Lecture 8 - Introduction to multiplayer

Written by Carl Gustafsson

Goal of the lecture

After this lecture, the reader should know how to create a connection between two computers in Game Maker. The reader should also have a basic understanding in how to construct multiplayer games, i.e. games that are played across more than one computer.

This lecture was revised for round 2 of the Game Maker programming course at www.gameuniv.net. Changes to the original document are shown with slightly greenish background. If you read this document for the first time, just ignore those markings and read it as if nothing was marked.

This lecture was also revised for round 3 of the Game Maker programming course at www.gameuniv.net. Changes to the previous revision of the document are shown with slightly blueish background. People reading this document for the first time could ignore the different background colors. Most screenshots are revised, but the change is very little, so they are not marked.

Introduction

I got the feeling that there were some of you who were interested in making multiplayer games. Therefore I will here try to introduce you all to how it can be done. Please understand that making a multiplayer game is not a trivial task. Even so, I will not go further than connecting two (2) computers together in a game. Using even more connections will be even harder and I must confess that I have not tried that myself yet.

I have however managed to get a two-player game to work across the Internet (see "DogFight" on my web page, hem.passagen.se/birchdale/carl, even though I made it with GM 4.1, so there will probably be problems running it with the new version of GM), so, I intend to try and teach you how to do that.

Making multiplayer games it a **lot** easier if you have access to two different computers connected via network cards (in a small LAN). It **is** possible to run multiple instances of the game on the same computer if you first make an **exe** file out of it, but it is very hard to playtest it, since you can only have one instance active at the same time.

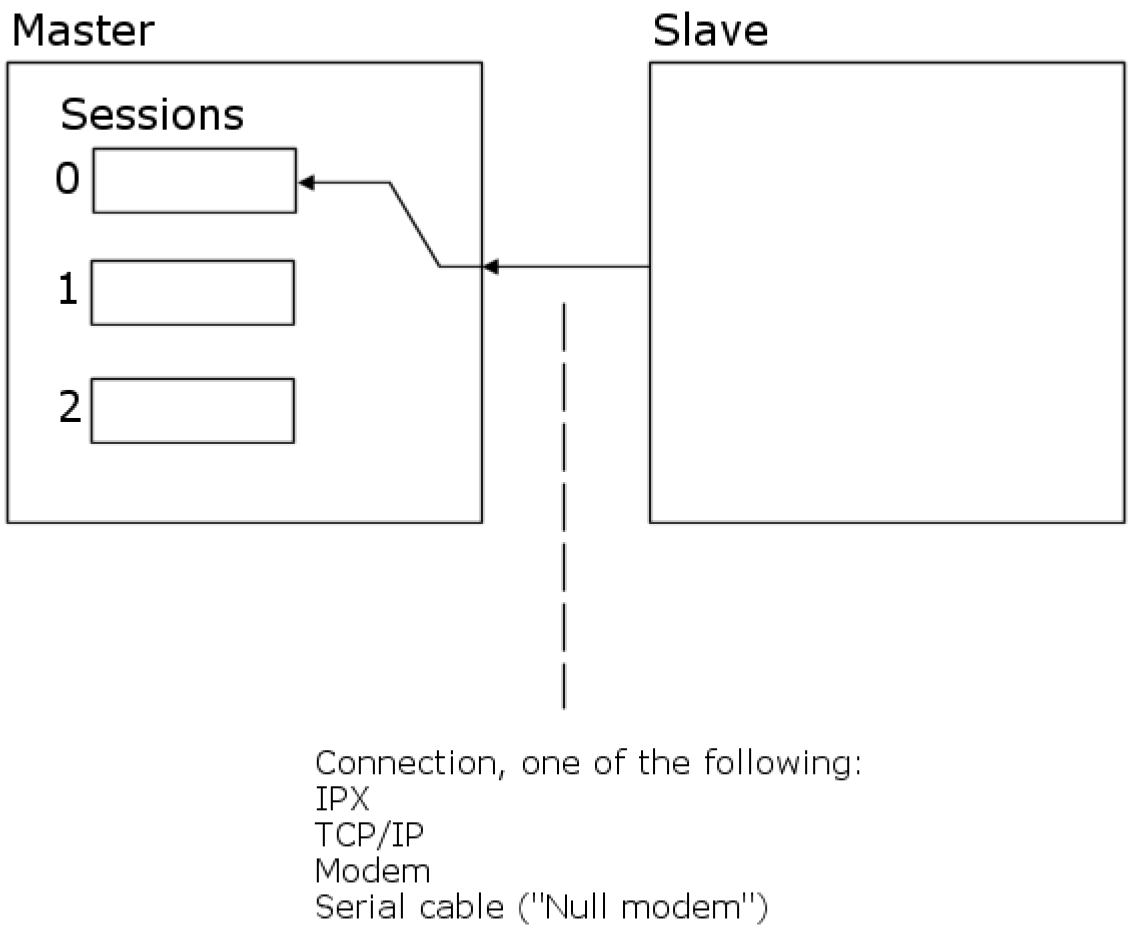
Theoretically speaking

We will try to look a bit at the theory behind it all first, before we get to actually doing it. (Just like learning to fly. :))

The connection

The first thing that is done in a multiplayer is to decide whether or not the computer will be a master or a slave in the multiplayer games. All multiplayer games have exactly one master (could be a "dedicated server") and any number of slaves (sometimes called "clients").

The master creates a new connection and a new session and then waits for the slave to connect to it. The slave also creates a new connection, and thereafter joins a session that it finds on the connection. Below is a little image that might help in explaining how it works:


Master
Slave
Sessions
0
1
2
Connection, one of the following:
IPX
TCP/IP
Modem
Serial cable ("Null modem")

The first thing that is made is thus the **connection**. I must say that I am not quite at home in the nitty-gritty details of the network protocols, but a **connection** is a kind of initialization of the protocols.

When making the connection, a decision has to be made to determine which protocol to use. Game Maker provides access to four different kinds of protocols:

- IPX
- TCP/IP
- Modem
- Serial cable ("Null modem")

Each of these protocols are quickly explained in the Game Maker manual, so please read section The Game Maker Language / Multiplayer Games / Setting up a connection of the manual before continuing.

So, now you know that the IPX protocol and the TCP/IP protocols are useful for Local Area Networks (LANs), the TCP/IP protocol is useful also for Internet connections, the modem protocol is useful for modem connections (duh) and the Serial protocol is useful for connections between two computers using a twisted serial cable, a "null modem".

(I know it means tautology to write "TCP/IP protocol", since the word "protocol" is included in both acronyms, but I find it better to write it down than to leave it out. Here in Sweden it is even common to say "CD-disc", which would then be "Compact-Disc-disc". Bah! Note that this side note is mostly aimed at those people that are kind enough to provide me with lists of linguistic errors from my lectures. (i.e. Mr Scarborough and Ms Pahl))

There are a few differences in how the master and the slave create their connections when it comes to the arguments that are passed to the initialization function. The slave needs to provide either an IP address or a telephone number to the master (depending on the protocol used), which the master does not need to do.

The session

Once the connection has been made, the master can create a new **session**. A session is essentially a game round. It can also be called an **instance** of the game. When creating the session, the master needs to provide a name for the session, the number of players that can join it to play, and the name of the player that plays on the master computer.

When the master has created a session, the slave can **join** it. This usually means searching for possible sessions and displaying them to the player. The player then decides on which session he wants to join. There can thus be multiple sessions going on at once, especially if the connection is not made to a certain computer (via Internet, Modem or serial) but via a LAN of some kind. Especially when using the IPX protocol it is important to let the player decide which session to join, since there can be lots of sessions going on in one LAN.

Once the master detects that there are enough players that have joined the session (possibly decided by the master player watching the number of joined "slave players", the game can begin, and this is where the **really** hard stuff starts - synchronizing **everything** between all the players so that everyone sees the same game world. This is especially hard in real-time games (like action games), but not that tough in turn-based games like chess and some other strategic games.

The synchronization

So, now the game is running and all the players are connected to the master and have joined the session. Time to make the data flow.

There are two ways of synchronizing the events between the players. One way is of using "Shared data". Shared data is kind of "super global variables". That means that those variables are global to the entire session and as soon as one player changes a variable, the change is reflected to all the other players. A shared data variable works about the same as other variables in Game Maker, except you can not call them what you want, and there is a limited amount of them (10.001 if I am correct). I find it hard to imagine that anyone would need more than that, but since Bill Gates is said to have assumed that 640 kB is enough RAM for anyone, I better keep my mouth shut on this. :)

The other way of sharing data is by the use of "messages". The player can decide to send a message to one or all of the other players. A message contains the ID of the player that should receive it (which can be all players), a message ID, and a value (like a variable).

I usually think like this: Use shared data for things that always need updating, like player positions, and use messages for things that does not happen as often, such as "Player 1 fires a bullet".

Trying it out

Making a connection

We will now try to make a connection. I will write examples of how it is made with the IPX and the TCP/IP protocol. If you need the other connection types, I think you can extrapolate from there.

Create a new game and add a new object. Call it **objConnection**. Then add a new script and call it **GeneralConnect**. In this script we will make the multiplayer connection. We will at first go through a lot of code snippets and I will try to explain them. Then we will put it all together in one large script at the end.

First, we will ask the player which protocol he wishes to use. We will only provide the IPX and the TCP/IP protocols here. In order to ask the player that, we use a pop-up menu. Add this to the empty script we just created:

```
// Ask player for protocol.
protocolChoice = show_menu("IPX|TCP/IP|Quit", 0);
```

We have not used the **show_menu** function earlier. It will display a pop-up menu, similar to the one that is displayed if you right-click the mouse in any Windows window. The first parameter to the function is a string that contains all the items of the menu. Here we have three items: IPX, TCP/IP and Quit. They are separated with a vertical line, also known as a "pipe". The second parameter tells Game Maker the default menu choice (0 means the first choice). When the player selects a menu item, the result is returned to Game Maker as a number. This number is above stored in the variable **protocolChoice**. If the number is 0, the user chose the first item. If it is 1, he chose the second item, and so on. So, after the call to **show_menu** we will have to check what the result of the menu choice was.

It is done like this (add it after the previous lines in the script):

```
switch (protocolChoice)
{
    case 0:
        // IPX Connection
        break;
    case 1:
        // TCP/IP Connection
        break;
    case 2:
        // Quit the script
        exit;
        break;
}
```

Ah. A hard introduction to the **switch/case** statement. This statement works similar to a range of **if** statements. It checks the expression inside the parentheses (protocolChoice) and then it runs the corresponding **case** statement. So, if **protocolChoice** is **0**, the code inside the **case 0:** statement will run, all the way up to the **break** statement, which breaks out of it all. If **protocolChoice** is **1**, the code inside the **case 1:** statement will run until a **break** is found. This goes on until the ending **}** in the **switch** statement. There is even a statement that corresponds to the **else** statement in **if** statements and will be executed if none of the above **case**s are true. It is called **default:**. It is wise to add a **default:** statement to the end of the **switch**, like this:

```
switch (protocolChoice)
{
    case 0:
        // IPX Connection
        break;
    case 1:
        // TCP/IP Connection
        break;
    case 2:
        // Quit the script
        exit;
        break;
    default:
        // None of the above happened...
        show_message("Now what???");
        game_end();
        break;
}
```

We now have a little menu skeleton. But we still need to add the actual code that makes the connections.

But before that, we will make the player choose whether he wants to **create** a game or **join** a game. I use these terms (create/join game) because I think those are the most common that are used in today's multiplayer games. If the player chooses to **create** a game, he becomes the **master**. If he chooses to **join** a game, he becomes the **slave**.

We need to know whether to **create** or **join** before making the connection, because there are different connection function calls depending on this situation. This is checked in a way that is similar to the protocol check:

```
// Ask the player for create/join
sessionChoice = ("Create game|Join game|Quit", 0);
```

Now it is beginning to be a lot of numbers to keep in mind. If the player chose "Create game" in this second menu, the variable **sessionChoice** will be **0**. Hmm. Could there be an easier way of keeping track of that? I use to make some constants that I can use instead of the literal numbers. You could add this to the start of the script:

```
CH_IPX = 0;
CH_TCPIP = 1;
CH_CREATE = 0;
CH_JOIN = 1;
```

If the player chooses to make the connection via the **IPX** protocol, we will call on the function **mplay_init_ipx()**. That function does not have any parameters at all and does not differ depending on the creation or joining of a session. On the other hand, if the player chooses the **TCP/IP** protocol, we will call the function **mplay_init_tcpip("<IP_ADDRESS>")**. Now, if the player wants to **create** a game, we just call that function with an empty string. But, if the player wants to **join** a game, we need to provide an "IP address". An "IP address" is a series of four numbers separated by a period, like "192.168.10.10" (which is the IP address of my computer on my little home LAN). You should use the IP address of the computer on which the game session is **created**. In order to check your computer's IP address, there are at least two ways. Either do it in Game Maker with the function **mplay_ipaddress()**, which returns a string containing the computer's current IP address, or run the command "ipconfig" in a DOS prompt. We will use the **mplay_ipaddress()** below in order to check the IP address.

All the connection functions return a value, either **true** or **false**. We will check this value to see if the connection was successfully created or not. In order to make any of the connections above you need to have the **TCP/IP** protocol installed on your computer (check the Network Settings in the Control Panel to see if you have that protocol installed. Otherwise, try to install it). If you want to use the **IPX** connection, you also need to have the **IPX** protocol installed on your computer. Both these protocols are provided on the standard Windows installation.

Oh, and one more thing. In the beginning of the script, we should check if there is already a connection running. If so, we should terminate it with the **mplay_end()** function. The connection is checked with the **mplay_connect_status()** function. All of the functions used here are described in the manual in the section **The Game Maker Language / Multiplayer games**. I suggest you read through the explanation of the functions we have discussed so far.

We have now seen a lot of code snippets. Let us put it all together and add some things. Among other things, we will have "nested" switch statements and all. I hope you will understand the following code with the comments I have put there. This is the entire contents of the **GeneralConnect** script.

```
// Some useful constants.
CH_IPX = 0;
CH_TCPIP = 1;
CH_CREATE = 0;
CH_JOIN = 1;

// Check if a connection already exists. If so, end it.
if (mplay_connect_status() != 0) {
    mplay_end();
}

// Ask player for protocol.
protocolChoice = show_menu("IPX|TCP/IP|Quit", 0);
// Quit if the player chose "Quit".
if (protocolChoice = 2) then exit;

// Ask player for create/join.
sessionChoice = show_menu("Create game|Join game|Quit", 0);
// Quit if the player chose "Quit".
if (sessionChoice = 2) then exit;

// Time for the nested switch statements.

// First, check the procotol chosen.
switch (protocolChoice) {
    case CH_IPX: // IPX
        // Now, check create/join.
        switch (sessionChoice) {
            case CH_CREATE:
                // Make the IPX connection
                ipxResult = mplay_init_ipx();
                // Check if the IPX connection was OK.
                if (ipxResult = true) {
                    show_message("IPX Connection was successful.");
                }
                else {
                    show_message("IPX Connection failed!");
                    game_end();
                }

                // Now it is time to create the session. We will do that soon...

                break;

            case CH_JOIN:
                // Make the IPX connection
                ipxResult = mplay_init_ipx();
                // Check if the IPX connection was OK.
                if (ipxResult = true) {
                    show_message("IPX Connection was successful.");
                }
                else {
                    show_message("IPX Connection failed!");
                    game_end();
                }

                // Now it is time to join a session. We will do that soon.

                break;
            default:
                // This should not happen.
                show_message("Oops...");
                break;

        }

        break; // The end of the IPX case.

    case CH_TCPIP: // TCP/IP
        // Now, check create/join.
        switch (sessionChoice) {
            case CH_CREATE:
                // Display the IP address of this computer:
                show_message("Your IP address is: " + mplay_ipaddress());

                // Make the TCP/IP connection, no IP address...
                tcpipResult = mplay_init_tcpip("");
                // Check if the TCP/IP connection was OK.
                if (tcpipResult = true) {
                    show_message("TCP/IP Connection was successful.");
                }
                else {
                    show_message("TCP/IP Connection failed!");
                    game_end();
                }

                // Now it is time to create a session. We will do that later.

                break;

            case CH_JOIN:
                // When joining a session on a TCP/IP connection,
                // an IP address is required. Ask the player for IP address.
                tcpipAddress = get_string("Enter IP address of master computer: ", "");

                // Make the TCP/IP connection.
                tcpipResult = mplay_init_tcpip(tcpipAddress);
                // Check if the TCP/IP connection was OK.
                if (tcpipResult = true) {
                    show_message("TCP/IP Connection was successful.");
                }
                else {
                    show_message("TCP/IP Connection failed!");
                    game_end();
                }

                // Now it is time to join a session. We will do that later.

                break;

            default:
                // This should not happen.
                show_message("Ooops * Ooops !!");

                break;
        }

        break; // End of the TCP/IP case.

    default:
    // This should not happen.
    show_message("Ooops * Ooops * Ooops !!!");

    break;

}
```

Now, we need someplace where the script can run. Put it in the **Mouse->Left Pressed** event of the **objConnection** object. We also need a little sprite for the object. You can use this button image for that. I just made it in POV-Ray (www.povray.org), another free 3D program (albeit one with a pretty steep learning curve). The button image is called **Button1.png**. Load it as a sprite and call it **sprButton**. Make it **Non-transparent**. Use this sprite for the **objConnection** object. Now we have a button to press.

Add a room and put an instance of the **objConnection** in the room. You can now test run the game (SAVE FIRST!!). Click the button and select a connection and either **create** or **join**. I hope that you will then receive the message "Connection was successful.". When asked for an IP address, just enter **0.0.0.0** for now. That should even work if the other computer is on the same LAN as your own computer.

If the connection fails, check in the Control Panel that the network protocols are installed correctly.

Making a session

Now that we have a connection we need to create a session (or, join a session if we are the "slave").

Creating a session is no big deal, but joining requires a bit more work. In order to create a session we use the **mplay_session_create()** function, and in order to join a session, we use the **mplay_session_join()** function.

First, since we need some text to be drawn in the next script, please add a font resource and call it **fntArial**. I call it that because I used the Arial font. You can use any font you like and any font name you like, just remember to use that name instead of **fntArial** below.

Create a new script and call it **CreateSession**. In this script we will first create a session and then wait for another player to join it. No meaning in starting the game until the other player has joined, right? :) So, make the script look like this:

```
// Create a session.

// First, end any ongoing session.
if (mplay_session_status() > 0) then {
    mplay_session_end();
}


// Then, create a new session.
sessionResult = mplay_session_create("My Session", 0, "Master Player");


// Check if the session was created OK.
if (sessionResult = true) then {
    show_message("A session was successfully created.");
}
else {
    show_message("Failed to create a session.");
    // Return false to the calling script.
    return false;
}

// Wait for other player to join. This is done with a little loop that
// displays a message on the screen.
// Look for other player
nPlayers = mplay_player_find();
// As long as there is only one player (this) the loop will wait.
// It should also be possible to quit the waiting with the "q" key.
while (nPlayers <= 1 and not keyboard_check(ord("Q"))) {
    // Draw some text on the screen and refresh it.
    draw_set_color(c_black);
    draw_set_font(fntArial);
    draw_set_halign(fa_center);
    draw_text(room_width / 2, room_height / 2, "Waiting for other players");
    draw_text(room_width / 2, room_height / 2 + 20, "Press Q to quit");

    // Refresh the screen. This is needed since this is not run
    // in the Draw event.
    screen_refresh();
    // Finally, check for other player again.
    nPlayers = mplay_player_find();

    // Update the keyboard status.
    io_handle();
}

// Check if there are any more players or if the player pressed "Q".
if (mplay_player_find() > 1) then {
    show_message ("Player '" + mplay_player_name(1) + "' has successfully joined this
session.");
    return true;
}
else {
    return false;
}
```

First any ongoing session is ended, then the new session is created. The parameters to the **mplay_session_create()** function are **session name** ("My Session"), **number of players** (0 = arbitrary) and **player name** ("Master Player"). A check is made to see that the session was created OK.

Once the session is created, the script waits for an other player to join. This is done in a loop that displays a message. This loop keeps going until there are more than one player in the game, or until the player presses "Q" on the keyboard to abort the loop.

I do not remember if we have used the **while** statement before, but here is a short explanation of it. You could say that it resembles the **if** statement. If the expression inside the paremtheses is resolved to **true**, all the statements inside the curly braces **{}** are executed (just like in an **if** statement). The difference here is that once all statements have been executed, the expression inside the parentheses is evaluated again. If it is still true, all the statements are executed again. This goes on all the time until the expression inside the parentheses becomes **false**. In our case the expression is true as long as there is only one player in the session **AND** the player has not pressed "Q" on the keyboard.

Oh, and another thing that I think is new for you is the **return** statement. This statement simply ends the script and returns the value that follows it. So, if everything works out as it should in the script above it should end at the **return true** line, second from the end.

This script can then be called like this:

```
if (CreateSession()) then {
    // It worked!!!
}
else {
    // Sob.... it did not work...
}
```

Let us make the **JoinSession** script too now. Write it like this:

```
// Join a session


// First, end any session that might be running.
if (mplay_session_status() > 0) mplay_session_end();


// Search for sessions.
sessionsFound = mplay_session_find();


// If no sessions were found, exit with "false" value.
if (sessionsFound = 0) then return false;


// Otherwise, present the user with a list of sessions to join.
// This is done with a "menustring" like "session1|session2|session3|..."
strSessionsMenu = "";
for (i = 0; i < sessionsFound; i += 1) {
    strSessionsMenu += "Join session '" + mplay_session_name(i) + "'|";
}
// Add the final "Quit" to the string.
strSessionsMenu += "Quit";

// Show the menu to the player.
joinChoice = show_menu(strSessionsMenu, 0);

// Exit if the player chose "Quit".
if (joinChoice = sessionsFound) then return false;

// Try to join the session and see if it worked.
joinResult = mplay_session_join(joinChoice, "Slave Player");

// Check if it worked to join the session.
if (joinResult = true) then {
    show_message("Joining session '" + mplay_session_name(joinChoice) + "' was
successful.");
    return true;
}
else {
    show_message("Failed to join session '" + mplay_session_name(joinChoice) + "'");
    return false;
}
```

Let's dissect that script a bit. First any existing session is ended. Then a search is made to look for available sessions. This search must be made, because there can be lots of ongoing sessions in a LAN. If there are no sessions found, the script exits returning the false value. But if there are available sessions, the script compiles a string with session names, separated with the **pipe** character (|). To this string we add "Quit" and then present it to the user with the **show_menu()** function.

If the user chose to join a session (and not to Quit), an attempt is made to join the selected session. If all goes well, the script returns **true**. Otherwise it returns **false**.

We now need to run these new scripts from somewhere. In the script **GeneralConnect** there are a few comments that say "Now it is time to create a session...." or "Now it is time to join a session....". Just below those comments, add the line:

```
sessionResult = CreateSession();
```

and

```
sessionResult = JoinSession();
```

respectively (I think you can figure out which line goes where. :) :)). Hint: There should be two of each one.

This is how we call our own scripts from code. Just the script name, followed by two parentheses. The parentheses are there because it is possible to use parameters when calling a script, and they will then go in between those (just like other function calls).

In order to test this you now need to run two instances of the game. The easiest way is probably to load up another instance of Game Maker and load this same game into that instance. Then run the game in both instances. Move the two windows apart a bit so that you can tell them apart. I also made my rooms much smaller than the default (300 x 300 pixels). In one instance of the game you choose one of the protocols and then **Create game**. It should then display some info about successfully creating connections and sessions and then a message that says that it is waiting for an other player to connect. This is when you go to the second game instance. In that, you choose the same protocol and then to **Join game**.

If everything goes well you should now be able to click the **first** game instance again to give it the chance to detect that someone wants to join. Then confirm the messages that appear and click a bit on both game windows to give them the possibility to react. When everything settles you should have a connection and a session running!!

I had some problems getting a connection because of my IP network settings (in Windows) was set on "Server assigned IP address". I did not have a network cable connected at the time of my first tries, and therefore did not have a "real" IP address. Then the Game Maker connections did not work either (I could not connect to any session). So, I connected to a network and was given an IP address, and then it worked.

I am afraid I do not know very much about the details in the TCP/IP stack, and therefore I can not really explain this. I also had problems connecting with IPX.

The next level

We are now ready to go to the next level and make some objects that can move and be controlled via the connection.

Some good moves

In order to make something happen we need some player objects. It is also a good idea to move to another room, a "play room" once the network connections are up and running.

So, add a new room. Then add some objects: **objPlayerParent**, **objPlayer1Master**, **objPlayer1Slave**, **objPlayer2Master**, **objPlayer2Slave**. Wow that was a lot of objects. Well, the thought is that each player have two versions of each object. The **Master** objects are used when the game is **created**, and the **Slave** objects are used when the game is **joined**. There is also one **parent** object that contains some of the actions that are common for all the player objects (like sprite assignment and such). That is right. Make the **objPlayerParent** the parent of all the other player objects.

We will attempt to make an eight-directional movement sprite here. I have made a little animated worm in Blender. I hope you find it cuter than the spaceships and planes we have used in previous lectures. :) (This is specifically aimed at you, Kathi ;-)) The thought is that there is one animated sprite for each of the eight directions. Add the sprites and call then e.g. **sprWormSW, sprWormS, sprWormSE,...** etc. The last letters are for South-West, South, South-East, etc, for the direction in which the worm is moving. Then assign one of the sprites to the **objPlayerParent**.

In the **Create** event of the **objPlayerParent**, do this:

```
// Set some constants
normalSpeed = 2;

// Set the initial facing_direction (made-up variable)
facing_direction = 0;
```

In the **STEP** event of the **objPlayerParent**, put this code in order to assign the correct sprite:

```
// Assign the correct sprite to the instance.
switch (facing_direction) {
    case 0: sprite_index = sprWormE; break;
    case 45: sprite_index = sprWormNE; break;
    case 90: sprite_index = sprWormN; break;
    case 135: sprite_index = sprWormNW; break;
    case 180: sprite_index = sprWormW; break;
    case 225: sprite_index = sprWormSW; break;
    case 270: sprite_index = sprWormS; break;
    case 315: sprite_index = sprWormSE; break;
    default: sprite_index = sprWormE; break;
}
if (speed = 0) then {
    // Stop the sprite animation.
    image_speed = 0;
}
else {
    // Start the sprite animation again.
    image_speed = 1;
    // Only copy the direction to facing_direction if
    // we are moving.
    facing_direction = direction;
}
```

Then open up the **objPlayer1Master**. Here we will add the following events and code (Be careful to choose "Key Press" and not just "Keyboard" events):

Event	**Action code**
Key Press <Left>	`hspeed = -normalSpeed;`
Key Press <Right>	`hspeed = normalSpeed;`
Key Press <Up>	`vspeed = -normalSpeed;`
Key Press <Down>	`vspeed = normalSpeed;`

Now the worm ought to move as it should. It will however not stop when we release the keys. So, put the following code in the **Step** event of the **objPlayer1Master**.

```
kLeft = keyboard_check(vk_left);
kRight = keyboard_check(vk_right);
kUp = keyboard_check(vk_up);
kDown = keyboard_check(vk_down);
if (not (kLeft or kRight)) then hspeed = 0;
if (not (kUp or kDown)) then vspeed = 0;
```

The first four lines will just assign the state of the four cursor keys to some shorter variables, just to make it easier to write the rest of the code. Then a check is made to see if neither the left nor the right keyboard keys are pressed. If they are not, the horizontal speed is set to 0. Similar actions are taken for the vertical keys and speed.

The problem with this is that now that we have defined a **Step** event for this object, it will **override** the **Step** event of the parent object. Hmmm. There is a very simple fix for that though. To the beginning of the **Step** event, add the **Call the inherited event** action (from the **control** tab). This will first execute the actions of the parent's event, before looking at the actions specified here. Good.

If you want to test out the worm, add an instance of the **objPlayer1Master** to the first room and try it out. I hope it wriggles somewhat satisfactory.

Beep...beep...data transfer...**B010000012f4ced...

(Sorry..got a little carried away again.. I tend to write these lectures after 22:00 every time and the more I grow tired, the more bad jokes and stuff I put here.
Oh...just happened to find my "OneLiners.txt" file. Contains lots and lots of wise words. How about these:
1 + 2 = 3 Therefore 4 + 5 = 6
2+2=5, for sufficiently large values of 2
A friend likes you even though he doesn't need you anymore
A gun gives you the body, not the bird
Apathy Error: Don't bother striking any key
Ben Kenobi at the dinner table: Use the FORKS, Luke!
DEFINITION: Disassembler - An unattended five year old child
SET PATH = LOOK AROUND ON HD; DESK DRAWER; UNDER DESK; HALL CLOSET;
Ah, well, I guess I'd better stop there...)

Alright! Let's get back to work. We can now move our worm, and we now need to transfer the movement to the other player's computer in some way. The best way to do that (in my opinion) is do make use of the "shared data variables". As said earlier there are 10001 of them. So, the first thing we have to do is to decide which ones to use.

We need to transfer the coordinates (2 values), the direction and the speed of the player object to the other computer. Say, how about using number 0, 1, 2 and 3? :)

Add this code to the **End Step** event of the **objPlayer1Master** object:

```
// Write the coordinates to the shared data variables
mplay_data_write(0, x);
mplay_data_write(1, y);
// Write the direction to the shared data variables
mplay_data_write(2, direction);
// Write the speed to the shared data varaibles
mplay_data_write(3, speed);
```

That was not so hard, was it? :) Better keep a little table of what the shared variables contain.

Index	Data
0	Player 1 x coordinate
1	Player 1 y coordinate
2	Player 1 direction
3	Player 1 speed

Now, the player one object on the slave computer (**objPlayer1Slave**) should read these values and use then for moving. Add a **Step** event to the **objPlayer1Slave** object. To that event, first add the **Call the inherited event** action. Then add this code:

```
// Read movement values from shared data.
x = mplay_data_read(0);
y = mplay_data_read(1);
direction = mplay_data_read(2);
speed = mplay_data_read(3);
```

There! Now the slave version should act just like the master version.

Putting it together

Create two new rooms and call then **MasterRoom** and **SlaveRoom**. I think this is the best approach since the master and the slave should have different objects existing in their respective room. In the **MasterRoom**, add an instance of the object **objPlayer1Master** and one instance of **objPlayer2Master**. In the **SlaveRoom**, add an instance of the object **objPlayer1Slave** and one instance of **objPlayer2Slave**.

We now need to make sure that the player is moved to the correct room after the connection and session hasve been created. That can be done if you add these lines to the **end** of the **GeneralConnect** script:

```
if (sessionResult = false) then exit;

if (mplay_session_status() = 1) then {
    room_goto(MasterRoom);
}
else if (mplay_session_status() = 2) then {
    room_goto(SlaveRoom);
}
```

This piece of code will just exit of the sessions were not completely created/joined. Otherwise (if all is OK) it will go to either the **MasterRoom** or the **SlaveRoom** depending on the result from the **mplay_session_status()** function. That function returns **0** if no session is active, **1** if a session has been successfully created and **2** if a session has been successfully joined.

If you now try out the game, player 1 movement should work. If not, something has been left out...

We should now try to make player 2 work too. Actually, there is so much that is the same as player 1, so, change the parent of **objPlayer2Slave** to **objPlayer1Master**. We will just change a little bit of it. The only thing that needs changing is the **End Step** event where the shared data is written. Add the **End Step** event to the **objPlayer2Slave** and enter the following code:

```
// Write the coordinates to the shared data
variables
mplay_data_write(4, x);
mplay_data_write(5, y);
// Write the direction to the shared data variables
mplay_data_write(6, direction);
// Write the speed to the shared data variables
mplay_data_write(7, speed);
```

And we update our shared data table, just for our own reference:

Index	Data
0	Player 1 x coordinate
1	Player 1 y coordinate
2	Player 1 direction
3	Player 1 speed
4	Player 2 x coordinate
5	Player 2 y coordinate
6	Player 2 direction
7	Player 2 speed

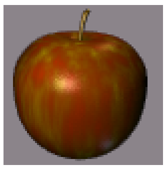
Finally, make the **Master** version of player 2 act like the **Slave** version. This is done by adding the event **Step** to the **objPlayer2Master** and first add the action **Call the inherited event** to it, and then this code:

```
x = mplay_data_read(4);
y = mplay_data_read(5);
direction = mplay_data_read(6);
speed = mplay_data_read(7);
```

The goal of the game

Fantastic! Now we can control two worms and make them appear on two different computers. Try it if you do not believe it. But there is still no real meaning to the game, and we still have not made use of the message sending function.

An apple a day keeps the doctor away



Let us make the worms go after an apple. This is common in worm games, right? Well, here is an apple anyway (also made in Blender). We will make it so that an apple appears in a random place after a short, random time. The first worm that makes it to the apple gets some points, then the apple appears again, in a new place, and so on. Create an apple sprite, **sprApple** using this image, and an apple object, **objApple**. Assign the sprite to the new object. We will also use the "boitt" sound, **boitt.wav**, when the apple is destroyed (taken by a worm). Add this sound too and make it play once in the **Destroy** event of the apple object. Great!

What we now need is some kind of controller that handles the distribution of apples (apple-monger? :)). Create the object **objAppleMaster** and the object **objAppleSlave**. Make them invisible and add the **objAppleMaster** to the **MasterRoom** and the **objAppleSlave** to the **SlaveRoom**. We will make it so that the **objAppleMaster** sends a message to the **objAppleSlave** when an apple should be created. Once a worm hits an apple we will make that worm send a message about this. The apple control objects will be used as message listeners.

In the **Create** event of the **objAppleMaster** object, create an instance of the **objApple** object at a random location. This could be done like this:

```
myApple = instance_create(round(random(room_width - 80)), round(random(room_height -
80)), objApple);
```

An apple instance will be created at a random location inside the room. The subtraction of 80 is there because of the width and height of the apple sprite. The we need to tell the slave where we have created the apple so that they both create the apples at the same location. That can be done by adding this after the line above:

```
coordinateString = string(myApple.x) + "," + string(myApple.y);
mplay_message_send_guaranteed(0, 0, coordinateString);
```

First a string is created. This string consists of the x and y coordinate of the apple that was just created. I have chosen to do it like this because it is only possible to send a single value in a message like this. That also means that we have to convert it back into two values after receiving it at the other end. We will look at that in a minute. Before that, take a look at the parameters of the message sending function. The first parameter is the receiver of the message. If we set 0 there (as above) the message is sent to all players. The second parameter is a message ID. It is used to determine what kind of message this is, and it is up to the designer to decide what each message ID means, just like with the shared data indexes.

To be good designers, let's put up a small table over the message IDs and their contents:

Message ID	**Description**	**Contents**
0	Create a new apple at the slave side	"<Apple x coordinate>,<Apple y coordinate>"

You might also have noted the word "guaranteed" in the function call. That is because there are two ways of sending messages: fast and insecure or slower and secure. The secure/insecure is just a matter of reaching the destination or not. When working with networks, there **is** a small chance of data getting lost along the way. Therefore there is some sort of built-in mechanism that makes sure that a message reaches the destination if it is sent "guaranteed". I do not know exactly how it works (I guess it relies some kind of acknowledge signals) but we need not know that in order to use it. Just remember that there are two "modes" when sending messages. Sending guaranteed messages takes a little longer because of the safety mechanisms.

Open up the **objAppleSlave** now and have a look at how we receive messages. Receiving messages is a matter of "polling the message queue".

*(Side note: Generally you can say that there exist two mechanisms for receiving signals: ether through polling or through interrupts. Polling is slower but easier to implement (I think) and interrupts are faster. You can think of the difference as the difference between receiving a phone call and an email (assuming you do not have an email notificaion sound on your computer). When you receive an email you will have to "Poll" your email box to see if there is an email. If you are expecting an important email you might have to poll very often in order not to delay the answer too much. When the phone rings on the other hand, you are interrupted in what you do and you can immediately take the call. (Well, at least **I** do not poll my phone to see if there is anyone there at the other end every now and then. :)). But in Game Maker it is polling that is the method in use.)*

Add the **Step** event to the **objAppleSlave** object. Then, add this code to that event:

```
// Poll the message queue to see if there are any messages.
while (mplay_message_receive(0)) {
    // Do different actions depending on the message ID.
    switch (mplay_message_id()) {
        // 0 - Create an apple
        case 0:
            // Get the position of the comma sign in the string
            commaPos = string_pos(",", mplay_message_value());
            // Get the x and y coordinates and convert to real values.
            xCoord = real(string_copy(mplay_message_value(), 1, commaPos - 1));
            yCoord = real(string_copy(mplay_message_value(), commaPos + 1,
string_length(mplay_message_value()) - commaPos));
            // Create a new apple at the given coordinates.
            myApple = instance_create(xCoord, yCoord, objApple);
            break;
    }
}
```

Phew! That was not the easiest... but now we should have a slave version of the apple object too. The string splitting might look a bit hairy if you are not used to working with strings. Unfortunately I do not have the time and energy to explain the string functions in further detail here.
If you are interested in a general string splitting script though, there is a pretty good working version among the examples on my web page, http://hem.passagen.se/birchdale/carl .
There need to be collision events defined too. In the **objPlayer1Master**, add the collision event with the apple object. When colliding with the apple, the apple should be destroyed and a message should be sent to the slave to remove the apple there too. Add this code to the collision event:

```
// Destroy the apple
with (other) instance_destroy();
// Send a destroy order to the slave
mplay_message_send_guaranteed(0, 1, 0);
// Set a timer for new apple
objAppleMaster.alarm[0] = random(90);
```

Here we first destroy the apple, and then send a message with ID **1** to the slave. Note that the **value** of the message (the third parameter) does not matter in this message, because the slave only needs to know the ID so that it can destroy the apple.
In order to destroy the apple at the slave when receiving the above message, modify the message polling code above so that it looks like this (additions written in **bold text**):

```
// Poll the message queue to see if there are any messages.
while (mplay_message_receive(0)) {
    // Do different actions depending on the message ID.
    switch (mplay_message_id()) {
        // 0 - Create an apple
        case 0:
            // Get the position of the comma sign in the string
            commaPos = string_pos(",", mplay_message_value());
            // Get the x and y coordinates and convert to real values.
            xCoord = real(string_copy(mplay_message_value(), 1, commaPos - 1));
            yCoord = real(string_copy(mplay_message_value(), commaPos + 1,
string_length(mplay_message_value()) - commaPos));
            // Create a new apple at the given coordinates.
            myApple = instance_create(xCoord, yCoord, objApple);
            break;
        // 1 - Destroy the apple
        case 1:
            if (instance_exists(myApple)) then {
                with (myApple) instance_destroy();
            }
            break;
    }
}
```

Then, do a similar thing with the slave worm. In the **objPlayer2Slave**, add the collision event with the apple object and enter:

```
// Destroy the apple
with (other) instance_destroy();
// Send a destroy order to the master
mplay_message_send_guaranteed(0, 2, 0);
```

Now there is a message sent from the slave to the master. So, we will have to poll for messages in the **objAppleMaster** too (funny... AppleMaster sounds like a TV-shop apple-peeling machine or something... :)). Add the **Step** event and write down this code:

```
while (mplay_message_receive(0)) {
    switch (mplay_message_id()) {
        // 2 - Destroy the apple
        case 2:
            if (instance_exists(myApple)) then {
                with (myApple) instance_destroy();
            }
            // Set the timer for the creation of a new apple.
            alarm[0] = random(90);
            break;
    }
}
```

Oh, and perhaps we should update the message ID table too?

Message ID	**Description**	**Contents**
0	Create a new apple at the slave side	"<Apple x coordinate>,<Apple y coordinate>"
1	Delete the slave apple	N/A
2	Delete the master apple	N/A

Now, I think, the only thing that remains is the **alarm0** event of the **objAppleMaster** object. Here we should create a new apple, just like in the **Create** event:

```
myApple = instance_create(round(random(room_width - 80)), round(random(room_height -
80)), objApple);
coordinateString = string(myApple.x) + "," + string(myApple.y);
mplay_message_send_guaranteed(0, 0, coordinateString);
```

Heeeee scores!

What are we missing now? Let me see. Some scores perhaps? First, make sure that no score is automatically written in the caption string. This can be done in the **objConnection** object (You should know how by yourself, hehe.. :)).

Then we decide on two new shared variables that can carry the scores of the players. Here is the new updated shared data table with the new scores at the end of the table:

Index	Data
0	Player 1 x coordinate
1	Player 1 y coordinate
2	Player 1 direction
3	Player 1 speed
4	Player 2 x coordinate
5	Player 2 y coordinate
6	Player 2 direction
7	Player 2 speed
8	Player 1 score
9	Player 2 score

We use the **objAppleMaster** object to reset those values in the beginning of the game. In the **Create** event, add this code:

```
// Reset the scores
mplay_data_write(8, 0);
mplay_data_write(9, 0);
```

Then we update the scores in the **objPlayer1Master** collision event with the apple. Add this there:

```
// Increase player 1 score
mplay_data_write(8, mplay_data_read(8) + 1);
```

And, in the same collision event in **objplayer2Slave**:

```
// Increase player 2 score
mplay_data_write(9, mplay_data_read(9) + 1);
```

We also have to display the score. I have chosen to display it in the caption bar. Add this to the end of the **Step** event of the **objAppleMaster** object:

```
// Update the window caption with the scores
player1Score = string(mplay_data_read(8));
player2Score = string(mplay_data_read(9));
room_caption = "Player 1: " + player1Score + "   Player 2: " + player2Score;
```

And, add the exact same code to the end of the **Step** event of the **objAppleSlave** object.

Now try the game. Preferrably on two computers playing against each other.

I really hope it works now. It took a very long time writing this and I had to do a lot of testing. I am very sorry if I missed something out. It is likely that I have done that. If you find anything to be wrong herein, please do not hesitate to tell me so.

Assignment 8 - Add "chatting" feature to the worms game

Due date: Saturday, 8 October 2005, 08:00 AM (29 days 19 hours early)
Maximum grade: 100

In this assignment you will add a little "chatting" feature to the worms game.
It should work like this:

- Pressing ENTER should bring up a string request popup (use the function get_string()).
- The player can then enter a string in the popup box and click OK.
- The string is then sent to the other player.
- When a player receives a string like this, it is displayed in a popup window (use the function show_message())

If you feel like making a more advanced interface for this, like for example having the messages being displayed as text in the game window and not in a popup window, you are of course free and encouraged to do so. 🙂 But the main thing here is to get the message over to the other player and display it in some way.

Good luck!

Regards
Carl

Read the GM Manual

Due date: Saturday, 8 October 2005, 08:00 AM (29 days 19 hours early)
Maximum grade: 0

Read the following sections from the GM 6.1 help file section "The Game Maker Language (GML)":

- FILES, REGISTRY AND EXECUTING PROGRAMS
- MULTIPLAYER GAMES

These sections cover file and registry handling and multiplayer stuff.