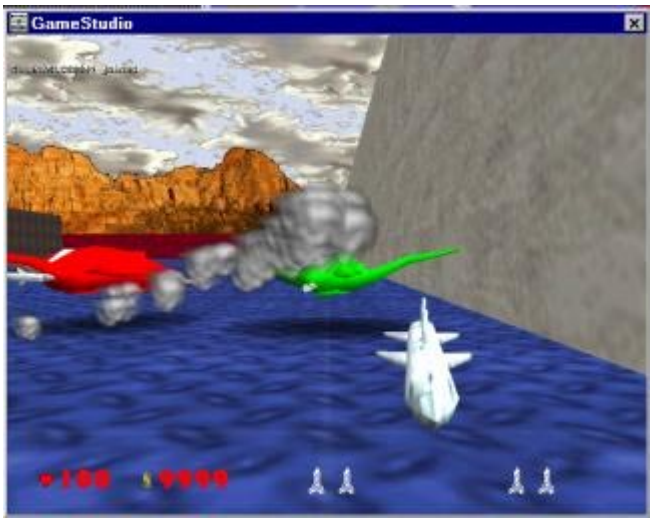
Les Ateliers de 3D Gamestudio

Créez votre premier shooter multi-joueur



par Alain BREGEON Octobre 2001

Les dernières nouvelles, les démonstrations, les mises à jour et les outils, aussi bien que le Magazine des Utilisateurs, le Forum des Utilisateurs et le concours annuel sont disponibles à la page principale GameStudio http://www.3dgamestudio.com/

Si vous êtes impatient :

Ou si vous hésitez sur la finalité de cet atelier, vous pouvez tester directement le niveau fini !

Nous supposons que vous avez au moins 2 machines de connectées (4 au maximum), l'une fera office de serveur et de client (mettre –sv –cl dans la ligne de commande) et au moins une autre sera client (-cl dans la ligne de commande).

Ouvrez WED, chargez le niveau level2.wmp, assignez-lui le script end_level.wdl et éclatez-vous.

Lorsque vous serez prêts à travailler, vous pourrez passer au chapitre suivant.

Avant propos de la part de l'auteur

Cher Lecteur,

J'ai produit cet atelier pour vous aider à répondre à la question "Comment faire un shooter multi-joueur avec 3DGameStudio ?". Cet atelier utilise des possibilités disponibles à partir de la version commerciale (5.10) ou supérieure.

Cet atelier, comme les autres ateliers avant cela, vise les utilisateurs qui ont un peu d'expérience de 3DGameStudio. Je suppose que vous avez travaillé les différents tutoriaux et savez comment employer les outils (WED, MED et WDL).

Pendant cet atelier l'accent sera mis sur les particularités du mode client / serveur et comment les utiliser plutôt que de donner de nombreuses explications pour comment faire une explosion ou comment créer son niveau. C'est pourquoi les différentes étapes ont été découpées en autant de niveau qui sont fournis. Et c'est également pourquoi vous constaterez qu'il y a de nombreuses imperfections mais elles ne nuisent pas à la bonne compréhension de ce tutorial

Nous supposons également que vous avez mis en pratique le premier atelier sur le mode multi-joueur, étape indispensable avant de démarrer celui-ci.

Pour une meilleure lisibilité de cet atelier, nous n'utiliserons que 2 machines. L'une en client serveur, l'autre en client. A la fin de ce tutorial, vous trouverez le niveau complet pour 4 joueurs.

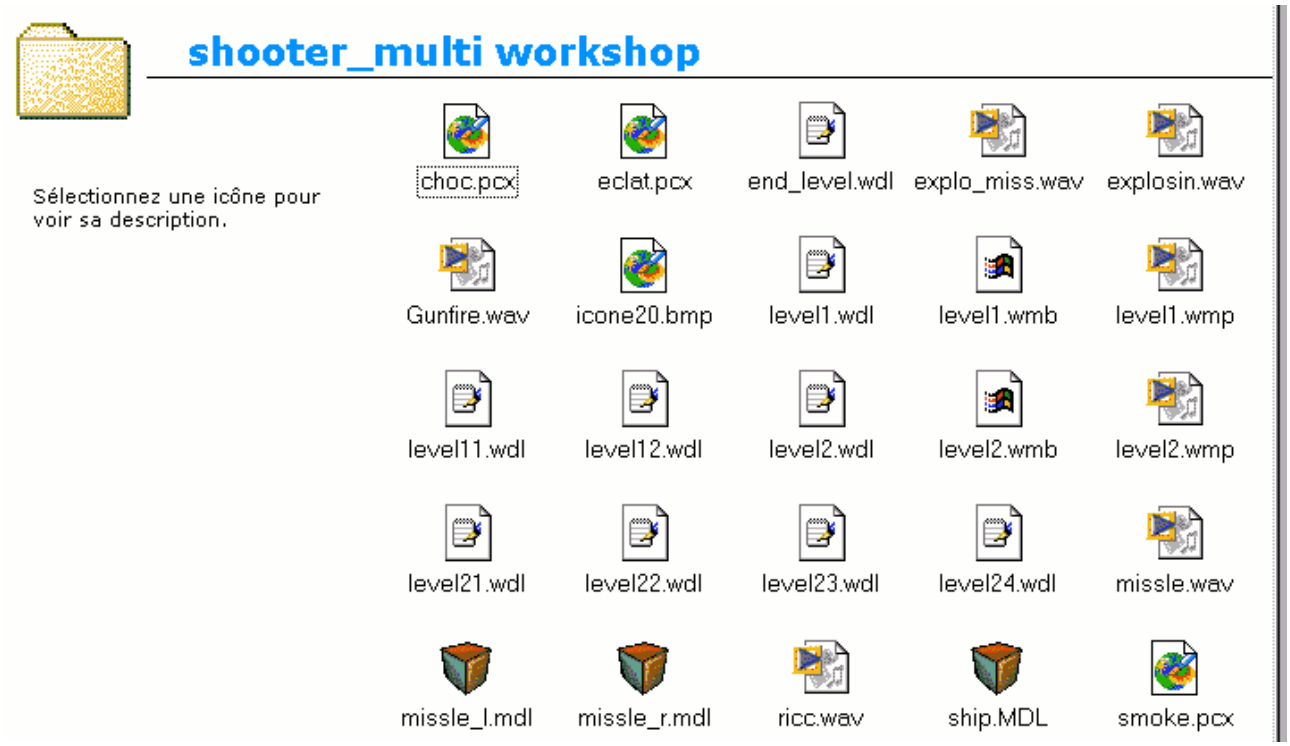
J'espère que vous trouverez cet atelier informatif et agréable.

Alain Brégeon
mailto:alainbregeon@hotmail.com

Obtenez la dernière version
Avant de commencer, assurez-vous d'avoir la dernière version de 3DGameStudio (5.10 ou au-dessus).

Préparez votre workspace
Créez un dossier appelé "shooter_multi Workshop" dans votre dossier GSTUDIO. C'est le répertoire ou vous stockerez tous les éléments du jeu.

Décompressez le contenu dans votre dossier. Votre dossier doit maintenant contenir au moins les fichiers:



répertoire shooter_multi Workshop

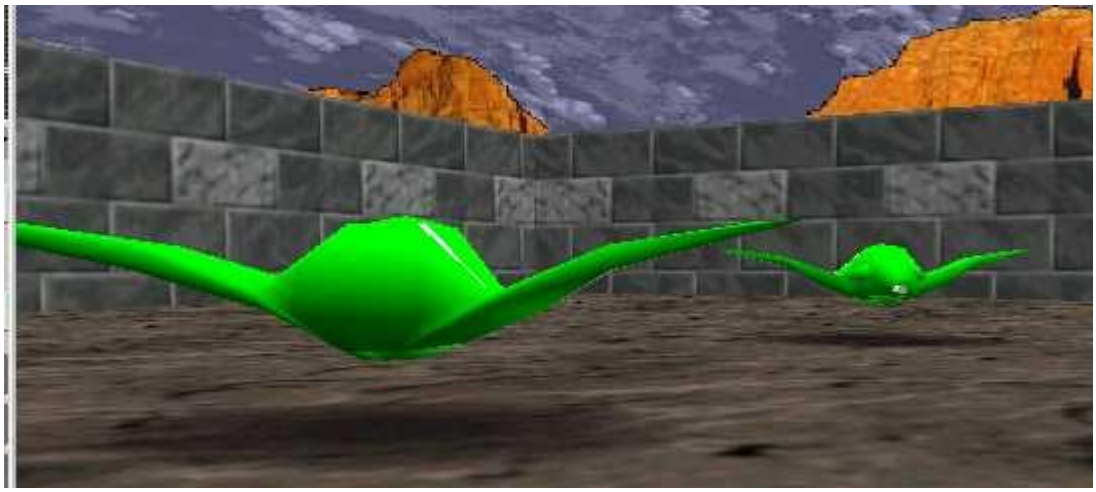
Notre premier shooter multi-joueur

Vous avez certainement déjà créé votre premier premier jeu multi-joueur, le tic tac toe. Il avait pour particularités d'être un jeu dit au tour, c'est à dire qu'un seul joueur joue à la fois, oubliant même la notion de client / serveur. Non seulement ce jeu était un jeu 2D mais en plus il fallait réfléchir pour gagner !!!

Avec le jeu que nous allons à présent créer ensemble, ce sera l'opposé. Le jeu sera en temps réel, en 3D, en mode client / serveur et pas besoin de réfléchir pour gagner. Il suffit de tirer, tirer et encore tirer. Et ça repose après l'intense réflexion nécessaire à la bonne compréhension du mode client / serveur.

Bien, assez parlé, entrons dans le vif du sujet.

Nous ouvrons WED et chargeons le niveau appelé level1, et nous l'exécutons sur les 2 machines. Pour l'instant il n'y a rien d'extraordinaire, nous pouvons nous déplacer comme nous le faisions dans office. Utilisez F7 pour les changements de vue et F10 pour quitter



Examinons les quelques instructions utiles que nous avons ajoutées :

Définition de notre joueur :

```
string player_mdl = <ship.mdl>;
```

Certaines fonctions doivent être définies avant leur utilisation, c'est le cas de la fonction player_client.

```
// function prototypes
function player_client();
```

Ensuite nous créons l'entité joueur sur chaque poste client en appelant client_move

```
// create the player entity on the client
ifdef CLIENT;
        client_move();     // enable multiuser mode
```

Puis nous attendons que la connexion soit établie

```
// wait until the level is loaded, and the connection is established
```

```
		while (connection == 0) { wait(1); }
```

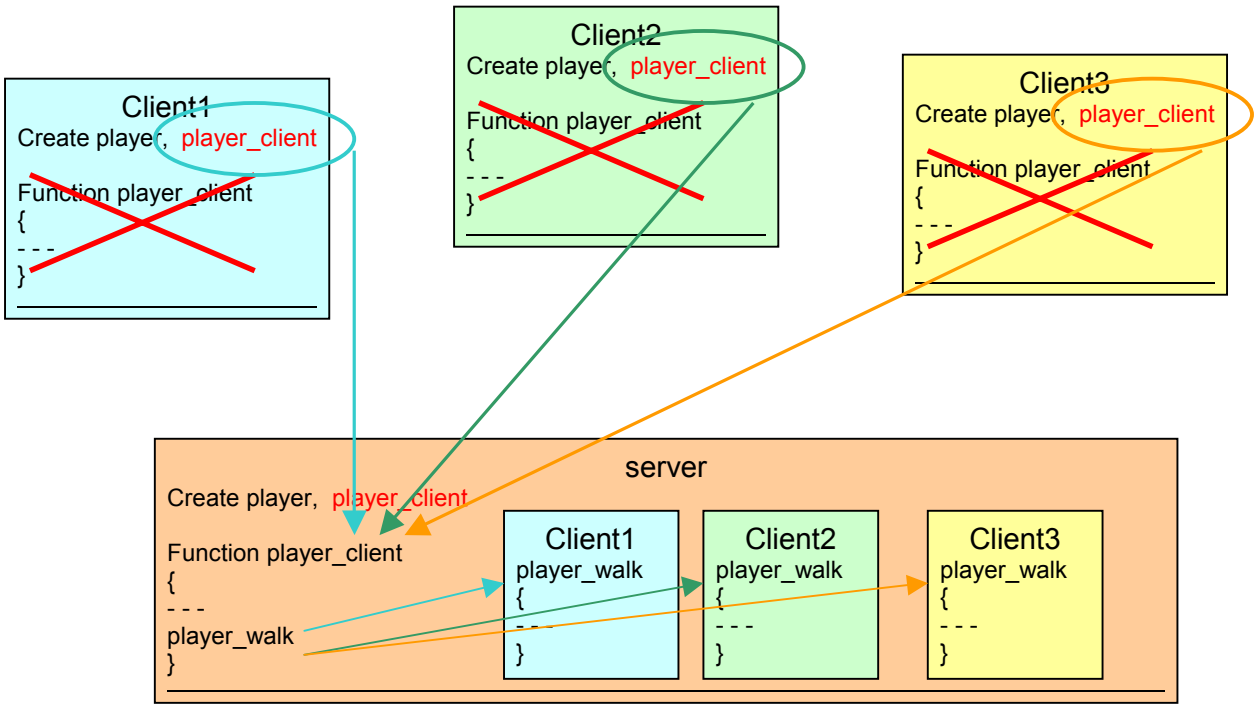
Et nous créons enfin notre joueur en définissant une position de départ aléatoire et ou nous retrouvons une définition classique pour un joueur.

```
// create a client entity at a random place
		temp.X = 60 + sys_seconds;
		temp.Y = 0;
		temp.Z = 0;
		player = ent_create(player_mdl,temp,player_client);
endif;


function player_client()
{
		MY.ENABLE_DISCONNECT = ON;
		MY.EVENT = _actor_connect;
		player_drive();
		if (MY.shadow == OFF) { drop_shadow(); }
}
```

Premier constat, les joueurs sont identiques, ça ne vas pas être facile de les différencier dans le jeu. Nous allons donc essayer de les différencier par une couleur de peau différente.

Pour tout comprendre

Avant d'aller plus loin, il est important de ne pas perdre de vue que nous sommes dans une logique client serveur, c'est à dire que les actions de toutes les entités sont exécutées sur le serveur et uniquement sur le serveur.



Démonstration de ceci : mettez en commentaire la ligne player_drive uniquement sur le level1.wdl qui se trouve sur la machine faisant office de serveur et exécutez sur les 2 machines. Aucun des joueurs ne se déplacent alors que le player_drive se trouve toujours sur la machine client.

Maintenant le contraire, remettons la ligne player_drive sur le serveur et enlevons là sur la machine client. Exécutons sur les 2 machines, les 2 joueurs se déplacent.

C'est bien la preuve que les 2 actions se déroulent sur le serveur.

Différencions nos joueurs

Revenons à notre changement de peau. Le modèle ship.mdl est fourni avec 4 peaux différentes (vert, rouge, jaune et bleu). Pour l'instant nous allons faire au plus simple, un compteur de peau qui s'incrémente de un à chaque passage dans la fonction 'player_client'. Nous restons dans le niveau level1 mais nous attachons le fichier level11.wdl (File ➔ Map Properties ➔ Script ➔ level11.wdl). A faire sur les 2 machines bien entendu

Les lignes ajoutées sont les suivantes :

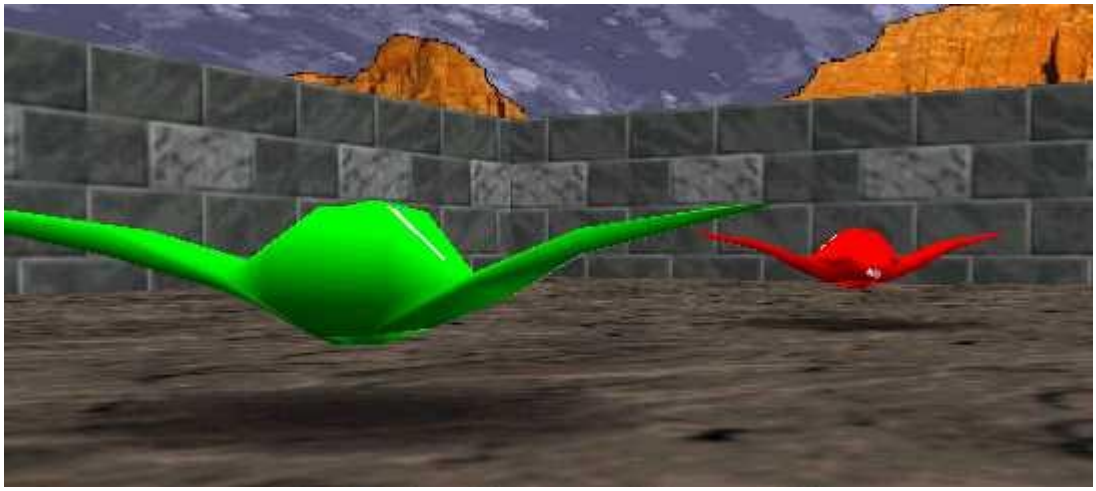
Définition de la variable :

```
var skin_client = 1;
```

Puis dans la fonction player_client :

```
my.skin = skin_client;
skin_client += 1;
if (skin_client > 4){skin_client =4;}
```

Et nous exécutons les niveaux sur les différentes machines. Chaque joueur à sa propre peau.



Ce n'est quand même pas bien compliqué le mode multi-joueur. Quoique …

Que se passe-t-il si un client quitte le jeu et se reconnecte ? Essayez…

Nous verrons un peu plus loin dans ce tutorial comment régler ce problème. Pour l'instant cela nous va très bien, l'essentiel étant de différencier les joueurs.

Mais nous allons quand même tenter une petite expérience, choisir la couleur de sa peau en utilisant les touches de 1 à 4.

Il vous suffit de copier ces lignes à la fin de level11.wdl sur chaque machine et d'exécuter.

```
function change_skin1
{
```

```
		player.skin = 1;
}
function change_skin2
{
		player.skin = 2;
}
function change_skin3
{
		player.skin = 3;
}
function change_skin4
{
		player.skin = 4;
}

on_1 change_skin1;
on_2 change_skin2;
on_3 change_skin3;
on_4 change_skin4;
```

Cliquez sur 1, 2, 3 ou 4 sur le serveur et vous voyez votre joueur changer de couleur. Faites de même sur le poste client et que se passe-t-il ? Rien !!!

Voilà vous venez de toucher du doigt toute la difficulté du mode client / serveur. Les fonctions change_skin sont bien exécutées sur le poste client mais player.skin est inconnu, il est géré sur le poste serveur.

Qu'à cela ne tienne, envoyons la variable au serveur en utilisant l'instruction send (entity.skill) Nous ajoutons les lignes en rouge :

```
function change_skin1
{
		player.skin = 1;
		send (player.skin);
}
function change_skin2
{
		player.skin = 2;
		send (player.skin);
}
function change_skin3
{
		player.skin = 3;
		send (player.skin);
}
function change_skin4
{
		player.skin = 4;
		send (player.skin);
}
```

Et nous exécutons. Et ce sont bien les touches 1, 2, 3 et 4 propres à chaque joueur qui changent leur peau.

Le tir

Que serait un shooter sans tir ? Bien monotone et ennuyeux !

Nous allons donc faire tirer nos vaisseaux. Pour tirer il faut des armes et des munitions. Nous avons déjà une arme sur notre vaisseau, le canon qui se trouve à l'avant. Nous allons lui faire tirer des balles. Nous partons du principe, pour l'instant ,que les balles sont en quantité illimitée.

Nous décidons d'utiliser la touche V pour tirer. Pourquoi pas ?

Nous utiilsons pour cela le scénario level12.wdl

Voici les instructions que nous avons ajoutées :

```
on_v gun_fire;
```

Cette instruction appelle la fonction gun_fire lors de l'appui sur la touche V.

Puis nous définissons nos variables :

```
string bullet_mdl = <bullet.mdl>;// définition du modèle

var bullet[3];// coordonnées de la balle

var bullet_speed[3] 40, 0,0;// vitesse de la balle

SOUND shotgun_s = <gunfire.wav>;// son du coup de feu
```

Puis notre fonction gun_fire comme suit :

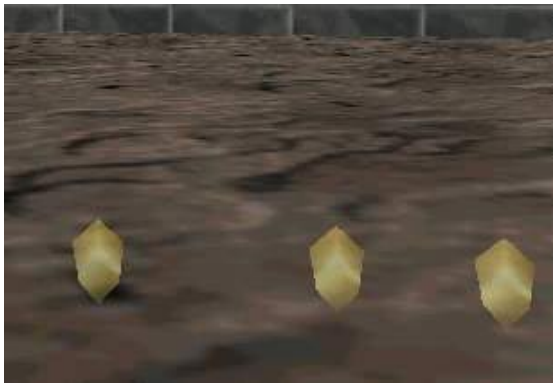
```
function gun_fire
{
        vec_set(bullet.x, player.x); // ➔ nous mettons bullet à la position de player
        bullet.x += 20;// ➔ nous avançons bullet devant le canon
        ent_create (bullet_mdl, bullet.x, gun_action);// ➔ crée la balle et exécute son action
}
```

Et enfin notre action gun_action comme suit :

```
ACTION gun_action
{
        play_entsound (player,shotgun_s,250);
        MY.ENABLE_BLOCK = ON;
        MY.ENABLE_ENTITY = ON;
        MY.ENABLE_STUCK = OFF;
        MY.PUSH = 0;
        MY.PASSABLE = OFF;
        MY.TRANSPARENT = ON;
        MY.AMBIENT = 100;
        vec_set(my.PAN, player.PAN);
        vec_set(my.x, player.x);
//      MY.event = bullet_event;
return;
        WHILE(1)
        {
                ent_MOVE (bullet_speed,nullvector) ;
                WAIT 1;
        }
}
```

Les deux valeurs en rouge sont provisoires.

Exécutons tout d'abord sur notre serveur et voyons ce qui se passe :



J'ai tiré 3 balles et les 3 balles sont bien apparues devant mon joueur sur la machine serveur.

Faisons à présent la même chose sur le poste client :

Tirons et que se passe-t-il ? Il y a bien un coup de feu d'entendu mais rien à l'écran.

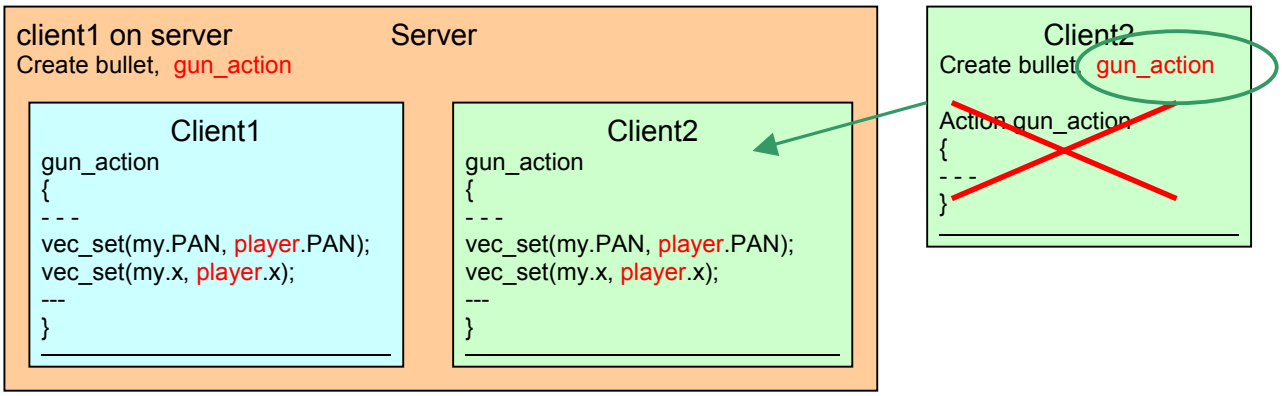
Ah si ! il y a une balle qui est apparue mais sur l'autre écran, sur le poste serveur. Aïe, aïe, aïe. Essayons de raisonner et de comprendre pourquoi.

Lorsque j'appuie sur V j'appelle la fonction gun_fire, fonction qui se trouve bien sur le poste client (je n'utilise d'ailleurs pas d'instruction send pour envoyer quoique ce soit au serveur), je peux d'ailleurs facilement le vérifier en glissant une instruction beep() au début de la fonction gun_fire.

Mais dès qu'il arrive à l'instruction ent_create, nous créons une entité, donc le serveur prend la main pour créer cette entité sur le serveur et exécuter son action (gun_action) sur le serveur. Si vous aviez déjà oublié reportez-vous à notre premier schéma.

Donc le serveur exécute 2 actions gun_action, l'une attachée à la balle créée par le serveur, l'autre attachée à la balle créée par le client.

Petit schéma récapitulatif pour ceux qui n'ont pas suivi :



Comme on peut le voir les 2 actions font référence à la même entité 'player' qui est le pointeur de l'entité créée sur le serveur, c'est donc normal que la balle s'affiche :

1 – sur le serveur dans les 2 cas
2 – se positionne par rapport au joueur du serveur.

Et oui les 'my' et les 'you' ne sont plus ce qu'ils étaient !!

La solution saute aux yeux…créons 2 gun_action, gun_action1 appelé par client1 et gun_action2 appelé par client2. Elémentaire ! Oui mais, si c'était si simple ça se saurait, il n'y aurait même pas besoin de faire un atelier.

Où est le problème me direz-vous ?

Première réflexion, nous sommes sur le net, il y a 10.000 joueurs qui se connectent, je ne vais quand même pas écrire 10.000 actions gun_action, d'autant que les joueurs ne vont pas se contenter de cette action.

Ok, vous avez raison, nous ne sommes pas sur le net, nous n'avons qu'une version commerciale, il n'y aura jamais plus de 4 joueurs, on peut bien faire une exception !
(La vérité c'est que je n'ai pas encore trouvé de solutions satisfaisantes à ce problème, que ça fait 2 mois que je m'arrache les cheveux et que je me suis dit qu'il faut quand même avancer et que l'on pouvait pour l'instant faire l'impasse sur cette contrainte d'autant qu'il y en a une autre de taille)

L'autre difficulté, qui n'est pas mieux mais que là il va falloir résoudre si on veut que cet atelier ressemble à quelque chose, c'est que notre scénario doit être **IDENTIQUE** sur chaque poste.

Je vais donc trouver sur chaque poste les instructions suivantes :

Si je suis client1 je crée une balle qui appelle gun_action1
Si je suis client2 je crée une balle qui appelle gun_action2
Si je suis client3 je crée une balle qui appelle gun_action3
Si je suis client4 je crée une balle qui appelle gun_action4

Bien oui et alors, nous ne sommes plus à 4 lignes près !

C'est vrai, autant avec le tutorial précédent, le mot d'ordre était 'optimisons, optimisons', autant avec celui-ci, je sents qu'on va se laisser aller…mais pour une fois il faut que la compréhension prime sur l'optimisation.

Non la vrai question, la seule, est :

Le schéma avec client 1, client 2, client 3, client 4 c'est bien joli mais comment le programme sait-il que je suis client1, client2, client3 ou client4 ? Vous êtes-vous déjà posé la question vous-même ? Moi oui !

J'ai tout d'abord pensé au plus logique, j'ai lu dans le manuel, (si, si, on y trouve des informations sur le mode multi-joueur !) que, dans un jeu multi-joueur, chaque entité avait un paramètre client unique et qu'une entité créée par un client héritait de ce numéro. C'est bien, je pense même que LA solution est là, mais tous mes essais sont restés infructueux. J'ai bien l'intention de reprendre mes essais après ce tutorial pour vous faire un atelier top du top, mais compte tenu des nombreuses questions posées sur le forum, j'ai pensé qu'il était urgent de donner une information qui permette à chacun d'avancer, quand bien même cette information n'est pas parfaite. Et à la lecture des nombreux mails que je reçois je sais que nous sommes nombreux à nous creuser la tête. Fermons la parenthèse.

Nous oublions donc pour l'instant le paramètre client.

La deuxième possibilité qui nous est offerte est le nom du client que vous saisissez dans votre ligne de commande (-p nom_client) ou qui est généré automatiquement par le serveur lorsque vous ne le renseignez pas. J'ai également travaillé sur cette piste. La première difficulté tient au fait que le nom est une chaîne de caractères, de longueur variable et que si le serveur connaît le

client qui lui envoie une information, il ne sait pas la renvoyer à ce même client. Les instructions send envoient à TOUS les clients en général, à chaque client de faire le tri de ce qui est pour lui. Même chose, c'est une solution qui doit fonctionner, mais que j'ai trouvé trop lourde et trop longue en développement pour encore une fois vous donner rapidement de l'information.

Il fallait que ce soit dit. Nous allons enfin pouvoir reprendre le cours de notre atelier et examiner la solution que j'ai retenue

Nous allons utiliser à présent le niveau nommé 'level2.wmb'. Vous l'ouvrez et vous lui associez le scénario level2.wdl.

Vous exécutez sur le serveur et vous exécutez sur le client.

Vous pressez la touche V sur chacun des postes et vous observez. C'est quand même mieux comme ça, vous ne trouvez pas ?

Etudions notre scénario pour comprendre ce que nous avons fait :

Tout d'abord une variable qui compte le nombre de client qui se connecte

```
var num_client = 1;
```

Nous définissons ensuite 3 pointeurs :

```
entity* player1;
entity* player2;
entity* player_for_action;
```

Pourquoi 3 pointeurs pour gérer 2 joueurs ?

Parce que malgré tout, écrire 2,3 ou 4 fois les mêmes actions de plusieurs dizaines de lignes n'est pas satisfaisant. Nous remplacerons donc player par player_for_action dans une action commune aux 4 clients, chaque action spécifique se résumant à quelques lignes.

Exemple pour notre balle (3 lignes pour chaque action 'client', 15 lignes pour l'action commune) :

```
ACTION gun1_action //appelée par le client 1
{
	if (player1)
	{
		player_for_action = player1;
		gun_action();
	}
}

ACTION gun2_action // appelée par le client 2
{
	if (player2)
	{
		player_for_action = player2;
		gun_action();
	}
}

action gun_action
{
	play_entsound (player_for_action,gun_s,250); // nous avons remplacé player par player_for_action
	MY.ENABLE_BLOCK = ON;
	MY.ENABLE_ENTITY = ON;
	MY.ENABLE_STUCK = OFF;
```

```
		MY.PUSH = 0;
		MY.PASSABLE = OFF;
		MY.TRANSPARENT = ON;
		MY.AMBIENT = 100;
		vec_set(my.PAN, player_for_action.PAN); // nous avons remplacé player par player_for_action
		vec_set(my.x, player_for_action.x); // nous avons remplacé player par player_for_action

		//MY.event = gun_event;
return;
		WHILE(1)
		{
			ent_MOVE (bullet_speed,nullvector) ;
			WAIT 1;
		}
```

Ensuite l'instruction d'appel pour la création du joueur ne gère plus son emplacement (nullvector).

```
player = ent_create(player_mdl,nullvector,player_client);
```

Par contre tout se fait donc sur le serveur, lors de la création de l'entité.

Nous allons utiliser le numéro de client, qui est incrémenté de 1 à chaque création d'une entité joueur. Chaque peau qui sera donc unique pour chacun des joueurs servira pour définir la position du joueur au départ et le pointeur correspondant. (player1, player2 etc)

```
my.skin = num_client;

if (my.skin==1){my.X = 4640;my.Y = -2568;my.z = 20;}
if (my.skin==2){my.X = -4640;my.Y = 2568;my.z = 20;}

if (my.skin ==1){player1 = my;while (player1 == null){wait(1);}}
if (my.skin ==2){ player2 = my;while (player2 == null){wait(1);}}
```

Nous ne gérons pas pour l'instant les déconnexion / reconnection

```
num_client += 1;
if (num_client > 3){wait(1);remove (me);return;}
```

Nous devons en permanence envoyer au poste client des variables d'entité, nous le faisons par cette instruction :

```
send_variable();
```

Nous retrouvons donc cette fonction qui pour l'instant n'envoie que la variable skin mais qui servira pour d'autres variables un peu plus tard :

```
function send_variable
{
	while(1)
	{
		if (player2)
		{
			send (player2.skin);
		}
		if (player1)
		{
			send (player1.skin);
		}
		wait(1);
	}
}
```

Notre fonction gun_fire, qui je vous le rappelle est exécutée sur chaque poste client s'écrit à présent comme ceci :

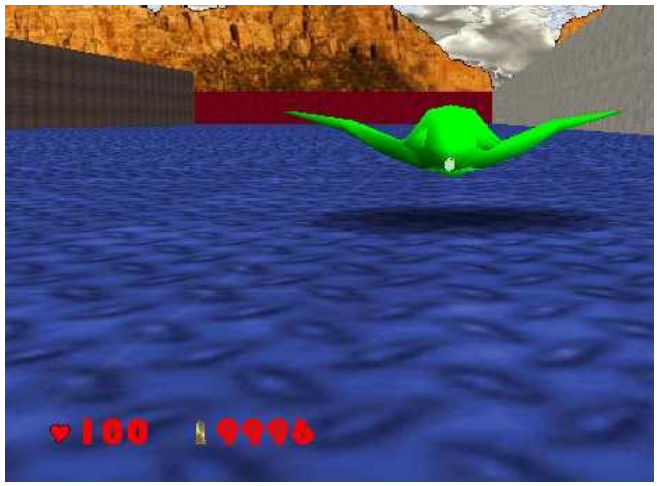
```
function gun_fire
{
	if (player)
	{
		vec_set(bullet.x,player.x);
		bullet.x +=20;
		if (player.skin == 1){ent_create (bullet_mdl, bullet.x,gun1_action);}
		if (player.skin == 2){ent_create (bullet_mdl, bullet.x,gun2_action);}
	}
}
```

Chaque client étant identifié par sa variable skin, appellera bien le gun_action qui lui appartient.

Bien, nous allons donc pouvoir à présent laisser filer la balle et gérer ses évènements. Vous assignez le scénario level21.wdl et vous exécutez. Cette gestion étant classique et standard, nous ne la commenterons pas.

L'affichage d'un panneau

Bien, nous souhaitons à présent gérer le nombre de balles, les points de vie des joueurs et afficher ces valeurs à l'écran. Nous assignons le scénario level22.wdl. Nous exécutons et vous devriez obtenir ceci :



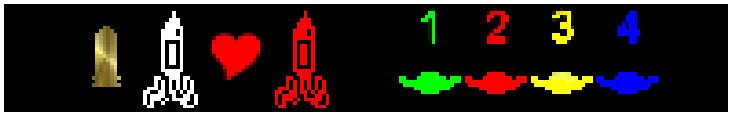
Je sais, j'ai fait fort avec le nombre de balles, mais je n'en ai pas de trop pour pouvoir gagner ! (on ne peut pas être bon en tout).

Voyons à présent le code.

Nous définissons tout d'abord 2 variables d'ENTITE, health pour les points de vie et ammo pour le nombre de balles.

```
define health,skill48;
define ammo,skill47;
```

Puis nous définissons notre panneau ainsi que nos polices.



icône.bmp

```
//panel
FONT panel_font, <bauhau20.bmp>, 18, 30;
FONT icone_font, <icone20.bmp>, 18, 30;
PANEL display_panel {
	LAYER		3;
	POS_X		0;
	DIGITS	30,0,1,icone_font,1,0;
	DIGITS	50,0,3,panel_font,1,player.health;
	DIGITS	130,0,1,icone_font,1,1;
	DIGITS	150,0,4,panel_font,1,player.ammo;

	FLAGS	= REFRESH,D3D;
}
```

Les panneaux étant propre à chaque client, c'est bien les variables du player de chaque client qui seront affichées.

Nous initialisons d'ailleurs ces variables lors de la création des joueurs, donc sur le serveur par ces lignes (dans la fonction 'player_client') :

```
my.health = 100;
my.ammo = 9999;
```

Et nous n'oublions surtout pas de les renvoyer à chaque client. Nous complétons donc notre fonction send_variable par les lignes en rouge :

```
function send_variable
{
	while(1)
	{
		if (player2)
		{
			send (player2.skin);
			send (player2.health);
			send (player2.ammo);
		}
		if (player1)
		{
			send (player1.skin);
			send (player1.health);
			send (player1.ammo);
		}
		wait(1);
	}
}
```

Nous ajoutons le décompte des points de vie dans la fonction gun_event ➔ event_entity :

```
your.health -= 5;
```

Et le décompte des balles dans la fonction gun_action :

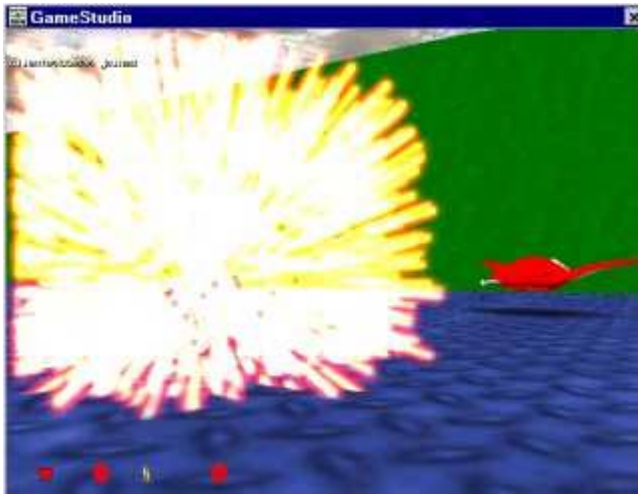
```
player_for_action.ammo -=1;
```

Gérons à présent notre mort, utilisons pour cela le scénario level23.wdl.

Nous définissons tout d'abord notre son d'explosion :

```
SOUND explo = <explosin.wav>;
```

Puis je n'ai pas hésité une seconde, j'ai pris l'exemple de la fonction effect_explo qui illustre la nouvelle instruction 'effect' dans le manuel WDL (je vous laisse donc vous y reporter). Vous verrez, elle est tout simplement magnifique :



Le test des points de vie se fait dans notre fonction send_variable comme suit (exemple pour un joueur) :

```
if (player1)
{
	send (player1.skin);
	send (player1.health);
	send (player1.ammo);
	if (player1.health <= 0)
	{
		player1.health = 0;
		play_entsound (player1,explo,2000);
		effect(effect_explo,2000,player1.x,normal);
		WAIT 1;
		remove player1;
		goto suite;
	}
}
```

Améliorons l'armement

Nous pourrions arrêter notre atelier ici, nous avons déjà matière à développer des jeux en réseau.

Mais il m'a semblé intéressant d'aller un peu plus loin dans notre armement. En effet le propre d'un shooter est de pouvoir changer d'arme. (le prochain tutorial sur le mode multi-joueur traitera d'un vrai shooter en jeu 1ère personne, avec possibilité de prendre des armes, des munitions et de se soigner, en bref de retrouver les templates actuels)

Pour l'instant nous allons doter notre vaisseau de roquettes, possibilité d'en avoir une sous chaque aile et de pouvoir la tirer avec réarmement à la demande. Nous allons tout d'abord nous intéresser à la roquette sous l'aile gauche, la droite n'étant qu'une simple copie.

Nous utilisons à présent le scénario level24.wdl.

Nous exécutons notre niveau, nous chargeons ou rechargeons la roquette gauche à l'aide de la touche 1 et nous tirons avec la barre espace.

Ce que nous avons fait :

Définition des variables :

```
string missile_l_mdl = <missle_l.mdl>;


var rocket_pos[3];
var rock_pan[3];
var v_smoke;
var rocket_speed[3] 10, 0,0;

define rocket_left, skill45;

SOUND explo_miss_s= <explo_miss.wav>;
SOUND launch = <missle.wav>;
```

Ensuite la fonction rocketl_fire qui ressemble comme 2 gouttes d'eau à la fonction gun_fire et qui est, je vous le rappelle, exécutée sur chaque poste d'ou l'utilisation de player :

```
function rocketl_fire //cette fonction est appellée sur chaque poste
//l'action qui en découle est toujours exécutée sur le serveur
{
	if (player)
	{
		if (player.skin == 1){ent_create (missile_l_mdl, player.x,rocket1l_action);}
		if (player.skin == 2){ent_create (missile_l_mdl, player.x,rocket2l_action);}
	}
}
```

Puis nos 2 fonctions clavier (touche 1 pour charger la roquette, touche espace pour tirer)

```
on_1 prepare_rocket_gauche;
on_space tire_rocket;
```

Puis les 2 fonctions qui en découlent, qui sont exécutées sur chaque poste client alors que l'action des roquettes est exécutée sur le serveur, il est donc nécessaire d'envoyer l'information au serveur. Nous utilisons la variable d'entité skill45, que nous avons redéfinie comme rocket_left est qui est mise à 1 pour charger la roquette et à 0 pour indiquer qu'on la tire.

```
function prepare_rocket_gauche//recharge une roquette à gauche
{
	if (player)
	{
		if (player.rocket_left == 1){return;}//on ne peut pas charger 2 roquettes à la fois
		player.rocket_left = 1;
		send (player.rocket_left);
		rocketl_fire();
	}
}

function tire_rocket//on tire une roquette
{
	if (player)
	{
		player.rocket_left = 0;
		send (player.rocket_left);
	}
}
```

Nos rocket_action, dont la 1 est donnée en exemple se différencie par contre de gun_action.Nous avons en effet 3 étapes avec la roquette.

- Etape 1, la roquette est chargée (rocket_left = 1), nous l'avons créée, il nous faut l'attacher au joueur pour qu'elle le suive partout, c'est le rôle de la boucle while.
- Etape 2, la roquette est tirée (rocket_left = 0), on appelle rocket_action.
- Etape 3 : la roquette se déplace seule dans rocket_action, on gère les évènements et son explosion.

```
ACTION rocket1l_action
{
	MY.ENABLE_DISCONNECT = ON;
	MY.EVENT = _actor_connect;
	my.passable = on;

	while(my !=null && player1.rocket_left ==1)//la roquette suit le vaisseau
	{
		vec_set(my.x,player1.x);
		vec_set(my.PAN, player1.pan);
		wait(1);
	}
//on exécute rocket_action uniquement si on tire
	if (player1.rocket_left == 0){player_for_action = player1;rocket_action();}
}
```

On modifie enfin notre test de mort car les roquettes doivent être détruites lorsque l'on meurt. J'ai pris l'option de mettre la variable rocket_left à 0 ce qui fait que lorsque vous détruisez un vaisseau ennemi, il a le temps de tirer ses roquettes. Attention si vous êtes en face ! (sur le niveau end_level les roquettes sont détruites en même temps que le vaisseau)

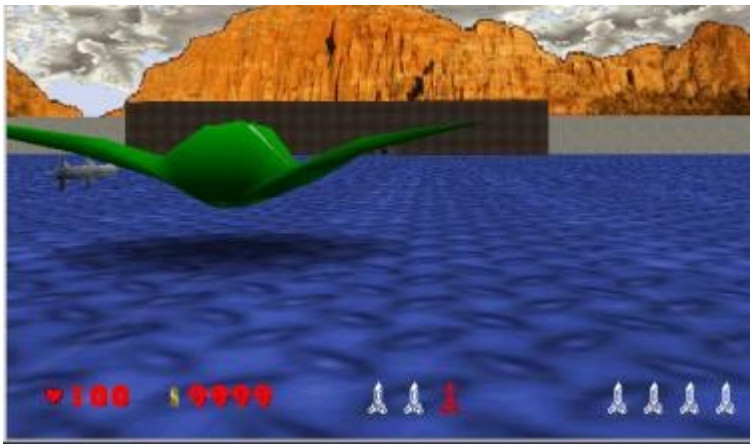
```
function send_variable
{
	while(1)
	{
		if (player1)
		{
			send (player1.skin);
			send (player1.health);
			send (player1.ammo);
			if (player1.health <= 0)
			{
				player1.health = 0;
				player1.rocket_left = 0;
				send_vec (player1.rocket_left);

				play_entsound (player1,explo,2000);
				effect(effect_explo,2000,player1.x,normal);
				WAIT 1;
				remove player1;
				goto suite;
			}

		}
```

Il ne nous reste plus qu'à faire la même chose avec la roquette droite. Elle se charge avec la touche 2, et pour le tir j'utilise la touche CTRL.

Le scénario à utiliser est level25.wdl. J'en ai profité pour gérer le nombre de roquettes.

Pour changer de nos balles qui sont illimitées, j'ai limité à 4 roquettes à gauche et à 4 roquettes à droite

Et pour l'affichage, plutôt que de réafficher des chiffres, je me suis fait plaisir en affichant directement les roquettes. (en rouge sous votre aile, prête à être tirée, en blanc prête à être chargée.)



J'utilise pour cela 2 tableaux, la première valeur étant le nombre de roquettes restantes, les 4 valeurs suivantes servant uniquement pour l'affichage des petites roquettes à l'écran. (2 affiche roquette blanche, 4 affiche roquette en rouge, 0 affiche rien)

```
var rocket_l[5] = 4,2,2,2,2;
var rocket_r[5] = 4,2,2,2,2;
```

Bien entendu, si nous voulons que ces valeurs s'affichent sur chaque poste nous devons les associer aux variables de l'entité. C'est fait avec ces instructions (skill37 à skill44) :

```
vec_set(player.skill37,rocket_l[1]);//transfert 3 variables d'un coup
player.skill40 = rocket_l[4];
```

Il reste à les envoyer au serveur avec ces instructions :

```
function send_var_vec
{
	send_vec (player.skill37);
	send_vec (player.skill40);
	send_vec (player.skill43);
	send_vec (player.skill46);
}
```

J'utilise les instructions send_vec qui envoie 3 variables à la fois, j'ai donc bien entendu groupé toutes mes variables (skill37 à skill48).

Conclusion :

Ce tutorial est presque terminé. (vous pouvez réutiliser le scénario end_level.wdl).

J'ai ajouté une gestion des vaisseaux en générant une fumée dont la densité est inversement proportionnelle aux point de vie et en réglant la vitesse de déplacement sur les mêmes points de vie.

```
v_smoke = (100-player1.health)/10;
player1._force = 0.5+(player1.health/100);
send (player1._force);
```

```
emit v_smoke,player1.pos,particle_smoke;
```

Il reste à dupliquer player1 et player2 pour créer player3 et player4. L'inconvénient du copier / coller fait que l'on oublie toujours de changer une ou plusieurs valeurs et généralement le jeu est boiteux. Pensez donc à utiliser la fonction recherche / remplace de votre traitement de texte favori.

Lors d'un ultime test j'ai rencontré un autre problème :
Lorsqu'un vaisseau meurt j'ai constaté que l'instruction 'remove playerx' faisait bien disparaître l'action du player x sur le serveur mais le pointeur player continue d'exister sur le poste client. Ce qui fait que bien qu'il n'y ait plus de vaisseau on pouvait continuer à appeler les roquettes s'il en restait. Le problème étant que l'entité de ce joueur n'existant plus pour le serveur il n'est plus possible d'envoyer des informations au client.

Après de nombreux essais, j'ai en fait utilisé la dernier variable d'entité envoyée avant la destruction de l'entité à savoir player.health qui a la valeur 0 à ce moment.

Voici la routine :

```
function wait_end
{
	while(1)
	{
		if (player)
		{
			if (player.health ==0) //on est mort
			{
				rocket_r =0;
				rocket_l = 0;
				send_rocket_empty_l();
				send_rocket_empty_r();
				wait(1);
				remove(player);
			}
		}
		wait (1);
	}

}
```

C'est une fonction qui s'exécute sur les postes clients,d'où la référence à player, qui remet à 0 les compteurs de roquettes, qui efface l'affichage des roquettes et qui supprime également l'entité sur le poste client.

Cette fonction est appelée depuis la fonction Main.

Prochain volet de ce long mais passionnant atelier sur les jeux multi-joueurs :

Normalement il devrait y avoir un dernier tutorial (il est déjà bien avancé pour tout vous dire). Ce tutorial fonctionne sur le principe d'un jeu ouvert, c'est à dire que nous ne sommes pas obligé d'attendre que quelqu'un soit connecté pour jouer.

Il y a des jeux ou il est important de démarrer une fois que tous les joueurs sont présents.

Ce dernier tutorial devrait donc présenter un écran d'accueil, avec possibilité de voir la machine serveur, d'afficher les joueurs connectés pour les accepter ou les refuser et chaque client devrait pouvoir choisir son camp de départ (type de joueur, couleur etc.)

Cerise sur le gâteau

Cela fait quelques jours que je joue avec ma famille et mes amis et il y a un reproche qui revient souvent, on ne sait pas qui est dans le jeu, qui est mort.

J'ai donc ajouté dans le bas de l'écran des petits vaisseaux qui matérialisent les joueurs présents.

ici, 1,3 et 4 sont dans le jeu. 2 est mort.

Voici à présent le code : (le scénario à utiliser est 'end_level_with_player_connected.wdl')

Nos variables :

```
var player_connected; //binary 1,2,4,8 for player 1,2,3,4
var player_connected1;
var player_connected2;
var player_connected3;
var player_connected4;
```

Notre panneau d'affichage :

```
DIGITS  5,0,1,icone_font,1,player_connected1;
DIGITS  25,0,1,icone_font,1,player_connected2;
DIGITS  45,0,1,icone_font,1,player_connected3;
DIGITS  65,0,1,icone_font,1,player_connected4;
```

C'est dans la fonction wait_end que nous mettons à jour les 4 variables :

```
function wait_end
{
	while(1)
	{
		player_connected1 = 5+(player_connected & 1);//6
		player_connected2 = 5+(player_connected & 2);//7
		player_connected3 = 5+(3*((player_connected & 4)==4));//8
		player_connected4 = 5+(4*((player_connected & 8)==8));//9


		if (player)
		{
```

La variable player_connected est mise à jour à chaque connexion lors de la création du joueur :

```
function player_client()
{
	my.skin = num_client;

	if (my.skin==1){my.X = 1000;my.Y = -1000;my.z = 20;}
----------

	if (my.skin ==1){player_connected |= 1;player1 = my;while (player1 == null){wait(1);}}
	if (my.skin ==2){ player_connected |= 2;player2 = my;while (player2 == null){wait(1);}}
	if (my.skin ==3){player_connected |= 4;player3 = my;while (player3 == null){wait(1);}}
	if (my.skin ==4){player_connected |= 8;player4 = my;while (player4 == null){wait(1);}}
```

```
---------

        player_drive();
        send_var (player_connected);
        send_variable();
        if (MY.shadow == OFF) { drop_shadow(); }
```

Et la variable player_connected est mise à jour à chaque fois qu'un vaisseau meurt :

```
if (player1.health <= 0)
{
        ------
        player_connected &=254;
        send_var (player_connected);
        ---------
if (player2.health <= 0)
{
        ------
        player_connected &=253;
        send_var (player_connected);
        ---------
if (player3.health <= 0)
{
        ------
        player_connected &=251;
        send_var (player_connected);
        ---------
if (player4.health <= 0)
{
        ------
        player_connected &=247;
        send_var (player_connected);
        ---------
```

Si vous êtes perspicace, vous aurez noté que je ne gère pas les déconnexions volontaires. La solution au prochain numéro. On peut régler le problème provisoirement en mettant

```
my.enable_disconnect = off; //à la place de on
```

dans la fonction player_client. Ce qui vous permettra d'aller détruire un pauvre vaisseau sans défenses.