

Tutorial: Multiplayer Spiele erstellen

Copyright Original: 2003-2004, Mark Overmars

Copyright Übersetzung: 2004, Windapple

Letzte Änderung am Original: 1. September 2004

Letzte Änderung an Übersetzung: 11. Dezember 2004

Verwendet: Version 6.0, erweiterter Modus, registrierte Version

Niveau: Fortgeschritten

Spiele gegen den Computer zu spielen macht Spaß. Aber Spiele gegen andere menschliche Spieler zu spielen macht noch viel mehr Spaß. Es ist sogar recht einfach ein solches Spiel zu machen, da du keine komplizierte Computergegner KI entwickeln musst. Du kannst natürlich mit 2 Spielern von dem selben Monitor sitzen und verschiedene Tasten oder Eingabegeräte nutzen, aber es ist viel interessanter, wenn jeder Spieler vor seinem eigenen Monitor sitzen kann. Oder noch besser, ein Spieler sitzt am anderen Ende des Ozeans. Dafür hat *Game Maker* Multiplayer Unterstützung. Dieses Tutorial beschreibt, wie man sie benutzt. Sei aber gewarnt. Effektive Multiplayer Spiele zu entwickeln ist nicht einfach. Dazu solltest du schon ein erfahrener *Game Maker* Benutzer sein. Du musst einigen Code verstehen können. Lass deswegen dies nicht dein erstes Spiel sein. Auch brauchen diese Funktionen die registrierte Version des *Game Maker*.

In diesem Tutorial werden wir ein einfaches 2-Spieler Pong Spiel und ein kleines Chat-Programm entwerfen. Der Schwerpunkt liegt nicht bei schönen Grafiken oder ausgeflipptem Gameplay sondern nur bei den Multiplayer-Aspekten. Du kannst es als Basis für ein ausgebuffteres Spiel verwenden. Wir behandeln die folgenden Themen:

- Eine Verbindung zu einem anderen Computer aufbauen
- Eine Spiel-Session erstellen oder ihr beitreten
- Die Spiele synchron halten

Der letzte Teil ist das schwierigste an Multiplayer Spielen. Die Herausforderung ist, sicherzustellen, dass beide Spieler exakt dasselbe sehen. Beispielsweise sollte bei dem Pong Spiel der Ball bei beiden Spielern an genau derselben Stelle sein. *Game Maker* unterstützt dich mit Hilfsmitteln dabei, aber du musst die Kommunikation für jedes Spiel selbst entwickeln.

Eine Verbindung zu einem anderen Computer aufbauen

Die normale Arbeitsweise eines Multiplayer Spieles ist die folgende. Jeder Spieler hat eine Kopie des Spieles laufen. Diese laufen aber in unterschiedlichen Modi. Ein Spieler lässt sein Spiel im Server Modus laufen. Der andere spielt das Spiel im Client Modus. Der Server muss das Spiel als erstes starten und eine Session anlegen. Die anderen können der Session beitreten um dem Spiel beizutreten. Der Spieler muss entscheiden, welchen Mechanismus er zum Verbinden verwendet. In einem lokalen Netzwerk ist es das einfachste, eine IPX Verbindung zu benutzen (weiter unten folgend Details). Wenn alle Spieler über das Internet spielen wird normalerweise TCP/IP verwendet. In diesem Protokoll müssen die Spieler die IP Adresse des Servers kennen. Deshalb muss der Betreiber des Server die IP Adresse auf irgendeinem Weg an die anderen Spieler weitergeben (beispielsweise per Email). Du kannst deine IP z.B. mit dem Programm winipcfg.exe in deinem Windows-Verzeichniss herausfinden. Du kannst auch die *Game Maker* Funktion `mplay_ipaddress()` dafür verwenden. Eine altmodischere Verbindungsvariante ist die Verbindung per Modem (in diesem Falle muss der Client die Telefonnummer des Servers kennen) oder per serielltem Kabel.

Bitte beachte, dass die Kommunikation heute viel schwerer geworden ist, weil die Leute Firewalls und Router einsetzen. Diese tendieren zum blockieren und Nachrichten und ändern die IP Adresse. Wenn du beim Aufbau einer Verbindung Probleme hast, könnte dies daran liegen. Am besten testest du erst mit einem kommerziellen Spiel, ob eine Verbindung aufgebaut werden kann. Siehe ??? für mehr Informationen zur Umgehung dieses Problems. Beachte auch, dass die interne IP Adresse anders als die Externe sein kann, bedingt durch beispielsweise einen Router.

Zwei Computer, die kommunizieren wollen, brauchen ein Protokoll. Wie die meisten Spiele bietet *Game Maker* vier verschiedene Verbindungstypen an: IPX, TCP/IP, Modem und Seriell. Die IPX Verbindung (um genauer zu sein, es ist ein Protokoll) arbeitet fast komplett transparent. Sie kann genutzt werden, um mit anderen Spielern in einem Local Area Network (LAN) zu spielen. Das Protokoll muss zuerst auf deinem Computer installiert werden. (Wenn es nicht funktioniert, konsultiere die Windows Dokumentation. Oder gehe in die Netzwerkeinstellung und füge IPX hinzu.) TCP/IP ist das Internet Protokoll. Es kann genutzt werden, um mit anderen Spielern irgendwo im Internet zu spielen, vorausgesetzt du kennst deren IP Adresse. In einem lokalen Netzwerk geht dies ohne den Adressaustausch. Eine Modem-Verbindung geschieht durch ein Modem. Du musst einige Parameter angeben (einen Initialisierungsstring und eine Telefonnummer) um sie zu verwenden. Schließlich, wenn du ein seriell Kabel verwendest (eine direkte Verbindung zwischen den Computern) brauchst du einige Anschlusseinstellungen. Es gibt vier GML Funktionen zum Initialisieren dieser Verbindungen:

- `mplay_init_ipx()` initialisiert eine IPX Verbindung.
- `mplay_init_tcpip(addr)` initialisiert eine TCP/IP Verbindung. `addr` ist ein String, der die Web-/IP-Adresse angibt, z.B. 'www.gameplay.com' oder '123.123.123.12', optional von einer Portnummer gefolgt (e.g. ':12'). Nur wenn einer Session beigetreten wird (siehe unten) musst du eine Adresse angeben. Die Person, die die Session erstellt, muss keine Adresse angeben (weil die Adresse ihres Computers die wichtige ist.) In einem lokalen Netzwerk muss keine Adresse angegeben werden, aber die Funktion muss trotzdem aufgerufen werden.
- `mplay_init_modem(initstr,phonenr)` initialisiert eine Modemverbindung. `initstr` ist der Initialisierungsstring für das Modem (kann leer sein). `phonenr` ist ein String, der die Telefonnummer angibt (z.B. '0201234567'). Nur wenn an einer Session teilgenommen werden soll (siehe unten) muss eine Telefonnummer angegeben werden.
- `mplay_init_serial(portno,baudrate,stopbits,parity,flow)` initialisiert eine serielle Verbindung. `portno` ist die Portnummer (1-4). `baudrate` ist die Baudrate die Verwendet werden soll (100-256K). `stopbits` gibt die Zahl der Stopbits an (0 = 1 Bit, 1 = 1.5 Bit, 2 = 2 Bits). `parity` gibt die Parität an (0=keine, 1=ungerade, 2=gerade, 3=markiert). Und `flow` gibt den Typ der Flusskontrolle an (0=keine, 1=xon/xoff, 2=rts, 3=dtr, 4=rts und dtr). Gibt zurück, ob es erfolgreich war. Ein typischer Aufruf ist `mplay_init_serial(1,57600,0,0,4)`. Gebe 0 als erstes Argument an, um dem Benutzer zu ermöglichen, die Einstellungen zu ändern.

Dein Spiel sollte eine dieser Funktionen genau einmal aufrufen. Alle Funktionen geben zurück, ob sie erfolgreich waren. Sie sind nicht erfolgreich, wenn das gewünschte Protokoll nicht installiert ist, oder nicht von deinem Computer unterstützt wird.

Nun sollte der erste Raum deines Spieles dem Spieler 4 Möglichkeiten geben. (Oder nur für die Protokolle, die du nutzen möchtest. Die letzten beiden könnten zu langsam für dein Spiel sein.) Wir rufen die initialisierende Funktion im Mouse Event auf und gehen, falls erfolgreich, in den nächsten Raum. Ansonsten geben wir eine Fehlermeldung aus. Im Mouse Eevent des IPX Knopfes platzieren wir also folgenden 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 global game settings 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,playername)` 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,playername)` makes you join session number `numb` (0 is the first session). `playername` 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. Then 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 1000000 values that are common to all entities of the game (so e.g. both to the master and to the slave). 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 1000000).
- `mplay_data_read(ind)` returns the value in location `ind` (`ind` between 0 and 1000000).

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. Also preferably only use the first few locations to save memory.

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 slave 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 slave), and lets a new ball start. You can check out the enclosed game `pong1.gm6` 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 sent to the new player (this would take too much time). So only new changes after that moment are sent 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 nonguaranteed 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 player's 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 could cause serious problems.)

For the playing of sounds we do the following approach. 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:

```
{
var from, name, messid, val;
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 hick-up 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:

```

{
var messid, val;
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.gm6`.

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.gm6`. 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 for 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.

Hopefully, by now you 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.

We use 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 sessions 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.gm6` 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