Tutorial: Creating Multiplayer Games

Last changed: September 1, 2004
Uses: version 6.0, advanced mode, registered version
Level: Advanced
Translation: Augenzeuge + http://dict.tu-chemnitz.de/

Spiele gegen den Computer zu spielen macht Spaß aber gegen/mit andere(n) Menschen zu spielen kann noch viel mehr Spaß bereiten. Zudem ist es relativ einfach solche Spiele zu erstellen, weil man keine komplizierte KI für den Computergegner implementieren muß. Man kann natürlich mit 2 Leuten vor demselben PC sitzen und verschiedene Tasten verwenden oder andere Eingabegeräte aber es ist viel interessanter, wenn jeder Spieler vor seinem eigenen PC sitzen kann. Oder noch besser: Wenn der andere Spieler jenseits des Ozeans sitzt. Deswegen besitzt der Game Maker Mehrspielerunterstützung. Dieses Tutorial erklärt, wie sie angewendet wird. Aber sei gewarnt! Das Erstellen effektiver Mehrspielerspiele ist nicht einfach. Es erfordert, dass du ein erfahrener Game Maker Benutzer bist. Du musst die Verwendung von einigen Programmcodes verstehen, deshalb sollte es nicht dein erstes Spiel sein, was du entwickelst. Um die Multiplayerfunktionen benutzen zu können, brauchst du eine registrierte Version des Game Makers.

Innerhalb dieses Tutorials werden wir ein simples 2-Spieler Pong und ein kleines Chatprogramm erstellen. Der Schwerpunkt liegt nicht auf tollen Grafiken oder raffiniertem game play, sondern nur bei den Multiplayer Aspekten. Du kannst es als Basis für ein kunstvolleres Spiel verwenden.
Wir werden folgende Punkte behandeln:

· Eine Verbindung zu einem anderen Computer herstellen
· Erstellen oder Verbinden mit einer Spielsitzung
· Die Spiele synchronisiert halten

Der letzte Punkt ist der schwierigste von allen Multiplayer-Spielen. Das Problem besteht darin, sicherzustellen, dass beide Spieler exakt das gleiche Spiel sehen. In dem Pong Spiel beispielsweise sollten beide Spieler den Ball an exakt derselben Position sehen. Game Maker liefert dir die Werkzeuge dafür aber du musst die Kommunikation für jedes Spiel, was du machst, selber entwickeln.

Herstellen einer Verbindung

Normalerweise funktioniert ein Multiplayer-Spiel wie folgt: Bei jedem Spieler läuft eine Kopie des Spiels. Allerdings laufen sie in einem unterschiedlichen Modus. Ein Spieler betreibt sein Spiel im Server-Modus.Die anderen Spieler betreiben ihr Spiel im Client-Modus. Der Server sollte das Spiel zuerst starten und die Spielsitzung (session) erstellen. Die anderen Spieler können dann dieser session beitreten, um am Spiel teilzunehmen. Die Spieler müssen sich für einen Kommunikationsmechanismus zwischen den Computern entscheiden. In einem local area network ist es am einfachsten mit einer IPX-Verbindung (siehe unten für weitere Details). Falls alle Spieler mit dem Internet verbunden sind, wird gewöhnlich TCP/IP verwendet. In diesem Protokoll muss der Client die IP-Adresse des Servers kennen. Demnach muß der Spieler, dessen Spiel im server-Modus läuft, seine IP-Adresse den anderen Spielern übermitteln (zum Beispiel durch senden einer email). Du kannst deine IP-Adresse ermitteln, indem du ein Programm namens winipcfg.exe aus deinem Windowsverzeichnis verwendest. Du kannst auch die Game Maker Funktion mplay_ipaddress() dafür verwenden. Eine etwas angestaubte Methode der Verbindung ist das Verwenden einer MoDem-Verbindung (in diesem Fall muss der Client die Telefonnummer des Servers kennen und bereitstellen) oder einer seriellen Leitung.

Realisiere bitte, dass die Kommunikation schwieriger wird, jetzt wo Leute firewalls und router benutzen. Diese tendieren dazu Nachrichten zu blockieren und IP-Adressen zu konvertieren. Falls du Probleme hast eine Verbindung aufzubauen könnte dies ein Grund sein. Am Besten ist, du probierst mit einem kommerziellen Spiel aus, ob eine Verbindung möglich ist. Suche bei ??? nach Informationen, wie man das Problem umgehen kann. Ferner sei dir bewußt, dass die interne IP-Adresse nicht zwingend dieselbe ist, wie die externe wegen eines Routers beispielsweise.

Damit 2 Computer miteinander kommunizieren können, benötigen sie ein Verbindungsprotokoll. Wie die meistens Spiele, bietet Game Maker vier verschiedene Verbindungstypen an: IPX, TCP/IP, MoDem und Seriell. Die IPX Verbindung (genauer gesagt ist es ein Protokoll) arbeitet fast vollständig durchschaubar. Es kann verwendet werden, um Spiele mit anderen Leuten zu spielen, die sich im selben LAN (local area network) befinden. Die Protokolle müssen auf deinem Rechner installiert sein, um sie anzuwenden. (Falls dies nicht funktioniert, konsultiere die Windows-Dokumentation oder füge das IPX-Protokoll manuell hinzu.) TCP/IP ist das Internet-Protokoll. Es kann verwendet werden, um mit anderen Spielern irgendwo im Internet zu spielen - sofern die IP-Adresse bekannt ist. Im LAN kannst du es verwenden ohne Adressen bereitzuhalten. Eine MoDem-Verbindung wird mittels eines MoDem hergestellt. Du musst einige MoDem-Einstellungen bereitstellen (Einen Initialisierungsstring und eine Telefonnummer), um es zu benutzen. Zum Abschluß noch die serielle Verbindung (eine direkte Kabelverbindung der Computer via RS-232), bei der du einige Porteinstellungen bereithalten musst. Es gibt vier -Funktionen, die verwendet werden können um diese Verbindungen zu initialisieren:

· mplay_init_ipx() initialisiert eine IPX-Verbindung.
· mplay_init_tcpip(addr) initialisiert eine TCP/IP-Verbindung. addr ist eine Zeichenkette (string), welche die web-Adresse oder die IP-Adresse beinhaltet zum Beispiel 'www.gameplay.com' oder '123.123.123.12' unter Umständen gefolgt von einer Port-Nummer (z.B.: '12'). Nur für den Beitritt zu einer Session (siehe unten) musst du eine Adresse bereithalten. Die Person, welche die Session erstellt, muss keine Adresse bereithalten (weil ja die Adresse seines Computers diejenige von Belang ist). In einem LAN sind keine Netzwerkadressen notwendig, trotzdem wird der Aufruf benötigt.
· mplay_init_modem(initstr,phonenr) initialisiert eine MoDem-Verbindung. initstr ist der Initialisierungsstring für das MoDem (kann leer sein). phonenr ist eine Zeichenkette, welche die anzurufende Telefonnummer enthält (z.B.'0201234567'). Nur für den Beitritt zu einer Session (siehe unten) musst du eine Telefonnummer bereithalten.
· mplay_init_serial(portno,baudrate,stopbits,parity,flow) initialisiert eine serielle Verbindung. portno ist die Portnummer (1-4). baudrate ist die zu verwendende Baudrate (100-256K). stopbits gibt die Anzahl der Stopbits an(0 = 1 bit, 1 = 1.5 bit, 2 = 2 bits). parity gibt Parität an (0=keine, 1=ungerade, 2=gerade, 3=mark) und flow gibt die Art der Flußkontrolle an (0=keine, 1=xon/xoff, 2=rts, 3=dtr, 4=rts und dtr). Rückmeldung, ob erfolgreich. Ein typischer Aufruf ist mplay_init_serial(1,57600,0,0,4). Gib 0 als erstes Argument an, damit sich ein Dialog öffnet in dem der Benutzer diese Einstellungen ändern kann.

Dein Spiel sollte eine dieser Funktionen exakt einmal aufrufen. Alle Funktionen geben wieder, ob sie erfolgreich waren. Sie sind nicht erfolgreich,
wenn das jeweilige Protokoll nicht installiert ist oder von deinem Rechner nicht unterstützt wird.

Der erste Raum in deinem Spiel sollte die vier Möglichkeiten anzeigen und den Spieler eine auswählen lassen. (Oder nur die Protokolle, welche du erlaubst. Die letzten beiden sind möglicherweise zu langsam für dein Spiel.) Wir rufen die Initialisierungsfunktion im mouse event auf und - sofern erfolgreich - wechseln zum nächsten room. Ansonsten generieren wir eine Fehlermeldung. Im mouse event der IPX-Schaltfläche (button) plazieren wir folgenden

Programmcode:

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

Wenn das Spiel zuende ist oder die Multiplayerfähigkeit nicht mehr vom Spiel benötigt wird solltest du folgende Routine verwenden, um die Verbindung zu beenden:

· mplay_end() beendet die aktive Verbindung. Gibt wieder, ob erfolgreich.

Du solltest diese Routine auch aufrufen, bevor du eine neue andere Verbindung erstellen möchstest.

Game sessions

Wenn du mit einem Netzwerk verbindest, können dort mehrere Spiele im selben Netzwerk stattfinden. Wir nennen diese sessions. Diese unterschiedlichen Sessions können mit verschiedenen Spielen korrespondieren oder mit demselben. Ein Spiel muß sich eindeutig/einzigartig im Netzwerk identifizieren. Glücklicherweise erledigt der Game Maker das für dich. Die einzige Sache, die du wissen musst ist das, wenn du die game id in den global game settings änderst, sich diese Identifikation ändert. Auf diese Weise kannst Du verhindern, dass Leute mit einer älteren Version deines Spiels gegen Leute mit neueren Versionen deines Spiels spielen.

Wenn du ein neues Mehrspielerspiel starten willst musst du eine neue session erstellen. Dafür kannst du sie folgende Routine verwenden:

· mplay_session_create(sesname,playnumb,playername) erstellt eine neue session in der aktuellen Verbindung. sesname ist eine Zeichenkette, welche den Namen der session angibt. playnumb ist eine Nummer, die die maximale erlaubte Anzahl der Spieler in diesem Spiel angibt (verwende die 0 für eine willkürliche Anzahl). playname ist dein Name als Spieler. Gibt wieder, ob erfolgreich.

In vielen Fällen wird der Spielername nicht benötigt und kann eine leere Zeichenkette sein. Desweiteren ist der Name der session nur relevant, wenn du den Leuten die Möglichkeit geben möchtest , sich auszusuchen, welcher session sie beitreten möchten.

Demnach muss eine Instanz des Spiels die session erstellen. Die andere(n) Instanz(en) des SPiels sollten dieser session beitreten. Dies ist geringfügig schwieriger. Zuerst musst du nachsehen, welche sessions verfügbar sind und dann diejenige aussuchen, der beigetreten werden soll. Es gibt drei dafür wichtige Routinen:
· mplay_session_find() sucht nach sessions, die noch Spieler aufnehmen können und gibt die Anahl der gefundenen sessions wieder.
· mplay_session_name(numb) gibt den Namen der session mit der Nummer numb wieder (0 ist die erste session). Diese Routine kann nur nach der vorherigen Routine aufgerufen werden.
· mplay_session_join(numb,playername) damit tritts du der session mit der Nummer numb bei (0 ist erste session). playername ist der Name des Spielers. Gibt wieder, ob erfolgreich.

Standardmässig ruft man mplay_session_find() auf, um alle verfügbaren sessions zu finden. Dann verwendest man entweder wiederholt mplay_session_name() , um sie dem Spieler anzuzeigen und

ihn eine Wahl treffen zu lassen oder tritt sofort der ersten session bei. (Beachte, dass das Finden der sessions etwas Zeit beansprucht. Rufe diese Routine also nicht in jedem step auf.)

Ein Spieler kann eine session mittels folgender Routine beenden:
· mplay_session_end() beendet die session für diesen Spieler.

Es ist nützlich zuerst die anderen Spieler darüber zu informieren aber es ist nicht grundsätzlich notwendig.

Demnach gibt der zweite room in unserem Spiel dem Spieler zwei Wahlmöglichkeiten: Entweder eine neue session zu erstellen oder einer bestehenden session beizutreten. Für den ersten Fall führen wir folgenden Programmcode im mouse event aus:

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

Beachte, dass wir eine globale Variable namens master auf true gesetzt haben. Der Grund dafür ist, dass wir im Spiel eine Unterscheidung zwischen dem Hauptspieler (er wird master genannt) und dem zweiten Spieler (er wird slave genannt) haben wollen. Der master wird für den Großteil des Spiels verantwortlich sein, während der slave sich einfach anschliesst.

Der zweite Fall ist das Beitreten zu einem bestehenden Spiel. So sieht der Programmcode aus.

```
{
    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.')
}
```

In diesem Spiel treten wir einfach der ersten verfügbaren session bei. Weil wir festgelegt haben, dass die maximale Anzahl der Spieler bei zwei liegt, kann kein weiterer Spieler aufgenommen werden.

Spielerbehandlung

Sobald der master eine session erstellt hat, müssen wir warten bis ein anderer Spieler beitritt. Es gibt drei Routinen, die sich mit Spielern befassen.
· mplay_player_find() sucht nach allen Spielern in der aktuellen session und gibt deren Anzahl wieder.
· mplay_player_name(numb) gibt den Namen des Spielers mit der Nummer numb wieder (0 ist der erste Spieler, welcher immer du selber bist). Diese Routine kann nur nach der vorherigen aufgerufen werden.
· mplay_player_id(numb) gibt die eindeutige id des Spielers mit der Nummer numb wieder (0 ist der ertse Spieler, welcher immer du selber bist). Diese Routine kann nur nach der ersten Routine aufgerufen werden. Diese id wird für das Senden und Empfangen von Nachrichten (messages) von und zu einzelnen Spielern verwendet.

In unserem dritten room warten wir einfach darauf, dass der zweite Spieler beitritt. Somit setzen wir ein object hinein und in dessen step event kommt folgender Programmcode:

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

(Der slave braucht eigentlich nicht in diesen room zu gehen.)

Handlungen synchronisieren

Nun da wir eine Verbindung aufgebaut haben, eine session erstellt haben und zwei Spieler drin sind, kann das eigentliche Spiel beginnen. Aber das bedeutet auch, dass jetzt das Nachdenken über die Kommunikation anfängt. Das Hauptproblem in jedem Multiplayer-Spiel ist die Synchronisation. Wie können wir sicherstellen, dass die Spieler das exakt gleiche Abbild der Spielwelt sehen? Dies ist entscheidend. Wenn ein Spieler den Ball in unserem Spiel an einer anderen Stelle sieht, als der andere Spieler, können seltsame Dinge geschehen. (Im schlimmsten Fall trifft für einen Spieler der Ball auf den Schläger während er für den anderen Spieler verfehlt.) Die Spiele verlieren recht einfach ihre Synchronität, was in den meisten Spielen Chaos hervorruft.
Was die Sache verschlimmert, ist dass wir das Ganze mit einem begrenztem Maß an Kommunikation regeln müssen. Wenn du beispielsweise über ein MoDem spielst, können die Verbindungen sehr träge sein. Demnach sollte man das Kommunikationsaufkommen so niedrig wie möglich halten. Zusätzlich können noch Verzögerungen auftreten, wenn die Daten auf der Gegenseite ankommen. Schliesslich gibt es noch die Möglichkeit, dass die Daten verloren gehen und nie das andere Ende erreichen.
Wie man am besten mit diesen Problemen umgeht hängt davon ab, welche Art von Spiel du entwickelst. In rundenbasierten Spielen wirst du wahrscheinlich einen unterschiedlichen Mechanismus verwenden, als in einem schnellen Action-Spiel.
Game Maker bietet zwei Mechanismen zur Kommunikation: shared data und messages. Shared data ist der am einfachsten zu verwendende Mechanismus. Das Senden von messages ist vielseitiger, setzt aber voraus, dass du besser verstehst, wie Kommunikation funktioniert.

Shared data Kommunikation

Shared data Kommunikation ist wahrscheinlich der einfachste Weg das Spiel zu synchronisieren. Jedwede Kommunikaton ist abgeschirmt von dir. Es gibt einen Satz von 1000000 Werten, der allen beteiligten Spielen geläufig ist (so zum Beispiel beiden: dem master und dem slave). Jedes Spiel kann Werte lesen und setzen. Game Maker stellt sicher, dass jedes Spiel die gleichen Werte sieht. Ein Wert kann sowohl eine reelle Zahl, alsauch eine Zeichenkette sein. Es gibt nur zwei Routinen:
· mplay_data_write(ind,val) schreibt den Wert val (string oder real) in die Position ind (ind zwischen 0 und 1000000).
· mplay_data_read(ind) liest den Wert von der Position ind (ind zwischen 0 und 1000000).
Anfänglich sind alle Werte auf 0 gesetzt.

Um diesen Mechanismus zu verwenden musst du festlegen, welcher Wert wofür benutzt wird und wer ihn verändern darf. Vorzugsweise sollte nur eine Instanz des Spiels den Wert verändern dürfen. Desweiteren ist es ratsam, nur die ersten Positionen zu verwenden, um Speicherplatz zu sparen.

In unserem Fall gibt es vier wichtige Werte: Die y-Position des Schlägers (bat) des masters, die y-Position des Schlägers des slaves und die x und y Position des Balles. Demnach verwenden wir die Position 1, um die y-Koordinate des master-Schlägers anzugeben, Position 2 wird für die y-Koordinate des slave-Schlägers verwendet und so weiter. Es ist offensichtlich, dass der master den Wert seines Schlägers schreibt und das der slave den seines Schlägers schreibt. Wir legen fest, dass der master verantwortlich für die Werte des Balles ist. Der slave zeichnet den Ball einfach an die richtige Stelle in jedem step.

Wie wird das jetzt realisiert? Zuersteinmal sollte der master-Schläger (der linke) nur vom master kontrolliert werden. Das bedeutet, dass in den up und down arrow keyboard events, welche ihn kontrollieren, wir sicherstellen müssen, dass die Position nur geändert wird, wenn wir master sind. Demnach sähe der Programmcode für up arrow etwa so aus:

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

Für den Schläger des slave schreiben wir einen ähnlichen Programmcode. Nun müssen nur noch beide sicherstellen, dass sie den Schläger des anderen an der richtigen Stelle plazieren. Wir erledigen das im step event. Wenn wir der slave sind, müssen wir im step event des master-Schlägers die Position setzen. Somit verwenden wir folgenden Programmcode:

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

Ähnlich für den slave-Schläger.

Schliesslich müssen wir für den Ball sicherstellen, dass er abprallt, wenn er auf eine Wand trifft oder auf einen Schläger. Konkret muss das nur für den master gemacht werden. Um die Ballposition dem slave mitzuteilen, fügen wir folgenden Programmcode im step event ein:

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

Damit ist die grundlegende Kommunikation abgehandelt. Was noch bleibt ist der Spielstart, der Punktestand und so weiter. Auch hier lassen wir die Kontrolle darüber am Besten beim master. Somit entscheidet der master, wann ein Spieler verliert, wann der Punktestand verändert wird (dafür können wir eine zusätzliche Position verwenden, um dies dem slave mitzuteilen) und wann ein neuer Ball ins Spiel kommt. Du kannst dir das ganze Spiel pong1.gm6 für weitere Details anschauen.

Beachte, dass mit dem beschriebenen Kommunikationsschema die Spiele trotzdem noch ein wenig unsynchron sein können. Dies ist normalerweise kein Problem. Da kannst dies vermeiden durch ein Synchronisierungsobjekt, welches einige Werte verwendet, die sicherstellen, dass beide Spiele bereit sind, bevor irgendwas auf dem Bildschirm dargestellt wird. Dies sollte mit vorsicht benutzt werden, da es Probleme verursachen kann - beispielsweise Blockierungen, wo beide Seiten auf die andere Seite warten.
Sei dir im klaren darüber, dass ein später hinzukommender Spieler nicht alle gemeinsamen Daten übermittelt bekommt (das würde zuviel Zeit kosten), sondern nur die Änderungen, seit er dabei ist.

Messaging

Der zweite Kommunikationsmechanismus, den Game Maker unterstützt, ist das Senden und Empfangen von messages (Nachrichten). Ein Spieler kann messages zu einem oder allen Spieler senden. Spieler können sehen, ob messages angekommen sind und entsprechende Maßnahmen ergreifen. Messages können in einem garantierten Modus versendet werden, wo du sicher sein kannst, dass sie ankommen (aber das kann langsam sein) oder in einem nicht-garantierten Modus, was schneller ist.

Wir werden zuerst messsages verwenden, um einige Klangeffekte unserem Spiel hinzuzufügen. Wir brauchen eine Sound für das Auftreffen des Balls auf den Schläger, für's Auftreffen des Balls an der Wand und für das erzielen eines Punktes. Nur der master kann diese Ereignisse ermitteln. Demnach muss der master entsheiden, ob ein Sound abgespielt werden soll. Es ist einfach, dies für das eigene Spiel zu machen. Es spielt einfach den Sound ab. Aber es muss dem slave auch mitteilen, dass der Klang abgespielt werden soll. Wir könnten shared data dafür verwenden aber es ist ziemlich kompliziert. Das Verwenden einer message ist einfacher. Der master sendet einfach eine message an den slave, damit dieser den Sound abspielt. Der slave horcht nach messages und spielt den korrekten Klang ab, wenn er dazu aufgefordert wird.
Die folgenden Nachrichten-Routinen existieren:
· mplay_message_send(player,id,val) sendet eine message zu dem angegebenen Spieler (entweder ein Bezeichner oder ein Name; verwende die 0, um die message an alle Spieler zu versenden). id ist ein ganzzahliger message-Bezeichner und val ist ein Wert (entweder ein reeller oder eine Zeichenkette). Die message wird im nicht-garantierten Modus gesendet.

· mplay_message_send_guaranteed(player,id,val) sendet eine message zu dem angegebenen Spieler (entweder ein Bezeichner oder ein Name; verwende die 0, um die message an alle Spieler zu versenden). id ist ein ganzzahliger message-Bezeichner und val ist ein Wert (entweder ein reeller oder eine Zeichenkette). Die message wird im garantierten Modus gesendet.
· mplay_message_receive(player) empfängt die nächste message aus der Warteschlange, welche vom angegebenen Spieler stammt (entweder ein Bezeichner oder ein Name). Verwende die 0 für messages von irgendeinem Spieler. Die Routinen geben wieder, ob tatsächlich eine neue message ansteht.
Wenn dem so ist, kannst du folgende Routinen verwenden, um an den Inhalt zu gelangen:
· mplay_message_id() Gibt den Bezeichner der letzten empfangenen message wieder.
· mplay_message_value() Gibt den Wert der letzten empfangenen message wieder.
· mplay_message_player() Gibt den Spieler, welcher die letzte empfangene message gesendet hat wieder.
· mplay_message_name() Gibt den Namen des Spieler, welche die letzte empfangene message gesendet hat wieder.
· mplay_message_count(player) Gibt die Anzahl der anstehenden messages in der Warteschlange des Spielers wieder (Verwende 0, um alle messages zu zählen).

Ein paar Anmerkungen an dieser Stelle. Wenn du nur einem bestimmten Spieler eine message senden möchtest, musst du die eindeutige id des Spielers kennen. Wie schon erwähnt erreichst du dies mit der Funktion mplay_player_id(). Diese Spieler-id wird auch benötigt, wenn man messages eines bestimmten Spielers empfangen will. Alternativ kannst du den Namen des Spielers als Zeichenkette angeben. Falls mehrere Spieler denselben Namen haben, bekommt nur der erste die message.
Du wunderst dich vielleicht, warum jede message einen ganzzahligen Bezeichner besitzt. Der Grund dafür ist, dass es deiner Anwendung hilft verschiedene Arten von messages zu versenden. Der Empfänger der message kann die Art der message ermitteln indem er den Bezeichner prüft und dann entsprechend reagieren. (Da messages nicht garantiert ankommen, kann das Senden von Bezeichner und Wert in unterschiedlichen messages ernsthafte Probleme verursachen.)
Für das Abspielen der Sounds wählen wir folgenden Ansatz. Wenn der master entscheidet, dass der Ball auf den Schläger trifft, wird folgender Programmcode ausgeführt:

```
{
    if (!global.master) exit;
    sound_play(sound_bat); // spiele selber den sound ab
    mplay_message_send(0,100,sound_bat); // sende es zum slave
}
```

Im step event des controller objects passiert folgendes:

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

Das heisst, es prüft ob eine message ansteht und wenn dem so ist, schaut es nach, welcher Bezeichner dazugehört. Wenn der Bezeichner 100 entspricht, wird der sound, der als Wert übermittelt wurde, abgespielt.

Allgemeiner formuliert hat dein Spiel ein controller Objekt in seinen rooms, welches im step event folgendes macht:

```
{
    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.
    }
}
```

Es ist von immenser Wichtigkeit, das zu verwendende Kommunikationsprotokoll (das heisst, festzulegen wer wann welche messages sendet und wie die Gegenseite darauf zu reagieren hat) sorgfältig zu entwerfen. Erfahrung und das Betrachten von Beispielen anderer hilft ungemein dabei.

Dead-reckoning (keine Ahnung, wie ich das treffend übersetzen kann)

Das oben beschriebene pong-Spiel hat ein ernsthaftes Problem. Wenn mal ein "Schluckauf" in der Kommunikation auftritt, steht der Ball beim slave zeitweise still. Keinerlei neue Koordinaten kommen an und daher bewegt er sich nicht. Dieses Problem tritt insbesondere dann auf, wenn die Distanz zwischen den Computern sehr groß ist und/oder die Kommunikation lahm ist. Je umfangreicher die Spiele werden, desto mehr Werte werden benötigt, um den Zustand des Spieles zu beschreiben. Wenn du viele Werte pro step änderst, muß eine Menge an Informationen übertragen werden. Dies kann viel Zeit kosten, was dein Spiel langsamer macht oder sogar dazu führt, dass manche Dinge nicht mehr synchron sind.

Ein erster Schritt die Kommunikation mittels shared data zu beschleunigen, ist auf den garantierten Modus zu verzichten. Dies kann durch folgende Funktion erreicht werden:
· mplay_data_mode(guar) legt fest, ob der garantierte Übertragungsmodus für shared data angewendet werden soll oder nicht. guar sollte entweder true (voreingestellt) oder false sein.

Eine bessere Technik, um dieses Problem zu beheben wird dead-reckoning genannt. Hier senden wir Informationen nur von Zeit zu Zeit. In der Zwischenzeit schätzt das Spiel selber, was passiert - basierend auf den Informationen, die es schon hat.

Wir werden dies nun für unser pong-Spiel verwenden. Statt die Ballposition in jedem step zu senden, versenden wir zusätzlich noch Information über die Geschwindigkeit und die Richtung des Balls. Jetzt kann der slave den Großteil der Berechnungen selber ausführen. So lange wie keine neue Information vom master ankommt, berechnet er einfach wohin sich der Ball bewegt.
In diesem Fall werden wir nicht shared data verwenden sondern messages. Wir verwenden

messages, welche die Ballposition angeben, die Ballgeschwindigkeit, die Schlägerposition und so weiter. Der master sendet solche Informationen immer, wenn sich etwas verädert. Das controller Objekt im slave lauscht nach diesen messages und setzt die korrekten Parameter. Wenn aber der slave nichts empfängt, bewegt er den Ball einfach weiter. Falls er einen kleinen Fehler dabei macht, wird die Position durch eine spätere message des masters wieder korrigiert. Somit schreiben wir beispielsweise folgenden Programmcode in das step event des Balls:

```
{
    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);
}
```

Im step event des controller object steht folgender Programmcode:

```
{
    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);
    }
}
```

Beachte, dass die messages nicht zwingend im garantierten Modus gesendet werden müssen. Wenn von Zeit zu Zeit eine verpasst wird, ist das kein ernsthaftes Problem. Du findest das geänderte Spiel in der Datei pong2.gm6.

Jetzt wirst du enttäuscht sein, wenn du das Spiel pong2 ausführst. Da sind immer noch Ruckler. Wodurch werden diese verursacht? Der Grund mag in einer langsamen Übertragung liegen. Das bedeutet, dass der slave messages empfängt, die schon vor längerer Zeit gesendet worden sind. Infolgedessen setzt er die Ballposition ein Stück zurück und dann, wenn er eine neue message erhält, wieder ein Stück vor. Laß uns einen dritten Versuch machen, den du in der Datei pong3.gm6 findest. In diesem Fall tauschen wir nur Informationen aus, wenn der Ball auf einen Schläger trifft. Somit wird der Rest der Bewegung mittels dead-reckoning realisiert. Der master ist dafür verantwortlich, was auf der Seite des master-Schlägers geschieht und der slave dafür, was auf der anderen Seite passiert. Während der Ball von der einen Seite zur anderen fliegt, werden nun keine messages mehr ausgetauscht.

Wie du sehen wirst, fliegt der Ball jetzt gleichmässig. Nur wenn der Ball auf einen Schläger trifft, kann es kurz ruckeln oder der Ball fliegt schon zurück, bevor er den gegnerischen Schläger berührt hat. Der Grund dafür liegt darin, dass dieser Mechanismus annimmt beide Spiele laufen mit exakt der gleichen Geschwindigkeit. Falls einer der beiden Rechner langsam ist, kann dies zu eiem Problem führen. Aber ein Ruckeln beim Auftreffen ist wesentlich leichter zu akzeptieren, als während der Bewegung. Um dieses letzte Problem zu vermeiden benötigst du einen fortgeschritteneren Mechanismus. Beispielsweise kannst du timing Informationen senden, so dass jede Seite weiß, wie schnell das Spiel auf der Gegenseite abläuft und entsprechende Korrekturmaßnahmen egriffen werden können.

Hoffentlich verstehst du jetzt, wie schwierig Synchronisation ist. Vielleicht fängst du sogar an zu schätzen, wie kommerzielle Spiele dies erreichen. Sie kämpfen mit exakt denselben Problemen.

Ein Chatprogramm

Als zweite Demonstration erstellen wir ein kleines Chatprogramm. Hier erlauben wir eine beliebige Anzahl an Spielern. Desweiteren erlauben wir auch mehrere sessions und lassen dem Spieler die Wahl eine bestimmte auszuwählen. Wir werden messages mit Zeichenketten verwenden, um den geschriebenen Text zu versenden.

Wir verwenden eine etwas komplizierteren Mechanismus, um die Verbindung herzustellen. Der erste room hat nun vier Wahlmöglichkeiten für die vier verschiedenen Verbindungstypen. Aber er stellt die Verbindungen nicht her, sondern schreibt nur einen Wert in die globale Variable connecttype. Im zweiten room kann der Spieler wieder entscheiden, ob er ein Spiel erstellen will (oder konkret eine Chatbox) oder beitreten. Nur hier wird die Verbindung initialisiert. Abhängig davon, ob der Spieler einem Spiel beitritt oder eines erstellt, werden einige Fragen gestellt.

Nachdem die Verbindung erfolgreich hergestellt wurde, wird die session erstellt bzw. ihr beigetretend. Diesmal wird der Spieler nach seinem/ihrem Namen gefragt, damit die Spieler identifiziert werden können. Der Teil zum Beitreten ist diesmal etwas komplizierter. Wir erstellen ein Menü mit allen verschiedenen verfügbaren sessions aus denen der Spieler eine wählen kann.

Normalerweise endet eine session, wenn das erstellende Spiel beendet wird. Bei einem Chatprogramm ist dies nicht unbedingt das was du willst. Die anderen Spieler sollten weiterhin chatten können. Dies kann durch folgende Funktion erreicht werden:
· mplay_session_mode(move) setzt, ob der session host auf einen anderen Computer übergeht, wenn der aktuelle beendet wird. move sollte entweder true oder false (voreingestellt) sein.

Den gesamten Mechanismus der ersten beiden rooms willst du wahrscheinlich für deine eigenen Spiele wiederverwenden, weil es in weiten Teilen oft immer gleich ist.

Danach gehen wir weiter zum chatbox room. Hier gibt es nur ein controller Objekt, welches die ganze Arbeit erledigt. Ich werde die Details zur Benutzereingabe und Darstellung der letzten Zeilen nicht erläutern. Es ist alles in einigen scripts enthalten, die du wiederverwenden kannst, wenn du möchstest. Es ist alles ziemlich einfach (sofern du weisst, wie man in programmiert). Der einzige Teil, der noch interessant ist der, wann immer ein Spieler die Enter-Taste drückt, wird die getippte Zeile zu allen Spielern gesendet - mit dem Namen des Spielers vorangestellt. Auch wenn ein Spieler hinzukommt oder geht sendet er eine message an alle Spieler, die angibt, was er machte.
Betrachte die Datei chat.gm6 für Details.

Zusammenfassung

Die Mehrspieler-Möglichkeiten des Game Makers erlauben es raffinierte Multiplayer-Spiele zu entwickeln. Allerdings helfen dir die Funktionen nur bei der Kommunikation auf einem niedrigen Niveau. Du musst schon selber den zu verwendenen Kommunikationsmechanismus entwerfen. Dies ist ein sorgfältiger Vorgang. Er sollte entwickelt werden, während du das Spiel planst. Es ist sehr schwierig effektive Multiplayerfunktion nachträglich hinzuzufügen. Hier sind einige allgemeine Richtlinien:

· Für die meisten einfachen Spiele ist ein master-slvae-Mechanismus am einfachsten zu realisieren
· Lege sorgfältig fest, wer für welche Daten verantwortlich ist
· Verwende dead-reckoning wann immer möglich
· Versuche sowenig wie möglich auf den garantierten Kommunikationsmodus angewiesen zu sein

Anmerkung des Übersetzers:

Das Original Tutorial findest Du hier.
Da meine letzte Englischstunde schon über 15 Jahre her ist und ich selber noch kein Multiplayer-Spiel gemacht habe, ist es nicht auszuschliessen, dass Fehler in der Übersetzung sind. Ich hoffe aber, dass ihr mich bei Fehlersuche/-behebung unterstützt.
Vielleicht spornt es ja einige von euch an, auch mal was der Community zurückzugeben (Ich spreche damit insbesondere unsere Englisch-Profis an, die sich aufgrund von Motivationsmangel bisher zurückgehalten haben)
Eine Bitte noch: Postet hier keine Fragen zum Thema selber, sondern lediglich Feedback zur Übersetzung - Danke!