = network_create_server(network_socket_udp,1234,32);
ret = udphp_createServer(server,buffer,player_list);
if (!ret) {
    //Server could not be created. Destroy instace. GMnet PUNCH will also show a␣
↪message if not silent.
    instance_destroy();
}
```

Server step event

In obj_server's step event the actual magic happens. Just run **udphp_serverPunch** with no arguments.

Server networking event

And in obj_server's networking event we share the magic: Simply run **udphp_serverNetworking** with no arguem-nts.

Server: Client identification

To identify clients in the networking event, you take the **ip** and the **port**of **async_load** and check if they are in the player list you created earlier. The player list stores keys in the format "ip:port".

To see if a player is in the player list:

```
var in_ip = ds_map_find_value(async_load, "ip");
var in_port = ds_map_find_value(async_load, "port");
var exists = ds_list_find_index(player_list, in_ip+":"+string(in_port));
//exists will give you the position of the player in the list or -1 if he's not in␣
↪there
```

To get ip and port of a player entry:

```
var entry = /* Entry from the player list */
var ip = udphp_playerListIP(entry);
var port = udphp_playerListPort(entry);
```

To send a message to all players iterate over the player list and send a packet to all players (see hello world demo in step event).

Client

Client create event

To create the udphp client with **udphp_createClient** you need a **UDP socket (not server!)**, a **buffer**, as well as the **server ip**. The rest is explained in the argument listing: ***How to use udphp_createClient (arguments)***:

- **UDP client socket** - A UDP socket
- **Server IP(string)** - The IP of the server to connect to (not the master server ip!)
- **buffer** - A buffer
- **directconnect(bool)** - If you set this to true, GMnet PUNCH will be bypassed. The client willdirectly connect to the server without hole punching. directconnect is automatically set to true when the master server is down or the server is not registered at the master server
- **server_port(real)** - This is optional. Set to 0 when unkown. This will be used (and is then required for successful connection) when the client connects directly and ONLY then. See above to know when it connects directly.

IMPORTANT: The function will **return the client id you have to store**. This id is used to identify this client. Using this you can run multiple clients in one game (as used in the demo). It will return -1 if the client could not be created.

```
buffer = buffer_create(256, buffer_grow, 1);
client = network_create_socket(network_socket_udp);

client_id = udphp_createClient(client,server_ip,buffer,false,1234);
if (client_id < 0) {
    //Client could not be created. Destroy instace. GMnet PUNCH will also show a␣
↪message if not silent.
    instance_destroy();
}
`
```

Client step event

In the step event simply run **udphp_clientPunch** with the **stored client id as argument**. This function will return false if the client's connection failed. This is the only way to determine if a connection process failed and it will return true in all other cases.

```
if (!udphp_clientPunch(client_id)) {
    //When this returns false, the connection failed or the client was destroyed.
    show_message("Connection failed!");
    instance_destroy();
}
`
```

Client networking event

In the networking event simply call udphp_clientNetworking with the client id as an argument.

Client: Get IP anf Port of server / check if connected

To communicate with the server after you are connected you of course need **IP and port of the server**, you also need to know **if you are already connected**. To do that use these functions:

```
var connected = udphp_clientIsConnected(client_id); //true or false
if (connected) {
   //The next functions will return invalid information if you use them without␣
↪making sure you are connected:
   var ip = udphp_clientGetServerIP(client_id); //String
   var port = udphp_clientGetServerPort(client_id); //real
}
```

Please see the hello world demo scripts in obj_client for an example.

GMnet PUNCH is now set up!

In the next page we will show the last feature: The online lobby!

Lobby

GMnet PUNCH features a lobby system. Let me guide you through it.

First things first

Server, client and obj_control are now persistent, so they exist in the lobby room as well. There is also a new object obj_lobbyclient, which is explained later.

Testing the lobby

When you **start a client in the demo project and click yes when asked** you will be brought to the demo lobby room. This is a very simple demo room that can only show four servers. Once we are done, you'll be able to create an even better lobby.

To test it, start a server and see if it appears in the lobby. If you use the demo master server, please note that there might be "GMnet ENGINE Demo Servers" in the server list. You can not join them, they were created using the GMnet ENGINE demo project. Both use the same lobby and the same demo master server.

Data strings

There are 8 strings that can be used to identify your game in the lobby.

Open the **create** event of **obj_server** and you'll see this addition to the server creation code:

```
udphp_serverSetData(1,"udphp_demo113"); //1 is reserved for game name and version!␣
↪Change it when making your own game!
udphp_serverSetData(2,"GMnet PUNCH demo Server"); //2 is used for game name in our␣
↪demo lobby
udphp_serverSetData(3,"This is a GMnet PUNCH demo server!"); //3 is used for game␣
↪description in our demo lobby
```

These 3 strings are used by the demo project for (1) an identifier for the demo project, (2) the name of the server, (3) a description of the server. You can use all 8 for whatever you want, but (1) as a game identifier is my recommendation.

We set the data strings using `udphp_serverSetData(n,string)` right after the server was created. They are stored in the variables `global.udphp_server_data1-8`.

If you want to update any data string later, after the server connected to the master server, you need to use `udphp_serverCommitData()`. This syncs all data strings to the master server. You MUST NOT use it here, because the server hasn't connected to the master server yet!

Building the lobby

The room of the lobby is the room `udphhtme_lobby`. This room only contains the object `obj_udphphtme_lobby`. This object controls the lobby. Let's dive into it!

The create event

```
//IF YOU USE GMnet PUNCH - it will only let you connect to GMnet PUNCH servers:
if (!script_exists(asset_get_index("htme_init"))) {
   self.game = "udphp_demo120"
}
//IF YOU USE GMnet ENGINE - it will only let you conect to GMnet ENGINE servers
else {
   self.game = "htme_demo121"
}
//IF YOU USE YOUR OWN SERVER - Change self.game!

///Recieve lobby data from the master server
udphp_downloadServerList(4,"date","DESC",self.game);
```

First the variable `game` gets set. We use this to prevent GMnet ENGINE players from joining GMnet PUNCH servers and vice-versa. As said before, this object is used in GMnet ENGINE demo project and GMnet PUNCH standalone demo project, this is why it has seperate code for both.

The important part here is `udphp_downloadServerList`. This will tell GMnet PUNCH to download a list of servers from the master server. The paramters allow for filtering and sorting the result, we use this to only get results that match our game name we store in datastring 1 in this example. More information about what you can filter, can be found on the usage page of udphp_downloadServerList in the GMnet ENGINE manual.

You use your own filtering variables later when creating your lobby.

The networking event

```
///Waits for master server response
udphp_downloadNetworking();
```

This code checks if the master server sent the server list and updates it.

The draw event

The draw event is split up into different sub-scripts:

'Background', 'Title and Controls', 'Online servers'

Draws some background colors and some text, not important

'Servers (Loop)'

This draws the actual server list.

Let's analyze it:

```
///Servers (Loop)
var l = global.udphp_downloadlist;
for (var i = 0; i<4;i++) {
    draw_text(10,85+80*i,"=("+string(i+1)+")=");
```

First, the list `global.udphp_downloadlist` is stored in the local variable `l` (because it's shorter). **This ds_list contains all the servers we got from the master server**.

Then it begins a loop that loops through the first 4 servers in the list we got by the master server, everything is in this loop and then it draws a nice little number for each server.

```
if (ds_exists(l,ds_type_list)) {
    if (ds_list_size(l)>i) {
```

Now, this is the interesting part.

First we check if the downloadlist was already created (it get's created once the list has been downloaded). After that we check if it has at least as many entrys as the server we want to list. For this example we assume `i` is 1. That means it checks if there is atleast one server in the list. If yes, we have an entry we can now draw.

```
//Get stuff from the downloadlist
var entry = l[| i];
var ip = entry[? "ip"];
var game = entry[? "data1"];
var servername = entry[? "data2"];
var description = entry[? "data3"];
```

Now the entry (a ds_map) for our server is extracted from the list and we get the gamename, which is stored in data1, the ip, which is stored in the key "ip", the name of the server, which we stored in data2, and so on.

```
            draw_text(70,85+80*i,servername+" | "+ip);
            draw_text(70,115+80*i,description);
        }
    }
    draw_line(0,160+80*i,room_width,160+80*i);
}
```

Now we just draw everything.

'Footer'

Again, just some text, not important.

The press 1-4 key events

Pressing 1-4 on the keyboard will connect to that game. Let's see how!

```
///LOAD GAME SERVER ON SLOT 1
var l = global.udphp_downloadlist;
if (ds_exists(l,ds_type_list)) {
    if (ds_list_size(l)>0) {
        var entry = l[| 0];
        var ip = entry[? "ip"];
        var game = entry[? "data1"];
```

We again open the downloadlist and check if server 1 is in it, if yes we continue.

```
if (game != self.game) {
   //Not compatible game, exit
   show_message("Game server or version is incompatible!");
   exit;
}
```

Remember the filtering variable we created in the create-event? We use it here to check if the server is a GMnet PUNCH demo game. If not we cancel. Please note, that this is propably not needed here, since we filtered out all, but our game in the create event, when we ran udphp_downloadServerList.

```
        //====UDPHP DEMO ONLY
        if (!script_exists(asset_get_index("htme_init"))) {
            //Create new client - See obj_client + manual for more information
            global.tmp_lobby_ip = ip;
            instance_create(0,0,asset_get_index("obj_lobbyclient"));
            //Return to main room
            room_goto(asset_get_index("udphp_room"));
        }
        //====GMnet ENGINE DEMO ONLY
        else {
            //This code is irrelevant for UDPHP and has been removed
        }
    } else {
      //Do nothing - There is no server on this slot
    }
} else {
  //Do nothing - There is no server on this slot
}
```

This is the rest of the script. We once again check if we are running the GMnet ENGINE demo project and then we begin connection.

We create a new instance of obj_lobbyclient, a object that is child of the object obj_client.This means it has the same events. We only changed the create event to get the ip from `global.tmp_lobby_ip` rather than asking for it. It will create a new client same way `obj_client` does and control it.

Done!

And this is how you create a lobby! Now go ahead and do it! :)

How can my clients get the gamename and data strings after they connected?

Please see script `udphp_clientReadData` for how this works.

CHAPTER 10

Function reference

These are very brief and incomplete information. Detailed information can be found in the script files

The following scripts are scripts **YOU as the game developer** use:

Setup:

- **udphp_config** One-time setup script Usage: udphp_config(master_ip,master_port,reconnect_intv,timeouts,debug,silent,delta time,upnp)

Server:

- **udphp_createServer** To be used in the Create event of the server Usage: udphp_createServer(udp_server,buffer,player_list,upnp port)
- **udphp_serverPunch** To be used in the step event of the server Usage: udphp_serverPunch()
- **udphp_serverNetworking** To be used in the networking event of the server Usage: udphp_serverNetworking()
- **udphp_stopServer** Usage: udphp_stopServer() Delete the instance afterwards

Client:

- **udphp_createClient** To be used in the Create event of the client Note: Returns client id Usage: udphp_createClient(udp_socket,server_ip,buffer,directconnect,directconnect_port)
- **udphp_clientPunch** To be used in the step event of the client Usage: udphp_clientPunch(id)
- **udphp_clientNetworking** To be used in the networking event of the client Usage: udphp_clientNetworking(id)
- **udphp_stopClient** Usage: udphp_stopClient(client_id) Instance should be deleted with the false return value of clientPunch

Tools:

- **udphp_clientGetServerIP** [for client] This will return the server ip of this client and should only be used if the client is connected. Usage: udphp_clientGetServerIP(client_id)
- **udphp_clientGetServePort** [for client] This will return the server port of this client and should only be used if the client is connected. Usage: udphp_clientGetServerPort(client_id)
- **udphp_playerListIP** [for server] Get an ip out of a player list entry (see serverCreate for details) Usage: udphp_playerListIP(player)
- **udphp_playerListPort** [for server] Get an port out of a player list entry (see serverCreate for details) Usage: udphp_playerListPort(player)
- **udphp_clientIsConnected** [for client] With this you can check if your client has conncted to the server. Usage: udphp_clientIsConnected(client_id)

Lobby:

- **udphp_downloadServerList** [no server or client has to be running] Download a list of servers from the lobby. More information. Usage: See GMnet ENGINE manual page

CHAPTER 11

GMnet PUNCH/GATE.PUNCH “protocol” (for building your own server)

This is a linear breakdown of what packets are sent:

SERVER:
-> Sends UDP packet to master server containing the string “reg” with the line feed character (unicode 10) at the end
-> (Re-)connects via TCP to master server
-> Sends TCP packet to master server containing the string “reg” with the line feed character (unicode 10) at the end
-> Repeats every X minutes [alternative smethod is to reconnect when TCP socket is closed]

MASTER SERVER:
<- When recieving the UDP packet with “reg” the UDP port for that server get’s saved
<- When recieving the TCP packet with “reg” this connection socket is saved so the master server can use it later to communicate with the server

CLIENT:
-> Sends UDP packet to master Server containing the string “connect” with the line feed character (unicode 10) at the end
-> Opens TCP connection to master server
-> Sends TCP packet to master Server containing the string “connect”+line feed+IP of the server we want to connect to+line feed

MASTER SERVER: <- When recieving UDP packet with “connect” the UDP port for that client get’s saved
<- When recieving TCP packet that has “connect” in the first line (unicode 10 is used as a newline).

- WHEN FOUND: -> Via the client’s TCP socket: Send UDP ip&port of server to client [packet: buffer_s8 (-1), buffer_string (ip), buffer_string (port)] -> Via the server’s TCP socket: Send UDP ip&port of the client to server [packet: buffer_s8 (-1), buffer_string (ip), buffer_string (port)]

- WHEN NOT FOUND: -> Send not found packet to client via TCP [packet: buffer_s8 (-2)]

CLIENT: <- Wait for packet of master server

- -> When id -2 (not found): Connect directly to the server
- -> When id -1 (found): Save the two strings, convert the port to a real and connect to server via UDP and the recieved ip&port (Knock-Knock) [packet: buffer_s8 (-3)]

SERVER: <- Wait for packet of master server

- -> When id -1 (found): Save the two strings, convert the port to a real and connect to client via UDP and the recieved ip&port (Knock-Knock) [packet: buffer_s8 (-3)]

<- Wait for incoming client Knock-Knock packet (-3):

- -> When not already connected with this client: add it to the connected clients and send a welcome packet via UDP back to the client that sent this packet [packet: buffer_s8 (-4)}

CLIENT:
-> When recieving the welcome packet: Mark client as connected.
-> When not recieving a welcome packet after a timeout: Tell the player the connection failed

Without hole punching the client simply sends the Knock-Knock packets to the server and waits for the servers welcome message.

CHAPTER 12

GMnet GATE PUNCH Manual

You can find the GMnet GATE PUNCH manual here. Please note, that the GMnet GATE PUNCH manual is outdated, as GATE PUNCH will soon be replaced with a new, rewritten, master server

CHAPTER 13

Support

In case the manual couldn't help you, feel free to go to our forums, we will help you :)

CHAPTER 14

Contribute to this manual

This manual is open source. If you want to contribute to it, visit our Github repository.