Mini tutorial - warcaby

Bardzo znaną grą logiczną są warcaby. Jest to uproszczenie nieco bardziej skomplikowanej gry, szachów. W tym tutorialu mam zamiar opisać proces tworzenia takiej gry w Game Makerze. Dzięki niemu stworzenie jej zajmie nam niewielką ilość czasu, i, mam nadzieję, nie będziemy mieli żadnego problemu przy jej tworzeniu.

Standardy

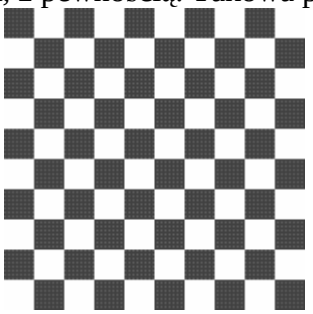
Niestety, istnieją setki różnych wersji warcabów, dlatego też tworząc naszą, możemy dopasować ją do naszych potrzeb. Ustalmy sobie pewne właściwości:

- **ilość pól: 100 (10 x 10), przy czym poruszamy się tylko po 50**
- **„damka" – może chodzić we wszystkie strony, zbijając maksymalnie jednego pionka (pionkiem może być także wroga damka)**
- **pionek, jak i dama chodzą na skosy, pion porusza się do przodu, może bić do tyłu**
- **celem gry jest zbić wszystkie wrogie pionki (nie, to nie jest rosyjska wersja „anty – warcaby")**

I to by było na tyle. W naszą grę będzie można grać tylko z drugim graczem, na jednym komputerze, lub na dwóch.

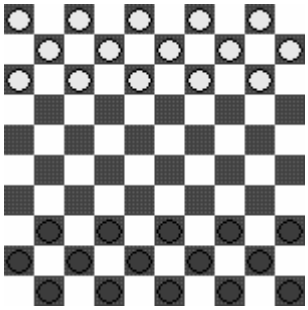
Teoria

Jeżeli mamy zamiar zrozumieć samą teorię, to będzie to z pewnością najtrudniejszy rozdział. Mogę dodać, że w owym podpunkcie dodamy także potrzebne grafiki. Wspomniałem już, że gra będzie na planszy 10 x 10? Tak, z pewnością. Takowa plansza będzie wyglądać tak:



1. schemat planszy do warcabów

Każde pole ma takie same wymiary, za pewne domyślicie się, czego użyjemy, by móc w łatwy sposób określać, jaki pionek się na każdym z nich znajduje. Tak, będą to tablice dwuwymiarowe. Ale, wracając do planszy, ustalmy sobie, jak będą rozmieszczone pionki na początku gry. Ja to widzę tak:



2. ustawienie pionków – schemat

W grze użyjemy stałych, które nam ułatwią nieco zadanie:

- **FLD_EMPTY – puste pole**
- **FLD_WHITE – biały pionek**
- **FLD_BLACK – czarny pionek**
- **FLD_DAMWHITE – biała dama**
- **FLD_DAMBLACK – czarna dama**

Wartość każdej stałej, to jej kolejność na tej liście. FLD_EMPTY wynosi więc -1, FLD_WHITE 0, a FLD_DAMBLACK 3. Dlaczego wymyśliłem takie głupie wartości? Otóż mam zamiar stworzyć jeden sprite z tymi wszystkimi pionkami – każda wartość odpowiada klatce animacji. Dodatkowo, przeciwieństwem zera jest jeden, a to ma akurat wielkie znaczenie. Możemy już przygotować nasze grafiki do gry, będą miały wymiary 60 x 60 pikseli. Tak wyglądają nasze pionki i damki:

3. grafiki pionków

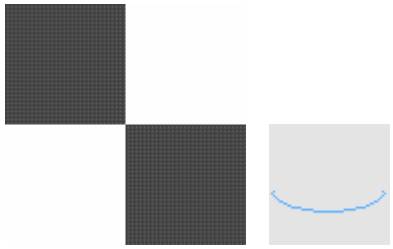
Stwórz więc już sprite'a na podstawie powyższych grafik i nazwij go sPionek.

Omówmy teraz, jak będzie działała mechanika gry, tzn. skąd będzie komputer wierdział, że nasz ruch jest poprawny. Zastosujemy ciekawą metodę: odnajdziemy wszystkie możliwe ruchy, zapiszemy je do tablicy, a gdy już zrobimy jakiś ruch, przeszukamy tablicę. Nie należy wtedy także zapomnieć o unicestwieniu zbitego pionka.

Pamiętajmy, że jak zbijamy pionka, to należy dać nam jeszcze jedną szansę i sprawdzić, czy możemy zrobić wielokrotne bicie, czyli czy możemy zbić kilka pionków na raz, jeśli nie, to zmieniamy turę. Czym jest tura? Jest to tak naprawdę zmienna, która ma wartość 0 lub 1. Zero dla „gracza białego", jeden dla „czarnego".

Teraz zagadka. Jak dwoma pętlami wstawić początkowe pionki? Wystarczy wykorzystać ten fakt, że w białych, jak i czarnych pionkach w rzędzie parzystym pionki znajdują się na parzystych polach, zaś w rzędzie nieparzystym na nieparzystych. Jest to ciekawa cecha, która zaoszczędzi nam wielu linijek kodu.

Dodajmy jeszcze kilka grafik – grafika tła i aktualnie zaznaczonego pionka, czyli back oraz sAktualny:



4. dodatkowe grafiki

Robimy grę!

Nazwa tego rozdziału wydaje się nieco śmieszna, jednak określenie procesu tworzenia tej gry dwoma słowami wydaje się trafnym. Przechodzimy do tworzenia obiektów i wykorzystania w kodzie ustalonych przez nas stałych i grafik. Zaczynajmy!

Najpierw stwórzmy obiekt *Controller*. On będzie odpowiedzialny za całą rozgrywkę. Stwórzmy także room o wielkości 600 x 600 (10 x 60 = 600) i ustaw dodane przez ciebie tło. Przejdźmy do tworzenia tablicy i pionków. Są to trzy pętle *for*, których tłumaczyć raczej nie muszę – by sprawdzić, czy liczby są podzielne, wykorzystałem resztę z dzielenia przez dwa, używając **mod**. Kod:

```
for( i = 0; i < 10; i += 1; )
    for( j = 0; j < 10; j += 1; )
        field[ i, j ] = FLD_EMPTY;

for( i = 0; i < 5; i += 1; )
    for( j = 0; j < 3; j += 1; )
        field[ i * 2 + j mod 2, j ] = FLD_WHITE;

for( i = 0; i < 5; i += 1; )
    for( j = 0; j < 3; j += 1; )
        field[ 1 + i * 2 - j mod 2, 9 - j ] = FLD_BLACK;
```

Dodajmy jeszcze trzy zmienne – turn, czyli tura, curx oraz cury. Są to współrzędne zaznaczonego przez nas pionka.

```
turn = 1;
curx = -1;
cury = -1;

change_turn();
```

Grę zaczynają pionki białe, jednak nadaliśmy wartość 1, ponieważ funkcja change_turn, która ustala ruchy, zmienia także turę. Teraz przejdźmy do rysowania – używając ponownie pętli for wyświetlamy poszczególne klatki animacji. Rysujemy także zaznaczenie wybranego przez nas sprite'a:

```
for( i = 0; i < 10; i += 1; )
    for( j = 0; j < 10; j += 1; )
        if ( field[ i, j ] != FLD_EMPTY )
            draw_sprite( sPionek, field[ i, j ], i * 60, j * 60 );

if ( curx != -1 )
    draw_sprite( sAktualny, 0, curx * 60, cury * 60 );
```

Nadszedł czas na wypełnienie funkcji change_turn. Stwórz więc takową. Będziemy umieszczali poszczególne fragmenty kodu i analizowali je. Pierw:

```
turn = !turn;

count = 0;

add = false;

tkill = false;
kill = false;

tkx = 0;
tky = 0;
tgx = 0;
tgy = 0;
tmx = 0;
tmy = 0;
```

Zmieniamy turę pierwszą linijką, używając operatora logicznego ! (not). Później nadajemy zmiennej Mount wartość zero – jest to bowiem ilość możliwych ruchów, jednym słowem – wyzerowujemy tablicę. Później mamy zmienną add – ma ona wartość true lub false i decyduje o tym, czy dodajemy ruch, czy nie. Mamy jeszcze dwie zmienne – tymczasową *tkill* i nie, *kill*. Te zmienne decydują o tym, czy jest przymus bicia, czy nie. Dodatkowo dodajemy sześć zmiennych określających współrzędne: pionka zbitego, punktu docelowego, punktu startowego. Na podstawie tych zmiennych będziemy w stanie uzupełniać tablice z ruchami.

```
for( i = 0; i < 10; i += 1; )
{
    for( j = 0; j < 10; j += 1; )
    {
```

Przechodzimy iteratorem po każdym polu – żadna nowość.

```
        tmx = i;
        tmy = j;
```

Oczywiste jest chyba, że każde pole, po którym przejdziemy, będzie polem początkowym w ruchu.

```
        if ( field[ i, j ] == turn )
        {
```

Jeżeli wartość pola jest równa aktualnej turze, czyli jeżeli napotkaliśmy na pionka naszego koloru. Nie uwzględniamy damki.

```
            for( z = 0; z < 4; z += 1; )
            {
                add = false;

                _x = 0;
                _y = 0;

                switch ( z )
                {
                    case 0:
                        __x = 1;
```

```
            __y = 1;
        break;
        case 1:
            __x = -1;
            __y = -1;
        break;
        case 2:
            __x = -1;
            __y = 1;
        break;
        case 3:
            __x = 1;
            __y = -1;
        break;
    }
```

Wstawiamy cztero – krokową pętlę for. Cztero – krokową, ponieważ tyle jest możliwych ruchów zwykłym pionkiem. Przyjmujemy, że pionka nie dodamy, w każdym momencie można zmienić wartość zmiennej *add*, więc owe rozwiązanie nie jest głupie. Teraz zerujemy zmienne _x i _y. One są odpowiedzialne za kierunek – WS, WN, SE, czy NE. W każdym bądź razie jest to przesunięcie się o jedno pole na ukos. Na podstawie zmiennej z ustalamy ich wartości, czy są dodatnie, czy ujemne.

```
_x = max( 0, min( 9, __x + i ) );
_y = max( 0, min( 9, __y + j ) );

__x = __x * 2 + i;
__y = __y * 2 + j;
```

Teraz zwiększamy wartość zmiennej _x o i, a zmiennej _y o j. Ustalamy także wartości zmiennych __x i __y, które byłby współrzędnymi punktu docelowego, gdybyśmy zbijali jakiś pionek. Robimy ograniczenie zmiennych _x i _y, by nie mogły być większe niż 9 i nie mogły być mniejsze niż 0.

```
tkx = _x;
tky = _y;

tgx = _x;
tgy = _y;
```

Możliwe, że takie będą współrzędne punktu docelowego i zbitego pionka.

```
if ( ( field[ _x, _y ] == FLD_EMPTY ) && ( !tkill ) )
{
    if ( turn && _y < j )
        add = true;
    if ( !turn && _y > j )
        add = true;
}
```

Jeżeli pole jest puste, nie ma przymusu zbicia i pole jest przed nami – mamy możliwy ruch.

```
if ( field[ _x, _y ] == !turn || field[ _x, _y ] == !turn +
2 ) && ( __x >= 0 )
&& ( __y >= 0 ) && ( __x < 10 ) && ( __y < 10 )
{
    if ( field[ __x, __y ] == FLD_EMPTY )
```

```
            {
                if ( !tkill )
                    count = 0;

                add = true;

                tgx = __x;
                tgy = __y;

                tkx = _x;
                tky = _y;

                tkill = true;
            }
        }
```

Jeżeli na polu (_x, _y) znajduje się pionek wrogi lub wroga dama i pole za nim jest puste i nie znajduje się poza planszą – wyzerujemy wszystkie możliwe opcje, tworząc przymus zbicia wrogiego pionka. Ruch jest już ostatecznie dodany, zmieniamy punkt docelowy oraz pole zabitego pionka. Ot, cała filozofia.

```
        if ( add && ( ( !kill ) || ( tkill ) ) )
        {
            gx[ count ] = tgx;
            gy[ count ] = tgy;

            kx[ count ] = tkx;
            ky[ count ] = tky;

            mx[ count ] = tmx;
            my[ count ] = tmy;
            k[ count ] = false;

            if ( tkill )
            {
                kill = tkill;
                k[ count ] = true;
            }

            count += 1;
        }
```

Jeżeli ruch ma być dodany, i nie ma przymusu bicia (chyba, że mamy kolejny możliwy ruch, którym zbijemy pionka) nadajemy tablicom odpowiednie wartości, ustalamy, czy jest przymus bicia i zwiększamy ilość ustalonych ruchów.

```
    else if ( field[ i, j ] == turn + 2 )
    {
        for( z = 0; z < 4; z += 1; )
        {
            switch ( z )
            {
                case 0:
                    __x = 1;
                    __y = 1;
                break;
                case 1:
                    __x = -1;
                    __y = -1;
```

```
            break;
        case 2:
            __x = -1;
            __y = 1;
        break;
        case 3:
            __x = 1;
            __y = -1;
        break;
    }

    _x = i + __x;
    _y = j + __y;

    c = 0;
```

Możliwe, że napotkamy zamiast naszego pionka naszą damkę. W takim wypadku początek kodu wygląda znajomo, nieprawdaż? Dochodzi jednak nowa zmienna – *c*, która odpowiedzialna jest za ilość napotkanych pionków po drodze.

```
    if ( _x < 0 ) || ( _x > 9 ) || ( _y < 0 ) || ( _y > 9 )
        continue;
```

Ograniczenie, by indeks tablicy nie był ujemny, bądź za duży.

```
    do
    {
        if ( _x < 0 ) || ( _x > 9 ) || ( _y < 0 ) || ( _y > 9 )
            continue;

        if ( field[ _x, _y ] == !turn || field[ _x, _y ] ==
!turn + 2 )
        {
            c += 1;

            tkx = _x;
            tky = _y;

            _x += __x;
            _y += __y;

            if ( _x < 0 ) || ( _x > 9 ) || ( _y < 0 ) || ( _y >
9 )
                continue;

            if ( field[ _x, _y ] == FLD_EMPTY )
            {
                tk = true;
                kill = true;
            }

            continue;
        }

        mx[ count ] = i;
        my[ count ] = j;

        gx[ count ] = _x;
        gy[ count ] = _y;
```

```
                    k[ count ] = tk;

                    kx[ count ] = tkx;
                    ky[ count ] = tky;

                    count += 1;

                    _x += __x;
                    _y += __y;
                }
                until ( _x == 10 || _y == 10 || _x == -1 || _y == -1 || c
== 2 );
            }
        }
    }
}
```

Tym razem tworzymy pętlę, która kończy się, gdy napotkamy po drodze dwa pionki lub gdy dojdziemy do brzegu planszy. Każdy krok oznacza jeden ruch do przodu. Jeżeli napotkamy wroga, a za nim puste pole, to zwiększamy wartość zmiennej *c* o jeden, jednocześnie powodując przymus bicia. Każde przejście to także dodanie kolejnego ruchu. Damka może chodził do tyłu, nie więc podczas jej poruszania się ograniczenia występującego przy zwykłych pionach.

Udało nam się ukończyć najgorsze, czyli funkcję change_turn. Ona właściwie jest kluczem do stworzenia tej gry. Teraz pozostaje nam już tylko ją odpowiednio wykorzystać. Czas się wziąć za event Global left buton. A oto i kod:

```
_x = mouse_x div 60;
_y = mouse_y div 60;

if ( field[ _x, _y ] == turn || field[ _x, _y ] == turn + 2 )
{
    curx = _x;
    cury = _y;
}
else if ( curx != -1 )
{
    for( i = 0; i < count; i += 1; )
    {
        if ( kill && !k[ i ] )
            continue;

        if ( mx[ i ] == curx && my[ i ] == cury )
        && ( gx[ i ] == _x ) && ( gy[ i ] == _y )
        {
            field[ _x, _y ] = field[ curx, cury ];
            field[ curx, cury ] = FLD_EMPTY;

            curx = -1;
            cury = -1;

            if ( gy[ i ] == 9 - 9 * turn )
                field[ gx[ i ], gy[ i ] ] = turn + 2;

            if ( kill )
            {
                field[ kx[ i ], ky[ i ] ] = FLD_EMPTY;
                turn = !turn;
```

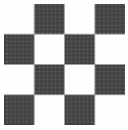
```
                change_turn();

                if ( !kill )
                    change_turn();
            }
            else
                change_turn();
        }
    }
}
```

Na początku obliczamy, na jakie pole kliknęliśmy na podstawie. Jeżeli klikniemy na naszego pionka, to zaznaczamy go – zmieniamy wartość zmiennej curx i cury. Jeżeli jednak już jakiegoś zaznaczyliśmy i nie kliknęliśmy na naszego piona, to sprawdzamy, czy istnieje taki możliwy ruch. Każdy ruch, który nie zbija, a jest przymus bicia, jest nieważny. Jeżeli trafimy na dobrą kombinację, to przesuwamy naszego piona, odznaczamy, a jeżeli dojdzie do końca planszy – zmieniamy w damkę. I teraz ostatni warunek: jeżeli był przymus bicia, tzn. atakowaliśmy, to niszczymy zbitego pionka, i, o dziwo, zmieniamy „ręcznie” turę, nie używając funkcji change_turn. Dlaczego? Ponieważ nie mamy zamiaru zmieniać tury (dwukrotna zmiana to brak zmiany), dając w ten sam sposób szansę na wielokrotne bicie. Sprawdzamy, czy jest przymus bicia. Jeżeli nie ma, to się myliliśmy – zmieniamy ponownie turę funkcją. Jeżeli jednak nie atakowaliśmy, to najzwyczajniej – zmieniamy turę skryptem.

I o to koniec procesu tworzenia warcabów. Życzę miłej gry ☺ !

Multiplayer

To nie musi być koniecznie koniec tego poradnika. Opiszę także, jak umożliwić rozgrywkę dwóch osób, na dwóch komputerach. Użyję do tego celu wbudowanych w Game Makera funkcji dotyczących multiplayera, ponieważ ich obsługa jest znacznie prostsza, niż popularnej biblioteki 39.dll. Na początku dodaj dwa dodatkowe tła:



5. tło wiadomości oraz bocznego panelu

Nazwij je back1 oraz back2. Dodaj także dowolną czcionkę o nazwie font. W roomie, w którym to wszystko się dzieje, dodaj pojedyncze tło, o położeniu poziomym (x) 600, a szerokość rooma zwiększ do 660 pikseli. Uzyskamy w ten sposób boczny panel. Jeżeli już to zrobiłeś, przystąpmy do tworzenie multiplayera.

Multiplayer jest o tyle ciekawy, że nie trzeba zawsze wysyłać wszystkich danych każdego obiektu, lecz informacje o tym, co aktualnie robimy. W naszym wypadku będą to po prostu informacje o współrzędnych różnych pól użytych w ruchu przeciwnika oraz o tym, czy zbił nam jakiegoś pionka i jaka jest aktualna tura oraz jakim pionkiem się poruszał. Łącznie 9 danych.

W takim razie zajmijmy się tworzeniem gry i dołączaniem do takowej. Stwórz dodatkowy room i nazwij go room_first. Stwórz także nowy obiekt – Creator, i wstaw go do tego rooma. W create daj mu taki kod:

```
message_background( back1 );
message_caption( true, "Wiadomość" );
message_text_font( "MS Sans Serif", 8, c_black, 0 );
message_button_font( "MS Sans Serif", 8, c_black, 0 );
message_input_font( "MS Sans Serif", 8, c_black, 0 );
message_button( sPrzycisk );
message_mouse_color( c_black );
message_alpha( 0.8 );
message_input_color( c_white );
```

Kod ten czyni ładniejszym standardowe okienko wiadomości Game Makera, niekonieczne, ale może się przydać. Kontynuując:

```
nick = get_string( 'Podaj swój nick:', 'gracz' );
select = show_message_ext( 'Chcesz stworzyć grę?', 'Tak', '', 'Nie' );

mplay_session_mode( true );

if ( select < 2 )
{
    mplay_init_tcpip( localhost );

    name = get_string( 'Podaj nazwę gry:', 'mojagra' );
    global.ses = mplay_session_create( name, 2, nick );
    show_message( 'Gra ' + name + ' została stworzona!' );
    global.color = 0;
    room_goto( room );
}
else
{
    mplay_init_tcpip( get_string( 'Podaj ip serwera: ', 'localhost' ) );

    global.ses = mplay_session_find();

    if ( global.ses )
    {
        show_message( 'Dołączono do gry ' + mplay_session_name( 0 ) + '!'
);
        global.color = 1;
        mplay_session_join( 0, nick );
        room_goto( room );
    }
    else
    {
        show_message( 'Nie znaleziono gry!' );
        game_end();
    }

}
```

Pierwsza linijka pyta nas o nasz nick w grze. Druga pyta się, czy tworzymy grę, czy dołączamy do stworzonej. Włączamy także tryb sesji. Sesją możemy nazwać grę stworzoną przez gracza. Jeżeli wybierzemy opcję pierwszą, to podłączamy się pod sieć lokalną używając protokołu tcpip. Podajemy nazwę gry (sesji), po czym wyświetlamy informację o sukcesie.

Zmieniamy nasz kolor na biały. Dzięki tej zasadzie, możemy wiedzieć, czy jesteśmy twórcą gry, czy podłączonym pod tą grę. Przechodzimy do głównego pokoju. Jeżeli podłączamy się do gry, to podajemy jej ip. Szukamy takowej sesji. Jeżeli znaleziemy sesję, wyświetlamy informację o powodzeniu i ustalamy kolor na czarny, po czym idziemy do pokoju room. Jeżeli nie uda się podłączyć – wyświetlamy informację o niepowodzeniu i wyłączamy grę.

Obiekt Creator zostawiamy w spokoju, włączamy edycję obiekty Controller. Pierwsze, co powinieneś zrobić, to wstawić taki kod w evencie Game End:

```
mplay_session_end();
mplay_end();
```

Kończy on sesję i jednocześnie używanie funkcji multiplayera. W evencie Draw dodajemy kod, który wyświetli na bocznym panelu nazwy graczy i ich kolory:

```
draw_set_font( font );
draw_set_color( c_red );
draw_set_halign( fa_center );
draw_set_valign( fa_center );

draw_sprite( sPionek, 0, 600, 0 );
if ( global.color == 0 )
    draw_text( 630, 90, mplay_player_name( 0 ) );
else
    draw_text( 630, 570, mplay_player_name( 0 ) );
draw_sprite( sPionek, 1, 600, 480 );

if ( mplay_player_find() == 2 )
{
    if ( global.color == 0 )
        draw_text( 630, 570, mplay_player_name( 1 ) );
    else
        draw_text( 630, 90, mplay_player_name( 1 ) );
}
```

Nick drugiego gracza wyświetlamy tylko w wypadku, gdy takowy istnieje. Zabierzmy się za edycję eventu Global Left Pressem. Oto nowy kod:

```
if ( mouse_x > 600 || turn != global.color || mplay_player_find() < 2 )
    exit;

_x = mouse_x div 60;
_y = mouse_y div 60;

if ( field[ _x, _y ] == turn || field[ _x, _y ] == turn + 2 )
{
    curx = _x;
    cury = _y;
}
else if ( curx != -1 )
{
    for( i = 0; i < count; i += 1; )
    {
        if ( kill && !k[ i ] )
            continue;

        if ( mx[ i ] == curx && my[ i ] == cury )
        && ( gx[ i ] == _x ) && ( gy[ i ] == _y )
```

```
        {
            mplay_message_send( mplay_player_name( 1 ), 0, kill );
            mplay_message_send( mplay_player_name( 1 ), 1, curx );
            mplay_message_send( mplay_player_name( 1 ), 2, cury );
            mplay_message_send( mplay_player_name( 1 ), 3, _x );
            mplay_message_send( mplay_player_name( 1 ), 4, _y );
            mplay_message_send( mplay_player_name( 1 ), 5, kx[ i ] );
            mplay_message_send( mplay_player_name( 1 ), 6, ky[ i ] );
            mplay_message_send( mplay_player_name( 1 ), 7, field[ _x, _y ]
);

            field[ _x, _y ] = field[ curx, cury ];
            field[ curx, cury ] = FLD_EMPTY;

            curx = -1;
            cury = -1;

            if ( gy[ i ] == 9 - 9 * turn )
                field[ gx[ i ], gy[ i ] ] = turn + 2;

            if ( kill )
            {
                field[ kx[ i ], ky[ i ] ] = FLD_EMPTY;
                turn = !turn;
                change_turn();

                if ( !kill )
                    change_turn();
            }
            else
                change_turn();

            mplay_message_send( mplay_player_name( 1 ), 8, turn );
        }
    }
}
```

W pierwszej linijce dodaliśmy po prostu ograniczenie – gdy nie ma drugiego gracza, lub naciśniemy na boczny panel – event jest nieważny. Jeżeli wykonamy jakiś ruch, przesyłamy wszystkie wcześniej wspomniane przeze mnie dane drugiemu graczowi (jego numer to 1, nasz 0). Tych danych jest oczywiście 9.

Trzeba je jednak odebrać – w Stepie możesz dać taki kod:

```
tmx = -1;

while ( mplay_message_receive( 0 ) )
{
    i = mplay_message_id();
    switch ( i )
    {
        case 0: ttk = mplay_message_value() ; break;
        case 1: tmx = mplay_message_value() ; break;
        case 2: tmy = mplay_message_value() ; break;
        case 3: tgx = mplay_message_value() ; break;
        case 4: tgy = mplay_message_value() ; break;
        case 5: tkx = mplay_message_value() ; break;
        case 6: tky = mplay_message_value() ; break;
        case 7: typ = mplay_message_value() ; break;
        case 8: tt = mplay_message_value() ; break;
```

```
    }
}

if ( tmx )
{
    field[ tmx, tmy ] = FLD_EMPTY;
    field[ tgx, tgy ] = typ;

    if ( ttk )
        field[ tkx, tky ] = FLD_EMPTY;

    turn = !tt;
    change_turn();
}
```

Najpierw tymczasowej zmiennej tmx dajemy wartość -1. Jeżeli ta wartość się nie zmieni, znaczy to, że przeciwnik nie wykonał żadnego ruchu. Pętlą while odbieramy id każdej wiadomości i na jego podstawie przypisujemy odpowiedniej zmiennej odpowiednią daną używając konstrukcji switch. Wykonanie ruchu wygląda podobnie jak w evencie poprzednim – zmieniamy pozycję pionka, a jeżeli gracz atakował, usuwamy odpowiedniego pionka. Zmieniamy „ręcznie” aktualną turę, po czym zmieniamy ją funkcją (znasz już owy sposób, przy którym tura zostaje ta sama, a my po prostu ustalamy możliwe ruchy).

I tak o to właśnie owy poradnik dobiega końca. Mam nadzieję, że pomógł ci on nieco oswoić to i owo, które na pewno może ci się przydać w tworzeniu swych gier ☺ .