

# Tutorial: Creating Multiplayer Games

*Copyright 2003, Mark Overmars*

*Last changed: March 31, 2003 (finished)*

*Uses: version 5.0, advanced mode*

*Level: Advanced*

Playing games against the computer is fun. But playing games against other human players can be even more fun. It is also relatively easy to make such games because you don't have to implement complicated computer opponent AI. You can of course sit with two players behind the same monitor and use different keys or other input devices, but it is a lot more interesting when each player can sit behind his own computer. Or even better, one player sits on the other side of the ocean. For this *Game Maker* has multiplayer support. This tutorial explains you how to use it. Be warned though. Creating effective multiplayer games is not easy. It requires that you are an experienced user of *Game Maker*. You will need to understand the use of some code. So don't let this be the first game you develop.

In this tutorial we will create a simple two-player pong game and a little chat program. The emphasis is not on nice graphics or fancy game play but only on the multiplayer aspects. You can use it as a basis for a more fancy game. We will treat the following aspects:

- Setting up a connection to a different computer
- Creating or joining a game session
- Keeping the games synchronized

The last part is the most difficult part of every multiplayer game. The problem is how to make sure that both players have exactly the same view on the game. For example, in the pong game, both players should see the ball at exactly the same place. *Game Maker* will provide the tools to do this but you will have to design the communication yourself for each game you make.

## Setting up a connection

The standard way in which a multiplayer game works is as follows. Each player runs a copy of the game. They do though run in different modes. One player runs his or her game in a server mode. The others run the game in a client mode. The server should start the game first and creates the game session. The others can then join this session to join the game. The players must decide on the mechanism used for communicating between the computers. On a local area network, the easiest is to use an IPX connection (see below for more details). If all players are connected to the Internet TCP/IP is normally used. In this protocol the clients must know the IP address of the server. So the player running the game in server mode must give his IP address to the other players (for example by sending them an email). You can find your IP address by using the program called winipcfg.exe in your windows directory. You can also use the *Game Maker* function `mplay_ipaddress()` for this. A more old-fashioned way of connecting is using a modem connection (in which case the client must know the phone number of the server and provide this) or using a serial line.

Please realize that communication is getting more difficult now that people use firewalls and routers. These tend to block messages and convert IP addresses. If you have problems setting up a connection this might be the reason. Best first test with some commercial game whether the connection can be made.

So for two computer to communicate they will need some connection protocol. Like most games, *Game Maker* offers four different types of connections: IPX, TCP/IP, Modem, and Serial. The IPX connection (to be more precise, it is a protocol) works almost completely transparent. It can be used to play games with other people on the same local area network. The protocol needs to be installed

on your computer to be used. (If it does not work, consult the documentation of Windows. Or go to the Network item in the control panel of Windows and add the IPX protocol.) TCP/IP is the internet protocol. It can be used to play with other players anywhere on the internet, assuming you know their IP address. On a local network you can use it without providing addresses. A modem connection is made through the modem. You have to provide some modem setting (an initialization string and a phone number) to use it. Finally, when using a serial line (a direct connection between the computers) you need to provide a number of port settings. There are four GML functions that can be used for initializing these connections:

- `mplay_init_ipx()` initializes an IPX connection.
- `mplay_init_tcpip(addr)` initializes a TCP/IP connection. `addr` is a string containing the web address or IP address, e.g. 'www.gameplay.com' or '123.123.123.12', possibly followed by a port number (e.g. ':12'). Only when joining a session (see below) you need to provide an address. The person that creates the session does not need to provide an address (because the address of his computer is the one that matters.) On a local area network no addresses are necessary, but you still need to make the call.
- `mplay_init_modem(initstr,phonenr)` initializes a modem connection. `initstr` is the initialization string for the modem (can be empty). `phonenr` is a string that contains the phone number to ring (e.g. '0201234567'). Only when joining a session (see below) you need to provide a phone number.
- `mplay_init_serial(portno,baudrate,stopbits,parity,flow)` initializes a serial connection. `portno` is the port number (1-4). `baudrate` is the baudrate to be used (100-256K). `stopbits` indicates the number of stopbits (0 = 1 bit, 1 = 1.5 bit, 2 = 2 bits). `parity` indicates the parity (0=none, 1=odd, 2=even, 3=mark). And `flow` indicates the type of flow control (0=none, 1=xon/xoff, 2=rts, 3=dtr, 4=rts and dtr). Returns whether successful. A typical call is `mplay_init_serial(1,57600,0,0,4)`. Give 0 as a first argument to open a dialog for the user to change the settings.

Your game should call one of these functions exactly once. All functions report whether they were successful. They are not successful if the particular protocol is not installed or supported by your machine.

So the first room in our game should show the four possibilities and let the player pick one. (Or only allow for those protocols that you want. The last two might be too slow for your game.) We call the initialization function in the mouse event and, if successful, go to the next room. Otherwise we give an error message. So in the mouse event of the IPX button we place the following piece of code:

```
{
  if (mplay_init_ipx())
    room_goto_next()
  else
    show_message('Failed to initialize IPX connection.')
}
```

When the game ends, or when the game no longer wants to use the multiplayer facility, you should use the following routine to end it:

- `mplay_end()` ends the current connection. Returns whether successful.

You should also call this routine before you want to make a new, different connection.

## Game sessions

When you connect to a network, there can be multiple games happening on the same network. We call these sessions. These different sessions can correspond to different games or to the same game.

A game must uniquely identify itself on the network. Fortunately, *Game Maker* does this for you. The only thing you have to know is that when you change the game id in the options form this identification changes. In this way you can avoid that people with old versions of your game will play against people with new versions.

If you want to start a new multiplayer game you need to create a new session. For this you can use the following routine:

- `mplay_session_create(sesname,playnumb,playname)` creates a new session on the current connection. `sesname` is a string indicating the name of the session. `playnumb` is a number that indicates the maximal number of players allowed in this game (use 0 for an arbitrary number). `playname` is the name of you as player. Returns whether successful.

In many cases the player name is not used and can be an empty string. Also, the session name is only important if you want to give people the option to choose the session they want to join.

So one instance of the game must create the session. The other instance(s) of the game should join this session. This is slightly more complicated. You first need to look what sessions are available and then choose the one to join. There are three routines important for this:

- `mplay_session_find()` searches for all sessions that still accept players and returns the number of sessions found.
- `mplay_session_name(numb)` returns the name of session number `numb` (0 is the first session). This routine can only be called after calling the previous routine.
- `mplay_session_join(numb,playname)` makes you join session number `numb` (0 is the first session). `playname` is the name of you as a player. Returns whether successful.

So what you standard do is call `mplay_session_find()` to find all existing sessions. The you either repeatedly use `mplay_session_name()` to show them to the player and let him make a choice, or you immediately join the first session. (Note that finding the session takes a bit of time. So don't call this routine in each step.)

A player can stop a session using the following routine:

- `mplay_session_end()` ends the session for this player.

It is useful to first notify the other player(s) of this but this is not strictly necessary.

So in our game, the second room gives the user two choices: either to create a new session, or to join an existing session. For the first choice we perform the following code in the mouse event:

```
{
  if (mplay_session_create('',2,''))
  {
    global.master = true;
    room_goto_next();
  }
  else
    show_message('Failed to create a session.')
}
```

Note that we set a global variable `master` to true. The reason is that in the game we want to make a distinction between the main player (called the master) and the second player (called the slave). The master will be responsible for most of the game play while the slave simply follows him.

The second choice is to join an existing game. Here the code looks as follows.

```
{
  if (mplay_session_find() > 0)
  {
    if (mplay_session_join(0, ''))
    {
      global.master = false;
      room_goto_next();
    }
    else
      show_message('Failed to join a session.')
  }
  else
    show_message('No session available to join.')
}
```

So in this game we simply join the first session that is available. Because we indicated that the maximal number of players is 2, no other player can join the session anymore.

## ***Dealing with players***

Once the master created a session, we have to wait for another player to join. There are three routines that deal with players.

- `mplay_player_find()` searches for all players in the current session and returns the number of players found.
- `mplay_player_name(num)` returns the name of player number `num` (0 is the first player, which is always yourself). This routine can only be called after calling the previous routine.
- `mplay_player_id(num)` returns the unique id of player number `num` (0 is the first player, which is always yourself). This routine can only be called after calling the first routine. This id is used in sending and receiving messages to and from individual players.

In our third room we simply wait for the second player to join. So we put some object there and in the step event we put:

```
{
  if (mplay_player_find() > 1)
    room_goto_next();
}
```

(We actually don't need to go to this room for the slave.)

## ***Synchronizing actions***

Now that we set up the connection, created a session, and have two players in it, the real game can begin. But this also means that the real thinking about communication must begin. The main problem in any multiplayer game is synchronization. How do we make sure that both players see exactly the same picture of the game world? This is crucial. When one player sees the ball in our game at a different place than the other player, strange things can happen. (In the worst case, for one player the ball is hit with the bat while for the other player the ball is missed.) The games easily get out of sync, which creates havoc in most games.

What is worse, we have to do this with a limited amount of communication. If you are e.g. playing over a modem, connections can be slow. So you want to limit the amount of communication as much as possible. Also there might be delays in when the data arrives on the other side. Finally, there is even the possibility that data gets lost and never arrives on the other end.

How to best handle all these problems depends on the type of game you are creating. In turn-based games you will probably use a rather different mechanism than in high-speed action games.

*Game Maker* offers two mechanisms for communication: shared data and messages. Shared data is the easiest mechanism to use. Sending messages is more versatile but requires that you understand better how communication works.

## **Shared data communication**

Shared data communication is probably the easiest way to synchronize the game. All communication is shielded from you. There is a set of 10000 values that are common to all entities of the game. Each entity can set values and read values. *Game Maker* makes sure that each entity sees the same values. A value can either be a real or a string. There are just two routines:

- `mplay_data_write(ind, val)` write value `val` (string or real) into location `ind` (`ind` between 0 and 10000).
- `mplay_data_read(ind)` returns the value in location `ind` (`ind` between 0 and 10000). Initially all values are 0.

To use this mechanism you have to determine which value is used for what, and who is allowed to change it. Preferably, only one instance of the game should write the value.

In our case there are 4 important values: the y-position of the masters bat, the y-position of the slaves bat, and the x and y position of the ball. So we use location 1 to indicate the y-coordinate of the masters bat, location 2 to indicate the y-coordinate of the slave bat, etc. It is clear that the master writes the values of his bat and the slave writes the values of the slaves bat. We decide that the master is responsible for the values of the ball. The slave simply draws the ball in the correct position in each step.

How do we put this into work? First of all, the master bat (being the left one) should be controlled by the master only. This means that in the up and down arrow keyboard events that control it we should make sure that the position only changes if we are the master. So the code for the up arrow key could be something like:

```
{
    if (!global.master) exit;
    if (y > 104) y -= 6;
    mplay_data_write(1,y);
}
```

For the slave bat we write a similar piece of code. Now both the master and the slave must make sure that they place the bat of the other at the correct position. We do this in the step even. If we are the slave, in the step event of the master bat we must set the position. So here we use the following code:

```
{
    if (global.master) exit;
    y = mplay_data_read(1);
}
```

Similar for the slaves bat.

Finally, for the ball we first of all must make sure that it bounces around when it hits the walls and the bats. Actually, only for the master this must be done. To communicate the position of the ball to the slave, add the following code in the step event:

```
{
```

```

if (global.master)
{
    mplay_data_write(3,x);
    mplay_data_write(4,y);
}
else
{
    x = mplay_data_read(3);
    y = mplay_data_read(4);
}
}

```

With this the basic communication is done. What remains is the handling of the start of the game, the scoring of points, etc. Again, all this is best left purely under control of the master. So the master decides when a player loses, changes the score (for which we can use an extra location to communicate it to the other), and lets a new ball start. You can check out the enclosed game `pong1.gmd` for details.

Note that with the communication scheme described, the two games can still be a bit out of sync. This is normally not a problem. You can avoid this by having some synchronization object that uses some values to make sure that both sides of the game are ready before anything is drawn on the screen. This should be used with care though because it might cause problems, like deadlock in which both sides wait for the other.

Realize that when a player joins a game later the changed shared values are NOT send to the new player (this would take too much time). So only new changes after that moment are send to the new player.

## Messaging

The second communication mechanism that *Game Maker* supports is the sending and receiving of messages. A player can send messages to one or all other players. Players can see whether messages have arrived and take action accordingly. Messages can be sent in a guaranteed mode in which you are sure they arrive (but this can be slow) or in a non-guaranteed mode, which is faster.

We will first use messages to add some sound effects to our game. We need a sound when the ball hits a bat, when the ball hits a wall, and when a player wins a point. Only the master can detect such events. So the master must decide that a sound must be played. It is easy to do this for its own game. It can simply play the sound. But it must also tell the slave to play the sound. We could use shared data for this but that is rather complicated. Using a message is easier. The master simply sends a message to the slave to play a sound. The slave listens to the messages and plays the correct sound when asked to do so.

The following messaging routines exist:

- `mplay_message_send(player,id,val)` sends a message to the indicated player (either an identifier or a name; use 0 to send the message to all players). `id` is an integer message identifier and `val` is the value (either a real or a string). The message is sent in non-guaranteed mode.
- `mplay_message_send_guaranteed(player,id,val)` sends a message to the indicated player (either an identifier or a name; use 0 to send the message to all players). `id` is an integer message identifier and `val` is the value (either a real or a string). This is a guaranteed send.
- `mplay_message_receive(player)` receives the next message from the message queue that came from the indicated player (either an identifier or a name). Use 0 for messages from any player. The routine returns whether there was indeed a new message. If so you can use the following routines to get its contents:

- `mplay_message_id()` Returns the identifier of the last received message.
- `mplay_message_value()` Returns the value of the last received message.
- `mplay_message_player()` Returns the player that sent the last received message.
- `mplay_message_name()` Returns the name of the player that sent the last received message.
- `mplay_message_count(player)` Returns the number of messages left in the queue from the player (use 0 to count all message).

A few remarks are in place here. First of all, if you want to send a message to a particular player only, you will need to know the players unique id. As indicated earlier you can obtain this with the function `mplay_player_id()`. This player identifier is also used when receiving messages from a particular player. Alternatively, you can give the name of the player as a string. If multiple players have the same name, only the first will get the message.

Secondly, you might wonder why each message has an integer identifier. The reason is that this helps your application to send different types of messages. The receiver can check the type of message using the id and take appropriate actions. (Because messages are not guaranteed to arrive, sending id and value in different messages would cause serious problems.)

For the playing of sounds we do the following. When the master determined that the ball hits the bat it executes the following piece of code:

```
{
    if (!global.master) exit;
    sound_play(sound_bat);           // play the sound yourself
    mplay_message_send(0,100,sound_bat); // send it to the slave
}
```

The controller object in the step event, does the following:

```
{
    while (mplay_message_receive(0))
    {
        if (mplay_message_id() == 100) sound_play(mplay_message_value());
    }
}
```

That is, it checks whether there is a message and if so checks to see what the id is. If this is 100 it plays the sound that is indicated in the message value.

More general, your game typically has a controller object in your rooms that, in the step event, does something like:

```
{
    while (mplay_message_receive(0))
    {
        from = mplay_message_player();
        name = mplay_message_name();
        messid = mplay_message_id();
        val = mplay_message_value();
        if (messid == 1)
        {
            // do something
        }
        else if (messid == 2)
        {
            // do something else
        }
        // etc.
    }
}
```

```
}
```

Carefully designing the communication protocol used (that is, indicating which messages are sent by who at what moments, and how the others must react to them) is extremely important. Experience and looking at examples by others helps a lot.

## ***Dead-reckoning***

The pong game described above has a serious problem. When there is a hiccup in communication the ball of the slave will temporarily stand still. No new coordinate positions arrive and, hence, it does not move. This problem occurs in particular when the distance between the computers is large and/or the communication is slow. When games get more complex, you will need more values to describe the state of the game. When you change a lot of values in each step, a lot of information must be transmitted. This can cost a lot of time, slowing your game down or making things go out of sync.

A first way to make the communication of shared data a bit faster is to no longer demand guaranteed communication of the data. This can be achieved by using the function:

- `mplay_data_mode(guar)` sets whether or not to use guaranteed transmission for shared data. `guar` should either be true (the default) or false.

A better technique used to remedy this problem is called dead-reckoning. Here we send information only from time to time. In between the game itself guesses what is happening based on the information it has.

We will now use this for our pong game. Rather than sending the ball position in each step we also send information about the balls speed and direction. Now the slave can do most of the calculations itself. As long as no new information arrives from the master, it simply computes where the ball moves.

We will not use shared data in this case. Instead we use messages. We use messages that indicate a change in ball position, ball speed, bat position, etc. The master sends such messages whenever something changes. The controller object in the slave listens to these messages and sets the right parameters. But if the slave receives nothing it still lets the ball move. If it makes a slight mistake, a later message from the master will correct the position. So for example, in the step event of the ball we put the following code:

```
{
    if (!global.master) exit;
    mplay_message_send(0,11,x);
    mplay_message_send(0,12,y);
    mplay_message_send(0,13,speed);
    mplay_message_send(0,14,direction);
}
```

In the step event of the controller object we have the following code:

```
{
    while (mplay_message_receive(0))
    {
        messid = mplay_message_id();
        val = mplay_message_value();
        // Check for bat changes
        if (messid == 1) bat_left.y = val;
        if (messid == 2) bat_right.y = val;
        // Check for ball changes
        if (messid == 11) object_ball.x = val;
```



```

    if (messid == 12) object_ball.y = val;
    if (messid == 13) object_ball.speed = val;
    if (messid == 14) object_ball.direction = val;
    // Check for sounds
    if (messid == 100) sound_play(val);
}
}

```

Note that the messages don't need to be sent in a guaranteed mode. If we miss one from time to time this is not a serious problem. You can find the adapted game in the file `pong2.gmd`.

Now you will be disappointed when you run game `pong2`. There are still hiccups. What causes these? The reason is that transmission might be slow. This means that the slave might receive messages that were sent a while back. As a result it will set the ball position back a bit and then, when it receives the new messages, sets it forward again. So let us do a third try, which you can find in the file `pong3.gmd`. This case we only exchange information when the ball hits a bat. So the whole rest of the motion is done using dead-reckoning. The master is responsible what happens at the side of the master's bat and the slave is responsible for what happens at the other side. While the ball moves from one side to the other no messages are exchanged anymore.

As you will see the ball moves smoothly now. Only when it hits a bat a short hick-up can occur or the ball might start moving before it reaches the opponents bat. The reason is that this mechanism assumes that both games run at exactly the same speed. If one of the computers is slow this might cause a problem. But a hick-up at a bat is much more acceptable than a hick-up during the motion. To avoid this last type of problem you need more advanced mechanisms. For example, you can send timing information such that each side knows how fast the game is running on the other computer and can make corrections accordingly.

I hope you by now understand how difficult synchronization is. You might also start appreciating how commercial games achieve this. They have to deal with exactly the same problems.

## ***A chat program***

For our second demo we make a little chat program. Here we will allow an arbitrary number of players. Also we will allow for multiple sessions and give the player the choice to pick a particular session. We will use messages with strings to send the text typed around.

I used a slightly more complicated mechanism to make the connection. The first room now has four choices for the four different types of connections. But it does not make the connections. Instead it only fills in a global variable `connecttype`. In the second room the player can again choose whether to create a game (or actually a chatbox) or join one. Only now is the connection initialized. Depending on whether the player creates or joins a game some questions are asked.

After the connection is made successfully, the session is created or joined. This time the player is asked for his/her name such that players can be identified. The join part is a bit more complicated this time. We construct a menu of all different session available from which the player can choose one.

Normally, when the game that created the session ends, the session ends. For a chat program this is probably not what you want. The other players should be able to continue chatting. This can be changed using the function:

- `mplay_session_mode(move)` sets whether or not to move the session host to another computer when the host ends. `move` should either be true or false (the default).

The whole mechanism of the first two rooms you will probably want to reuse for your games because it is often largely the same.

After this we move to the chatbox room. There is just one controller object that does all the work here. I will not explain the details of letting the user type in the lines of text and displaying the last couple of lines. It is all put in some scripts that you can reuse if you want. It is all rather straightforward (when you know how to program in GML). The only part that is still interesting is that whenever the player presses the enter key, the typed line is sent to all other players, with the player's name in front of it. Also when a player joins or quits, he sends a message to all other players indicating what he did.

Look at the file `chat.gmd` for details.

## **Conclusion**

The multiplayer facilities in *Game Maker* make it possible to create fancy multiplayer games. The functions though only help you with the low-level communication. You yourself have to design the communication mechanism used. This is a careful process. It should be designed while you design the game. It is very difficult to add effective multiplayer later. Here are some global guidelines:

- For most simple game a master-slave mechanism is easiest to make
- Carefully determine who is responsible for what data
- Use dead-reckoning whenever possible
- Try to rely as little as possible on guaranteed communication