# MMSBYOD

## Jeu Massivement MultiSmartphone, Bring Your Own Device

BISSON Bastien

BONNAFOUX Florian

DUFOUR Mathieu

JALLY Jordan

THUAIRE Clement

IUT de Clermont-Ferrand
Département Informatique – 2013/2014

# 1- Autorisation de publication

Nous autorisons la diffusion de ce rapport sur l'intranet de l'Institut Universitaire et Technologique de l'Université d'Auvergne.

# 2 - Remerciements

Nous tenons à remercier nos deux tuteurs Mr. Philippe Kauffmann et Mr. Jacques Laffont pour toute l'aide qu'ils nous ont apportée mais aussi pour les différents conseils qu'ils ont pu nous donner.

# 3 - Sommaire

# 4 – Introduction

## 4.1. Présentation du sujet

Notre sujet nous a été proposé par Mrs. Laffont et Kauffmann en novembre 2013. Le projet consiste à créer un jeu d'action en tour par tour.

Le but de ce sujet a été de développer un jeu à partir du moteur Raydium (moteur graphique open source). Ce jeu serait un champ de bataille où des unités (pilotées par smartphone) s'affronteraient.

Nous avions comme contrainte que le plus grand nombre de joueurs devait pouvoir se connecter en parallèle en utilisant différentes évolutions de clients. Que ce soit de la page web minimaliste au client Android plus « lourd ».

C'est pourquoi, dans ce rapport, nous vous présenterons les différentes tâches que nous avons chacun effectuées pour permettre de réaliser ce projet en commençant tout d'abord par le serveur Raydium, puis par la connexion des clients « léger », enfin nous terminerons par la connexion du client « lourd » (Android) et par le level-design.

## 4.2. Présentation de l'équipe

Pour réaliser ce projet, nous étions une équipe de 5 personnes, toutes issues de la filière SI du DUT informatique de l'Université d'Auvergne. Cette équipe était composée de :

- BISSON Bastien
- BONNAFOUX Florian
- DUFOUR Mathieu
- JALLY Jordan
- THUAIRE Clément

Malheureusement, Florian BONNAFOUX a dû nous quitter au bout de 7 semaines de projet, car il n'avait pas validé son second semestre.

## 4.3. Présentation des tâches et de la répartition

Le projet a vu dès le début apparaitre plusieurs grandes thématiques qui se distinguaient : le développement du moteur de jeu, le design, la connexion avec un client léger (web) et enfin la connexion et le jeu avec un client lourd (Android).

Pour pouvoir mener à bien ce projet, nous avons décidé avec nos tuteurs de diviser notre temps global en trois sous-versions pour arriver à la version 1 en mars, date de fin de projet. Ensuite nous avons attribué à chacun de nous une tâche en fonction de nos envies en essayant de les découper elles-mêmes. Ce qui nous a permis de réaliser un diagramme de Gantt final *(Annexe 1)*.

# 5 – Développement

## 5.1. Présentation des outils

### 5.1.1. Raydium

Raydium est un moteur graphique de jeu initié début 2001. Il permet de développer des jeux 3D sans avoir d'expérience dans le domaine du développement de jeu. Ce moteur graphique n'est pas le plus optimisé pour créer des jeux d'un niveau graphique optimal, mais il permet de développer des versions de très bonne qualité tout de même, sans être un véritable développeur de jeux vidéo. Il permet la gestion des entrées d'un joueur (clavier, souris, joystick, joypad, retour de force), l'affichage, le son…

Il est un aussi un moteur physique, il est possible de gérer par exemple la gravité, le poids des objets, etc… sans oublier que ce moteur permet le scripting PHP. Des tests complexes mais aussi des jeux complets ont déjà été créés avec Raydium comme ManiaDrive un jeu de course de voitures, NewSkyDriver et Isaac. Ce moteur est un logiciel libre, sous licence GPL. Pour conclure sur la présentation de cet outil, Raydium se présente comme un moteur qui se veut facile permettant l'écriture d'un jeu 3D souple et rapide.

### 5.1.2. Blender

Blender est un logiciel de modélisation, d'animation et de rendu en 3D. Il dispose de fonctions avancées de modélisation, de sculpture 3D, de dépliage UV[1], de texturage[2], d'animation 3D, et de rendu. Il gère également le montage vidéo non linéaire, la création d'applications en 3D interactives, ainsi que diverses simulations physiques telles que les particules, les corps rigides, les corps souples et les fluides. Blender est donc un logiciel complet pour réaliser des images et des animations de synthèse. Il regroupe, au sein d'une interface stable et moderne, de puissants outils de modélisation, d'animation et de rendu d'une grande qualité.

### 5.1.3. Android

Android est un système d'exploitation principalement développé pour smartphones et tablettes tactiles. Il a été mis au point en 2005 par la multinationale Google. Depuis, Google a mis au point onze mises à jour à destination des tous les utilisateurs d'Android (plus d'un milliard). Le développement d'applications pour cette plate-forme utilise le langage Java et XML pour le développement des différentes vues affichées pour l'utilisateur. Le téléchargement de ces applications se fait via le Play Store où les développeurs peuvent mettre leurs applications en téléchargement contre une inscription payante.

[1] Dépliage UV : Déplier un objet polygonal dans un plan 2d via un système de coordonnées.
[2] Texturage : Fait de texturer un objet, c'est à dire lui appliquer une texture.

### 5.1.4. Eclipse

Le logiciel Eclipse est un IDE[3]. Ce logiciel est spécialisé dans le développement de logiciels et d'applications utilisant le langage JAVA. Certaines versions ont aussi été adaptées spécialement pour le développement d'applications Android. C'est le cas du logiciel *Android Developer Tools*. Les IDE (comme Eclipse) sont un ensemble d'outils servant à faciliter le travail du développeur en mettant à sa disposition un grand nombre de fonctionnalités (debogger, éditeur de textes, raccourcis,…).

### 5.1.5. CodeBlocks

CodeBlocks est un IDE libre et multiplateformes (linux, Windows, mac) dont la 1ere version stable a vu le jour le 28 février 2008. Cet IDE est surtout orienté C/C++ mais il supporte aussi d'autres langages comme le D. Les numéros de version s'apparentent beaucoup à Ubuntu, la dernière version stable est donc la 12.11 cependant il existe aussi des versions non officielles.

## 5.2. Raydium

### 5.2.1. Prise en main des fonctions raydium

Au niveau du moteur de jeu, l'étude de Raydium a été importante dans le but d'utiliser une partie des capacités de ce moteur. Suite à l'installation de Raydium, le lancement de diverses applications de base proposées par ce dernier a été fait afin de voir les possibilités du moteur. Il se présente comme un ensemble de fonctions simples en C qui permettent de rendre l'écriture d'un jeu 3D souple et rapide. Une description de l'API[4] claire est présente sur le site officiel du moteur de jeu (raydium.org) mais aussi quelques tutoriaux basiques afin de créer une simple application, de déplacer des objets mais aussi d'utiliser la physique, sont fournis.

Le premier tutoriel a permis de savoir comment créer une fenêtre, la dimensionner, influer sur la distance de la caméra, régler la lumière principale qui est codée sous la forme d'un tableau de flottant RGBA (RED, GREEN, BLUE, ALPHA), il s'agit d'un tableau ou chaque valeur est comprise entre 0 et 1, pour notre projet, nous avons fixé le tableau de la manière suivante :

```
GLfloat light_color[] = {1.0, 0.9, 0.8, 1.0};    //definition de la couleur de la lumière
```

Figure 1 – Tableau fixant l'éclairage

[3] Integrated Development Environment (Environnement de développement intégré)
[4] API : Application Programming Interface :ensemble des fonctions.

Ce qui donne une scène bien éclairée et donc agréable à jouer. Tout ce qui est affichage se situe dans la fonction display qui est appelée dans le *main* grâce à *raydium_callback(&display)*

Ce tutoriel nous apprend aussi à traduire les instructions envoyées par un utilisateur, ici il s'agissait de la distance de la caméra. La caméra cible un point trois variables camx, camy et camz qui représentent respectivement les coordonnées x, y et z du point ciblé dans la scène. Pour ensuite prendre en compte les entrées utilisateur, cela est géré grâce à desf tests : par exemple si l'utilisateur appuie sur la flèche du haut, camz est incrémenté, ce qui nous permet de zoomer. Ci-dessous, les tests permettant un zoom et un dé-zoom de la caméra.

```
camx+=raydium_joy_x;
camy+=raydium_joy_y;

if(raydium_key[GLUT_KEY_PAGE_DOWN]) camz--;
if(raydium_key[GLUT_KEY_PAGE_UP]  ) camz++;
```

Figure 2 – Test zoom et dé-zoom

Ce code se situe dans la fonction display.

Le second tutoriel présente comment déplacer des objets de façon très simple, sans utiliser la physique, il se base logiquement sur le premier tutoriel et montre aussi comment utiliser une caméra dynamique (qui suit un objet). Le troisième tutoriel étudié permet de concevoir une application avec une voiture qui peut se déplacer, en utilisant des moteurs physiques.

### 5.2.2. Fusion des 2 tests

Ensuite une analyse de certains codes sources a permis une compréhension des fonctions que présente Raydium comme le test6 et le test8 qui sont respectivement un personnage qui tire des missiles et un tank qui se déplace avec plusieurs angles de vue possibles. La combinaison de ces deux tests ressemble particulièrement au sujet du projet.

C'est ce qui a été fait dans un premier temps prenant plusieurs semaines de réalisation et permettant ainsi de prendre en main le moteur de jeu mais aussi de développer, d'avancer le projet. La mise en place du missile sur le tank a été assez compliquée au départ puisque il a fallu récupérer le code du missile du test 6 pour l'intégrer dans le code du test8.

De plus, cette version a été ensuite améliorée, un temps de rechargement entre chaque tir a été mis en place, mais aussi un déplacement vertical du canon et un déplacement horizontal de la tour du tank ont été rajoutés.


Figure 3 – Captures d'écran des applications test6 et test8

Ces deux derniers déplacements sont basés sur des moteurs angulaires. Pour en revenir au temps de rechargement mis en place, il s'agit d'une fonction nommée *step_frame* qui décrémente le temps de rechargement de chaque tank toutes les secondes, cette fonction est appelée dans la fonction *main* grâce à la fonction :

*raydium_timecall_add(void* funct ,GLint Hz);*

Cette fonction prend en paramètres la fonction à appeler et la fréquence de l'appel à cette dernière. Ci-dessous la fonction *step_frame* :

```
void frame_step(GLfloat step)
{
    int i=0;
      for(; i < MAX_PLAYERS; ++i){
    player[i].reload-=step;
    if(player[i].dead >0)
        player[i].dead -=step;
    }

}
```

Figure 4 – Fonction frame_step

Ce temps de rechargement est important car il présente deux propriétés intéressantes : en effet, il permet d'établir un certain mode de jeu, le jeu devient plus stratégique, basé sur la rapidité du joueur à viser un tank adverse. Dans un second temps, couplé à la suppression d'un missile lors d'une collision, ce temps de rechargement permet de brider les objets à traiter qui pourraient rapidement devenir ingérables par le processeur et faire freezer[5] le jeu.

[5] freezer = chute d'images par seconde jusqu'à arriver à une image par seconde, injouable par l'utilisateur

### 5.2.3. Mise en place d'une structure (multi-joueurs)

Au niveau de la programmation, le moteur de jeu assimile les objets comme un assemblage d'éléments, concrètement un tank étant un objet constitué d'une base, d'un canon, d'une tour et de chenilles composées de 6 roues de chaque côté. Il possède aussi un missile lors du déclenchement du tir.
Chaque élément possède un tag, c'est-à-dire un numéro qui les désigne et qui va être important pour la fonction de collision. Une fonction *simul_create* permet la construction physique d'un tank que l'on peut qualifier d'objet, par un assemblage d'éléments. C'est dans cette fonction que les tags sont donnés à chaque élément, exceptés les missiles qui ne sont pas encore assemblés avec le tank et qui ne le sont qu'en cas de tir.

```
sprintf(temp_str,"TANK-%03d",num);

player[num].tank_object=raydium_ode_object_create(temp_str);
sprintf(temp_str,"TANK_BODY-%03d",num);
player[num].tank_body=raydium_ode_object_box_add(temp_str,player[num].tank_object,
10,RAYDIUM_ODE_AUTODETECT,0,0,RAYDIUM_ODE_STANDARD,num+Tank_corps,"t80b.tri");
```

Figure 5 – Fonction *simul_create*

On passe en paramètre à la fonction *simul_create* un entier 'num' qui correspond au numéro du tank.

Afin de pouvoir développer un jeu multi-joueurs une modification importante du design du code a dû être réalisée : il s'agissait de réadapter toute les fonctions issues de la fusion des test6 et test8 et les fonctions ajoutées comme le déplacement du canon verticalement et horizontalement. Pour cela, une structure nommée 'player' a été ajoutée. Cette structure possède toutes les informations relatives à un joueur :
- Son nom
- Son identifiant
- S'il est actif
- Le tank contrôlé ainsi que tous les éléments relatifs au tank
- Son temps de rechargement
- Sa vie mais aussi son équipe

Une fonction d'initialisation de ces informations a donc été mise en place et est appelée pour chaque joueur dans le *main*. (annexe 2)

Une partie essentielle du jeu a été la gestion des collisions. En particulier, la collision d'un missile sur un tank, puisque Raydium gère « automatiquement » les collisions des objets avec le décor. Cette collision a été la partie la plus complexe à réaliser malgré le fait qu'elle soit déjà implémentée dans le test 6 mais pour un unique tank. C'est ici qu'intervient la notion de tag expliquée quelques lignes auparavant.
Mr Laffont m'a proposé dans un premier temps une première technique qui se basait sur le fait que chaque élément créé possédait un tag sur 4 octets (un int) décomposé en 2 parties. La partie haute représentant le type d'éléments, par exemple la tour du tank, le canon, etc… et la partie basse présentant le numéro correspondant au tank.

Cependant, cette technique présentant des opérations m'a paru dans un premier temps assez compliquée, je me suis donc reporté sur une autre technique qui était d'assigner à chaque élément un numéro unique. J'ai donc implémenté cette fonction et la création des éléments avec mes propres tags que j'ai assignés de manière statique.

En commentaire de la fonction de collision, nous avons établi les tags des objets afin qu'en éventuel cas de reprise du code, ils soient compris.

```
#define Tank_corps    64
#define Tank_tour    128
#define Tank_canon   256
#define Tank_sphere  512
#define Missile     1024
```

Figure 6 – Définition des tags objets

L'implémentation de cette fonction n'est pas volumineuse, elle représente une quinzaine de lignes mais est basée sur des tests. Grâce aux éléments passés en arguments (e1 et e2) on va récupérer leurs tags et donc connaître leur nature (partie du tank ou un missile). Cette fonction (*collide*) permet de modifier la vie d'un tank, et de faire disparaître le missile lorsqu'il percute un tank.

### 5.2.4. Mise en place des points

Dans la dernière partie du projet, un système de points a été instauré pour donner un véritable intérêt au jeu. Au niveau du code, nous avons déclaré et initialisé à zéro, deux variables globales nommées point1 et point2 qui représentent les points des deux équipes. Lorsqu'un tank subit des dégâts et que son niveau de vie devient nul, 100 points sont affectés à l'équipe adverse. Le tank ne peut plus tirer de missile pendant une période de 10 secondes puis sa vie est remise à 100.

Ce traitement est aussi effectué dans la fonction de collision. Les points sont cumulés et lorsqu'une des deux équipes passe le seuil des 2000 points, ce qui correspond à 20 tanks «détruits», le jeu se termine en affichant un message qui indique le nom de l'équipe gagnante. La suppression d'un tank lors de sa « destruction » n'a pas été mise en place, faute de temps.

Il a aussi fallu mettre en place l'apparition sur certains points de la scène (sur les plateaux) grâce à la fonction *raydium_pos_rot_from_file* proposée par l'api raydium qui récupère les points dans un fichier et qui permet de traiter la position, en l'occurrence ici de placer le tank par la suite dans la fonction qui ajoute un joueur nommé *player_add_num*. Ce fichier et cette mise en place des points seront expliqués plus en détails dans la partie modélisation.

## 5.3. Connexion avec client externe

### 5.3.1. Serveur web

Nous voulions que la connexion à notre jeu puisse se faire pour tout le monde à partir d'un smartphone, pas forcément sous Android, mais aussi sous d'autres systèmes d'exploitations. Nous avons donc été dans l'obligation de mettre en place une connexion à partir d'une page web.

Le serveur web initialise tout d'abord des extensions de fichiers (.jpg, .html, etc...) grâce à la fonction *raydium_web_extension_add*. Ce qui nous permet de pouvoir utiliser dans le serveur web divers fichiers utiles à son exploitation maximale. Une extension importante rajoutée est le 'htmlp', qui correspond à des pages web paramétriques à arguments variables pouvant être remplacées dans le serveur web par des valeurs de jeu.

Nous avons donc dû mettre en place un système pouvant récupérer ces arguments, les modifier de façon interne, et les renvoyer sur nos pages web afin qu'ils puissent être affichés.

Cette récupération est réalisée grâce fonction *parse_requete* (annexe 3) quiregarde si l'url contient des paramètres. Si c'est le cas, tant que le nombre de paramètres est inférieur à 32, la fonction récupére le paramètre et sa valeur qu'elle va aussi stocker puis incrémenter le nombre de paramètres.

Ensuite pour les modifier, nous avons implémenté une fonction *str_replace* (annexe 4). Les valeurs dans notre url sont entourées de {{ et }} afin de pouvoir mieux les repérer.

Pour cela, l'utilisation de cette requête de modification se fait ainsi :

```
str_replace(response, "{{Joueur}}", "Toto");
```

Figure 7 – Requête de modification str_replace

Pour finir, l'envoi sur la page web se fait via un callback fait sur notre extension htlmp.

A chaque fois qu'il y a une nouvelle requête, une fonction de callback est appelée, et dans cette fonction on va faire des *parse_requete*, et des *str_replace*.

Notre serveur raydium va donc nous permettre une connexion à partir d'une page web, de récupérer les différents paramètres et valeurs d'un joueur, et afficher sur nos pages webs les différentes valeurs essentielles au bon déroulement de la partie (point de vie d'un joueur, point de chaque équipe, nombres de joueurs connectés, position d'un joueur, etc....).

### 5.3.2. Les pages webs

Nos pages webs vont pratiquement toutes avoir la même structure afin de pouvoir garder une cohérence au niveau du code, mais aussi de la charte graphique.

Chaque page possède donc trois grandes parties : une partie dite « fond blanc » qui comportera soit une présentation succincte du projet pour la page d'accueil (annexe 5), soit des formulaires pour interagir avec le joueur pour l'initialisation et la page de jeu (annexe 6 et 7) ; une partie *header* qui contiendra dans notre cas l'image avec le tank. Attention, elle est différente de l'en-tête d'une page web (ce qui se trouve entre <head> et </head> au niveau du code source) ; et pour finir, une partie *footer* qui contiendra soit le bouton start présent dans la page d'accueil, soit rien, suivi d'un pied de page.

Au niveau du css[6], on retrouve sur chaque page la même charte graphique (tons gris et vert) afin d'avoir une homogénéité dans l'utilisation du client web.

Comment notre application raydium va-t-elle récupérer les actions réalisées par un joueur sur une page html ? Pour raydium, cela concerne la fonction de rappel sur les pages html paramétriques (comme expliqué dans le 1.). Côté html, cela se fera par un formulaire ayant pour action la même page et de la méthode GET qu'utilise ce formulaire (annexe 8). Cela afin d'avoir les différentes valeurs dans l'url de notre page et de pouvoir ensuite découper cette url dans notre application raydium afin de récupérer les différents paramètres du joueur, de son action sur le moment et du jeu.

Cependant les valeurs et paramètres récupérés à partir de l'url sont des caractères. On ne peut donc pas directement avoir des entiers, flottants, degrés, etc... Il est donc nécessaire tout d'abord de convertir la chaine de caractères reçue en un type voulu puis nous pouvons ensuite effectuer les calculs. De même, pour envoyer de nouvelles valeurs sur la page web, nous devons effectuer une conversion dans le sens inverse afin de pouvoir retrouver des caractères, sans cela il y aurait des affichages incohérents de valeurs sur la page web, par exemple un tank ayant 28050 point de vie ou des caractères spéciaux liés à un mauvais encodage au niveau des positions.

Le déroulement normal du jeu à partir de notre client web devient donc simpliste pour le joueur et se déroule de cette façon :

1) Arrivée sur la page d'accueil : accueil.htmlp et appui sur le bouton START.

2) Transition sur la page permettant à notre serveur de connaître son Pseudo et son équipe.

3) Le joueur arrive ainsi sur la page de jeu, il peut d'ores et déjà commencer à jouer en choisissant parmi les diverses possibilités (déplacement tourelle de canon verticalement et/ou horizontalement, déplacement du tank) soit en lançant une action, soit en effectuant un tir avec son tank. Nous avons mis en place une sécurité à ce niveau-là, afin d'empêcher le joueur

[6] Fichier sources de styles d'une page HTML

de pouvoir commander son tank tout en tirant. Il y a donc 2 actions bien différenciées, *Commander* ou *Feu*.

Une autre sécurité a été mise en place pour empêcher le joueur de revenir en arrière et de pouvoir à chaque fois recommencer avec un tank au maximum de ses capacités (nombre de points de vie maximal, apparition à un nouveau spawn). Ceci consiste à l'utilisation d'un champ caché dans la page de jeu qui nous signale si le joueur vient juste de commencer le jeu ou s'il a déjà effectué des actions.

De cette même façon, nous utilisons un champ caché pour l'ID du joueur, afin que celui-ci ne change pas son ID et ne se retrouve à la place ou au contrôle d'un autre tank en jeu.

Les différents paramètres de joueurs (ceux entourés de {{ et }}) remplacés dans notre page de jeu sont : le nom du joueur, son id, son équipe, le nombre de points de l'équipe rouge, le nombre de points de l'équipe bleue, la vie du tank du joueur ainsi que sa position sur l'aire de jeu en X, Y et Z. De plus, les paramètres récupérés à partir de l'url de la page de jeu et permettant le bon fonctionnement du tank du joueur et du jeu sont : l'id du joueur afin de savoir quel tank est à utiliser, son nom afin de pouvoir retrouver le joueur, l'état du jeu (qui est soit *Init* soit *Jeu).*

Si l'état récupéré est *Jeu*, nous allons alors récupérer l'action faite par le joueur et regarder si c'est *Commander*, si c'est le cas, nous regardons le déplacement du tank voulu par le joueur, le déplacement de la tourelle horizontale voulu par le joueur, ainsi que le déplacement verticale de cette tourelle, et en conséquence de ces paramètres, effectuer diverses actions en jeu. Si l'action est *Feu*, nous faisons tirer le tank.

### 5.3.3. Le multijoueur

Si l'état est *Init*, on sait que le joueur joue pour la première fois, et nous allons donc mettre en place le multijoueur au niveau de l'application raydium. Nous devons d'abord savoir si le joueur est déjà connecté grâce à la fonction *find_player(char *name)* (annexe 9), si c'est le cas nous le faisons juste rentrer en jeu graphiquement avec initialisation de son équipe, point de vie et position initiale grâce à la fonction *player_add_num(char *name, int p, char *equipe)* (annexe 10). Sinon nous allons regarder si il y a moins de 16 joueurs connectés et faire comme précédemment en le connectant en jeu, sinon le joueur ne pourra pas se connecter car il n'y a plus suffisamment de place disponible. Une fois le joueur arrivé en jeu, le nombre de joueur présent est augmenté. Il y a bien sur un test d'effectué pour savoir si le nombre limite de 16 joueurs est atteint ou non.

La gestion des équipes est faite par une variable binaire dans la structure d'un joueur, le joueur pouvant définir son équipe grâce à l'application web. Nous devons savoir quelle équipe il a choisi, si l'équipe est rouge, la variable présente dans la structure d'un joueur sera de 0, si c'est une équipe bleue elle sera de 1. Le choix de l'équipe sur la page d'initialisation est crucial pour le point d'apparition d'un joueur. Dans l'équipe rouge, le joueur apparaitra entre les points de réapparition 1 et 8, alors que dans la bleue ce sera entre les points de réapparition 10 et 17.

Le fait d'appartenir à une équipe définit aussi le montant total des points d'une équipe. En effet, un joueur dans l'équipe rouge qui est détruit fait gagner des points à l'équipe adverse.

Ce système de choix d'équipe permet donc d'éviter qu'un joueur qui meurt ne fasse gagner des points à sa propre équipe.

## 5.4. Connexion avec client Android

### 5.4.1. Serveur de communication sous Raydium

La connexion au serveur est effectuée par l'envoi de trames dans un réseau local. Pour permettre ces différents envois entre le serveur et les terminaux, nous avons dû créer des sockets sur le serveur et sur les terminaux pour mettre en place la communication. Ces sockets sont donc ouvertes au lancement du serveur et/ou du mobile). Pour cela nous avons d'abord implémenté la partie serveur en utilisant différentes commandes.

- Tout d'abord la commande *sockets* qui permet de créer des sockets différentes en fonction des différents protocoles.
- Ensuite la commande *bind* nous permet de rattacher cette socket sur un port de l'ordinateur afin qu'il puisse être connecté sur une adresse externe bien précise.

Pour pouvoir rendre le jeu complètement indépendant du serveur, nous avons dû mettre en place un broadcast : c'est-à-dire que le serveur devra envoyer, à intervalles réguliers, une trame contenant son adresse IP et son port de communication à tous les terminaux connectés sur le même réseau que lui. Après l'envoi de son adresse IP, le serveur attend la réponse d'un ou de plusieurs clients en recevant le nom du joueur et l'équipe à laquelle il s'inscrit.

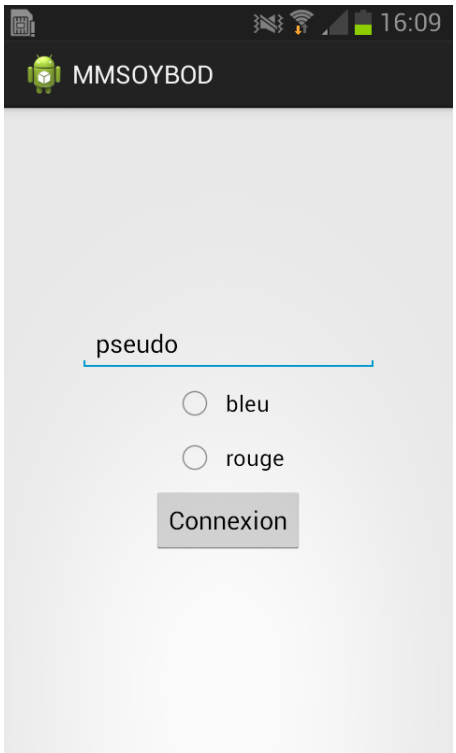



Figure 8 – Page d'accueil

### 5.4.2. Connexion du client

En développant la partie mobile, nous avons dû mettre en place un thread, c'est à dire que deux tâches différentes vont s'exécuter en même temps. Les deux tâches qui s'exécutent en même temps sont la connexion du mobile au serveur distant ainsi que l'affichage d'une petite fenêtre pour indiquer à l'utilisateur que le mobile est en train de se connecter au serveur.

La connexion s'effectue quasiment de la même manière que pour le serveur, à savoir : création de la socket, connexion à un port, et réception/envoi des trames. Pour cela, le mobile écoute sur un port précis et attend une trame en provenance du serveur. Après réception de celle-ci, le mobile connaît l'adresse IP et le port de communication du serveur. Il répond donc au serveur (avec pseudo et équipe du client) et attend une réponse de sa part qui lui signale ainsi qu'il est bien connecté au serveur de jeu et lui transmet sa position sur la carte.

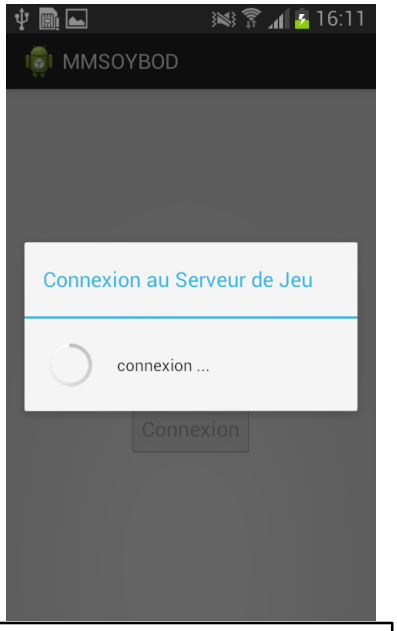



Figure 9 – Écran de chargement

Pour pouvoir permettre une connexion facile au serveur, l'écoute du broadcast (message transmis par le serveur) s'exécute en tâche de fond pendant que l'utilisateur saisit son pseudo et son équipe. Lorsque celui-ci appuiera par la suite sur le bouton connexion, le terminal aura déjà en mémoire l'adresse du serveur ainsi que son port de communication. La connexion sera alors plus simple car il n'y aura plus qu'à transmettre les données saisies par l'utilisateur pour que la connexion soit effective.

Une fois la connexion établie, le mobile pourra donc envoyer des ordres de jeu au serveur qui pourra les exécuter. A cause de la possibilité de perte des trames UDP, le mobile enverra sa trame avec un timer jusqu'à ce que le serveur envoie une réponse pour confirmer la réception du message.

## 5.5. Modélisation et design

### 5.5.1. État initial

Les différents éléments du tank, c'est à dire la tourelle, la base, et les chenilles sont déjà modélisés et nous décidons de ne pas les modifier car ils correspondent à ce dont nous avons besoin. L'ensemble de l'aire de jeu doit être pensé et modélisé et la totalité des textures doivent être créées.

### 5.5.2. Prise en main des fonctions de modélisation

Au niveau du logiciel de modélisation Blender, dans un premier temps il a donc fallu, comme pour les autres outils, apprendre à utiliser ce logiciel. En effet bien qu'ayant de grosses capacités cet outil est assez compliqué à prendre en main car assez peu intuitif au début, néanmoins nous nous sommes adaptés rapidement au fonctionnement de ce logiciel. De nombreux raccourcis-clavier sont quasiment indispensables pour pouvoir utiliser pleinement les différentes fonctionnalités proposées il a donc fallu les apprendre.

Pour avoir un apprentissage rapide et efficace nous avons donc décidé de regarder un grand nombre de tutoriaux, premièrement à propos des connaissances nécessaires pour une utilisation basique puis à propos des fonctionnalités dont nous aurions besoin tout au long du projet.

Après nous être entrainés en modélisant différents objets simples comme par exemple un arbre avec très peu de faces, composé d'un cylindre et d'une iso-sphère, ou encore une maison ou une barrière routière, elles aussi composées de formes simples comme un cube et une pyramide pour la maison.

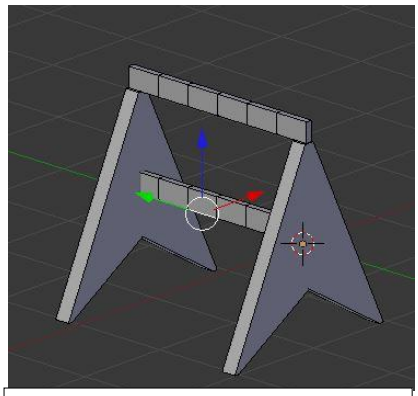


Figure 10 - Barrière routière

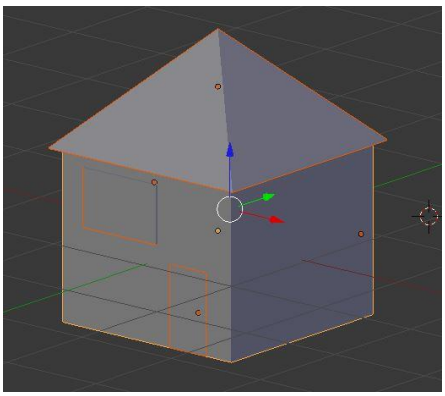


Figure 11 - Maison

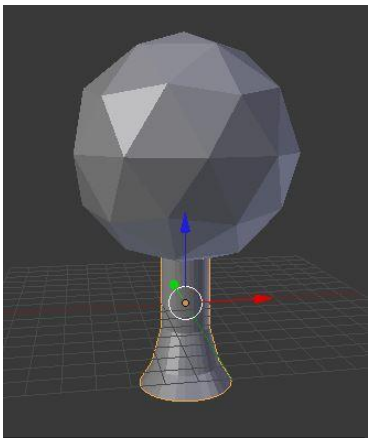


Figure 12 - Arbre

Nous avons ensuite commencé à créer des objets plus complexes comme ce personnage qui a été assez long à modéliser mais qui nous a permis de bien assimiler les connaissances nécessaires pour la suite du projet. Ce personnage a été construit entièrement en se basant sur des cylindres qui ont été modifiés pour arriver à un résultat satisfaisant.

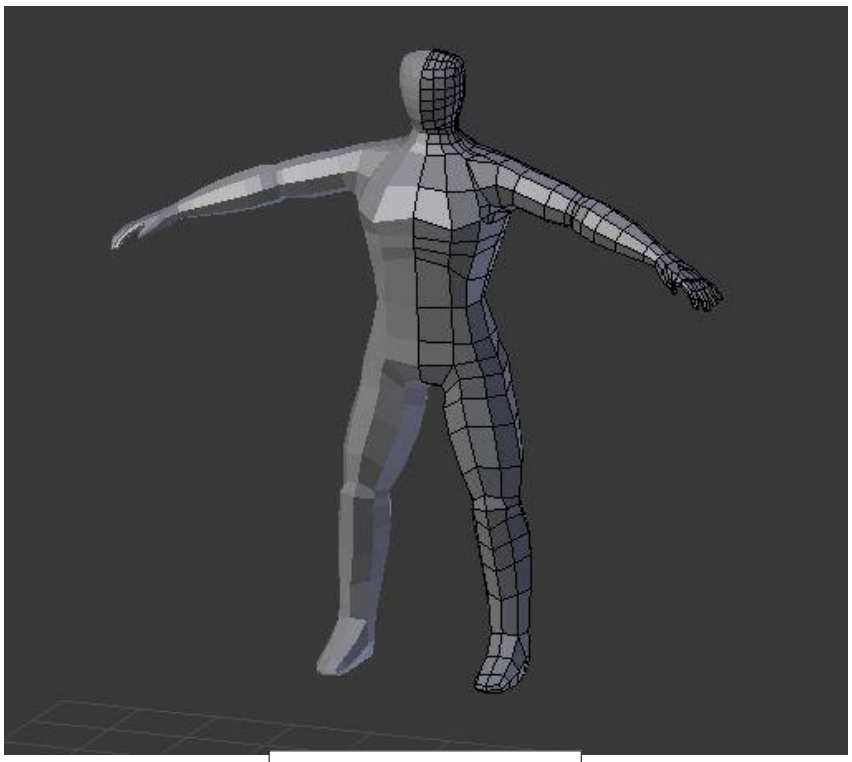


Figure 13 - Personnage

### 5.5.3. Première aire de jeu

Après ces différents essais nous avons finalement créé notre première aire de jeu avec de légers dénivelés et en incluant les différents objets créés précédemment. Ne sachant pas encore incruster les textures nous avons donc mis des couleurs unies pour chaque objet comme on le voit ci-dessous.

Figure 14 - Première aire de jeu

Cette seconde vue prise en jeu, rend compte du problème des couleurs unies qui ne permettent pas de bien juger les distances et qui ne sont pas très esthétiques.


Figure 15 - Première aire-de-jeu (vue en jeu)

### 5.5.4. Le design choisi et sa mise en œuvre

Beaucoup de propositions et d'idées ont été apportées pour choisir comment serait constituée l'aire de jeu. La première était de créer une aire de jeu simple sans dénivelé, une seconde était de reconstituer une aire ressemblant à l'IUT mais cela paraissait complexe, une troisième était de faire une aire proche de la première aire de jeu que nous avions fait mais en ajoutant des textures diverses. Finalement la proposition que nous avons validé était celle qui consistait à créer un espace de jeu séparé en deux par un ravin, chaque côté devant contenir des collines de même hauteur ainsi que des éléments de décors tels que des arbres ou des rochers, cette aire devant être limitée par des murs par exemple constitués de rochers.

Pour commencer nous avons donc dessiné sur papier une ébauche de l'aire de jeu en question, nous avons ensuite créé une face plane sous Blender que nous avons coupé au milieu pour créer le ravin. La première difficulté rencontrée lors des tests était les tanks qui passaient au travers des collines et tombaient dans le vide.

Nous avons donc eu l'idée de créer un objet colline à part que nous poserions sur la face plane, de plus cela nous a permis de simplement faire des copier-coller des collines en nous contentant de les agrandir ou de les rapetisser. Le dénivelé de ces collines a été fait de manière à ce que les tanks puissent les gravir, ce qui permettrait de ne pas avoir à modifier l'aire de jeu dans le cas où nous voudrions ajouter les déplacements. Après quelques tests nous nous sommes rendu compte que le sommet des collines arrondies ne permettait pas une bonne stabilité pour tirer les missiles depuis les tanks.

Après quelques tests nous nous sommes rendus compte que le sommet des collines arrondies ne permettait pas une bonne stabilité pour tirer les missiles depuis les tanks. Nous avons donc mis en place une modification pour palier à ce problème. Pour ne pas compliquer d'avantage et dans un souci d'optimisation des performances sur le serveur qui fera

fonctionner le jeu, j'ai simplifié certaines parties de l'aire de jeu comme les murs qui servent de limite que j'ai rendu plat.


Figure 16 - Aire-de-jeu finale vide

Cette aire de jeu déjà utilisable, n'a aucune texture seulement des nuances de gris, il faut donc apprendre à incruster des textures aux objets, pour cela le visionnage de tutoriaux a encore une fois été d'une grande aide car cette partie était amplement plus complexe que nous ne le pensions.

Prenons l'exemple d'une texture de rocher, pour créer cette texture il faut commencer par récupérer une image qui pourrait convenir pour servir de texture ou bien en créer une à partir de Photoshop, il faut ensuite l'ouvrir avec GIMP et la redimensionner à l'une des tailles suivantes : 64x64, 128x128, 256,256, 512x512 ou 1024x1024 puis l'exporter en .tga (seul GIMP peux le faire).

Pour pouvoir finalement l'utiliser dans Blender, pour cela il faut sélectionner les faces ou l'on souhaite appliquer cette texture, il faut ensuite mettre ces faces à plat à l'aide de l'option « unwrap » qui effectue un dépliage UV, après cela on peut choisir l'orientation de la texture sur les faces en appliquant des rotations ou bien des translations à ces faces.

La principale difficulté est d'éviter les faux raccords entre différentes faces qui se suivent, c'est à dire, un trait visible séparant les deux faces dues à des textures mal positionnées. On voit ci-dessous une modélisation d'un rocher avec et sans texture, et on voit clairement que l'ajout de texture permet de donner un rendu plus réel aux objets.

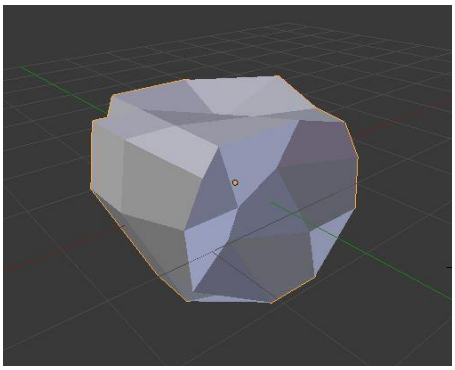


Figure 17 - Rocher sans texture

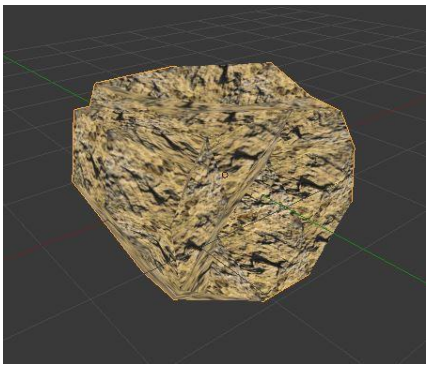


Figure 18 - Rocher avec texture

Il a ensuite fallu créer la totalité des textures, en général en nous basant sur des images trouvées sur le web, comme les troncs d'arbres, l'eau ou les roches comme on le voit ci-dessous. Chacune des textures a été choisie pour permettre une bonne lisibilité de l'environnement par le joueur.

Figure 19
Texture arbre

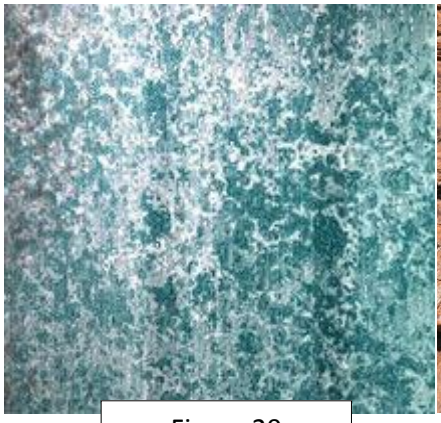


Figure 20
Texture eau

Figure 21
Texture roche rouge

Des images auxquelles nous avons dû faire subir les mêmes modifications que pour la roche présentée précédemment, pour nous permettre de les appliquer sur les différents objets de l'aire de jeu.


Figure 22 - Aire-de-jeu finale texturé

Après cela nous nous sommes rendu compte que l'aire de jeu était assez vide, nous avons donc ajouté des arbres et des rochers. Pour éviter tout ralentissement en jeu les arbres et les rochers sont constitués du minimum de faces possibles et un nombre restreint de chacun de ces objets a été implanté sur l'aire de jeu.

Le problème était que tous les rochers et que tous les arbres étaient identiques, nous avons donc décidé de leurs donner des tailles différentes mais cela ne suffisait pas. La solution trouvée a été de créer une nouvelle texture de rocher et une nouvelle texture d'arbre ce qui permet d'avoir deux arbres différents et deux rochers différents et qui en donnant des tailles différentes permet de ne pas donner l'impression que tous ces objets sont identiques. (annexe 11).

À ce moment-là l'aire de jeu était terminée il fallait donc ajouter les points de réapparition pour les tanks, il a été choisi d'en placer huit de chaque côté du ravin. Les emplacements ont été choisis de manière à ce que chaque tank puisse être touché et puisse toucher les ennemis. Dans cette optique, un point de réapparition a été placé sur chaque colline soit quatre points de chaque côté. Les points restants ont été placés entre les collines au nombre de quatre de chaque côté également. Pour créer ces points de réapparitions, il faut créer un objet, par exemple une sphère à l'endroit voulu puis regrouper tous les points de cette sphère en un seul point grâce à l'option « merge », il faut ensuite que cet objet soit caché pour cela il suffit simplement d'ajouter un point devant son nom, par exemple « .spawn ».

Lors de l'exportation le format choisi doit être « .tri » pour pouvoir être utilisable sous Raydium, il faut également cocher l'option qui permet de générer un fichier « .pos » qui enregistre la position de chaque objet de l'aire de jeu et qui permettra donc de situer précisément l'emplacement de chacun des points de réapparition à partir de leur nom.

# 6- Bilan Technique

Au niveau du serveur Raydium, nous avons pu mettre en place une connexion « multijoueur » ainsi que le déplacement du tank et de la tourelle à partir de la page web. L'ajout dynamique d'un joueur a aussi été réalisé ainsi que la communication entre le serveur et la page web (réponses dynamiques, remplacement des valeurs dans la page web).

Par contre, certaines améliorations ou ajouts peuvent être apportés à ce serveur tels que la connexion et communication avec un mobile Android, l'ajout d'une base de données pour conserver les données d'un joueur (meilleur score, nombre tués / nombre de morts…) ou encore la mise en place d'une mini-carte avec la position des tanks.

Au niveau du moteur Raydium, nous avons réalisé le développement d'une application où le joueur peut déplacer son tank, déplacer sa tourelle horizontalement et verticalement, et tirer des missiles. Un temps de rechargement a été mis en place tout comme un système simpliste de gestion de vie. Enfin un début de système de points a été instauré, à chaque fois qu'un joueur a sa vie inférieure ou égale à 0, l'équipe adverse gagne 100 points. Et lorsqu'une équipe atteint 2000 points, la partie s'arrête.

Par contre, nous aurions aimé modifier la gestion de la vie, en effet la collision d'un missile sur un tank est gérée grâce à un calcul simpliste qui n'est pas forcément le plus adapté. D'autre part, différentes sortes d'armes pourraient être instaurées. La gestion de la période lorsqu'un tank est dans l'état « mort » doit être débogué et pourrait être amélioré. Enfin, une limite de temps de jeu pourrait être mise en place, tout comme un score individuel.

Au niveau de l'application Android, la connexion au serveur de Raydium est effective ainsi que les fenêtres de login et de chargement (connexion au serveur).

Par contre, nous aurions aimé amélioré la connexion au serveur ainsi que la page de login. Enfin nous aurions aimé mettre en place la fenêtre de jeu ainsi que un déplacement de la tourelle par orientation du téléphone. Enfin nous aurions voulu récupérer la vue du tank du joueur et l'afficher sur son smartphone.

Pour ce qui concerne l'aire de jeu, la modélisation a été complétement réalisée selon les choix de l'équipe, ainsi que l'ajout de textures sur tous les éléments du décor.

Enfin pour améliorer celle-ci, nous aurions aimé ajouter des éléments de décor avec leurs textures ainsi que l'ajout d'animation pour la cascade centrale. Nous aurions aussi aimé agrandir la carte pour augmenter le nombre de joueurs (32 en 16x16).

# 7 – Conclusion

Ce projet nous a permis de découvrir le fonctionnement d'un groupe de travail sur un projet conséquent. L'organisation des tâches entre les différentes personnes sur un projet conséquent a été quelque chose de nouveau, ce qui nous a permis d'avoir une vision concrète d'un projet professionnel. Étant donné le nombre d'étudiants investis dans ce projet, la communication a été un facteur important dans la bonne conduite de cette réalisation, ce que nous avons réussi à mettre en place au fur et à mesure.

Cette expérience nous a aussi montré l'importance d'une gestion de projet afin de nous guider pas à pas durant ces trois mois. De plus, le fait de respecter une date de livraison du travail, nous a imposé un travail rigoureux et méthodique.

Nous sommes cependant déçus de ne pas avoir pu aller au bout de ce que nous avions prévu de réaliser, car malgré le fait que le jeu soit fonctionnel, cela aurait été une vraie satisfaction d'y jouer à partir d'une version Android. Nous aurions aimé pouvoir réaliser ce projet grâce à un moteur 3D implémentant la programmation orientée objet comme Ogre3D. Il serait donc une bonne chose de proposer ce projet aux futurs étudiants mais en utilisant Ogre3D plutôt que Raydium.

# 8 – Bibliographie


http://wiki.raydium.org/wiki/Raydium

http://fr.wikipedia.org/wiki/Blender

http://developer.android.com/

http://docs.oracle.com/

http://sberfini.developpez.com/tutoriaux/android/broadcast_udp/

http://androtruc.wordpress.com/2010/07/19/le-cycle-de-vie-dune-application-android/

http://www.techrepublic.com/blog/software-engineer/androids-indeterminate-progressdialog-tutorial/

http://wiki.raydium.org/wiki/RaydiumApiReference

http://wiki.raydium.org/wiki/ApiNetworkIntro

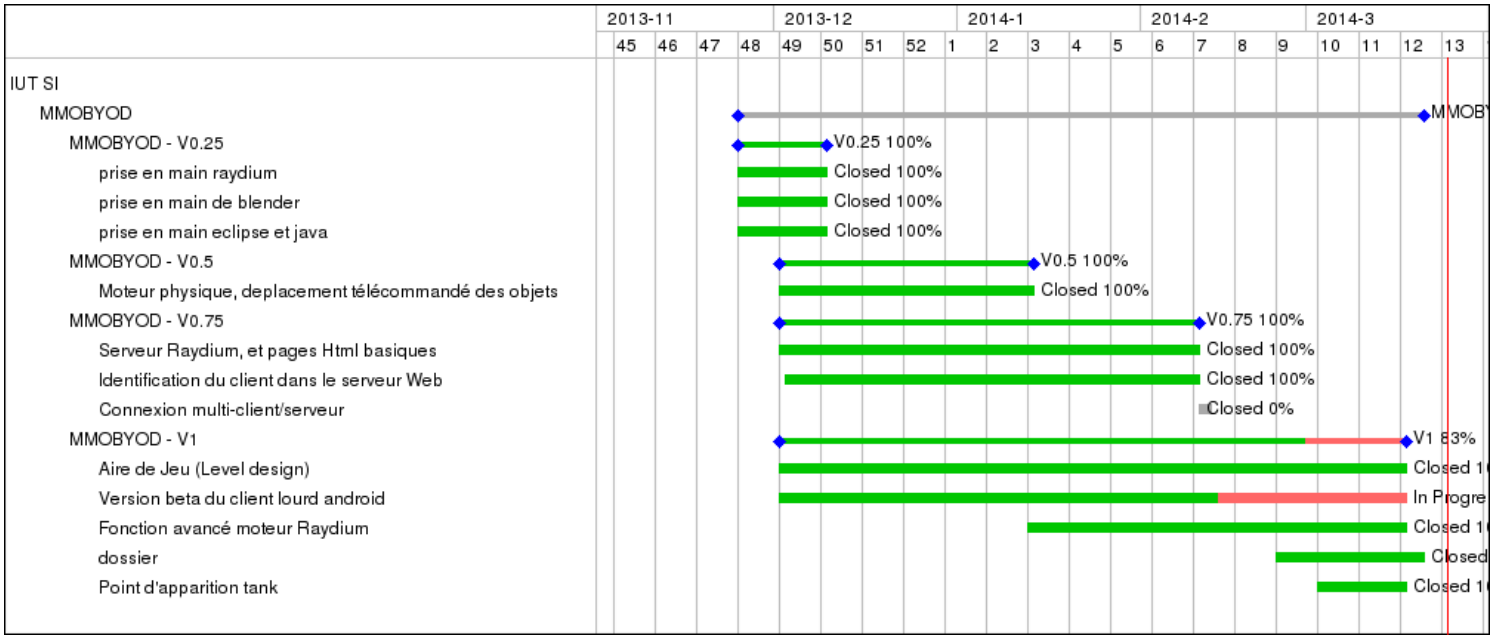
http://wiki.blender.org/index.php/Doc:FR/2.6/Manual

https://www.youtube.com/playlist?list=PLDHwkMpIGgVzWfBPDPHaVzFNPOIk5ckNB

# 9 – Annexes

## Table des matières

# Annexe 1 : Diagramme de Gantt final du projet



# Annexe 2 : Fonction d'initialisation d'un joueur

```
//Initialisation des joueurs
//TODO: faire une initilisation par joueur à appeler en boucle.
void player_init(int i){

    if(i < MAX_PLAYERS)
    {
        player[i].name[0]='\0';
        player[i].id[0]='\0';
        player[i].active=0;
        player[i].tank_object=-1;
        player[i].tank_motor_l=-1;
        player[i].tank_motor_r=-1;
        player[i].tank_body=-1;
        player[i].tank_turret=-1;
        player[i].tank_turret_motor=-1;
        player[i].tank_canon=-1;
        player[i].tank_canon_motor=-1;
        player[i].az=0;
        player[i].el=0;
        player[i].ammo=0;
        player[i].reload=0;
        player[i].life=100;
        player[i].equipe=raydium_random_0_x(1);
    }
}
```

# Annexe 3 : Parseur d'une requête HTML

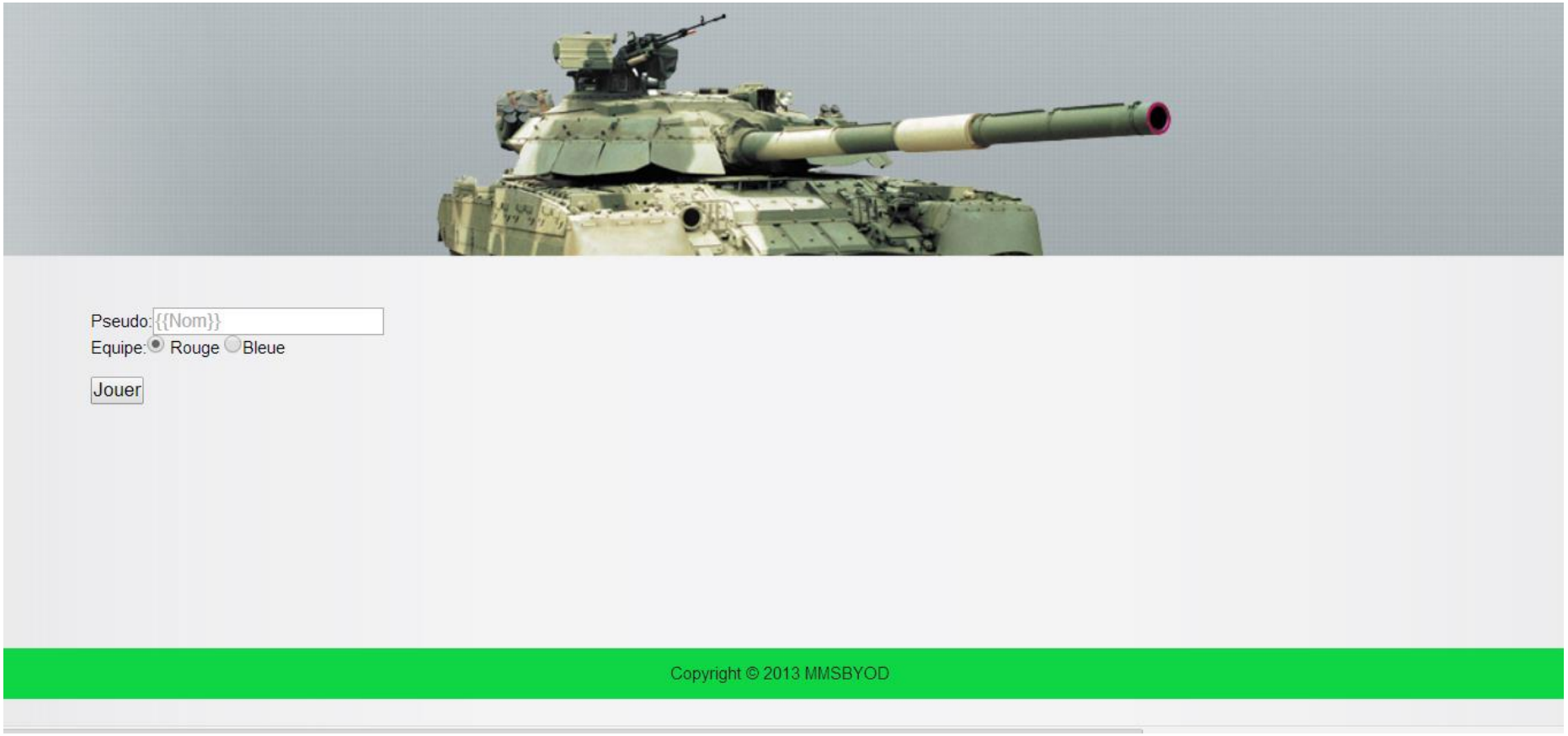
```
void parse_requete(char *page_name,char * req,char req_param[32][64],char req_param_value[32][64],int * req_n_param){
char *p,*pn;

    p=req;
    p=strchr(req,'?');
    *req_n_param=0;
    if (p){ // Si l'url contient des paramètres
        strncpy(page_name,req,p-req);
        page_name[p-req]='\0';
        p++;
        strcat(req,"&"); //Simplification de la recherche des paramètres
        do {
            pn=strchr(p,'=');
            if (!pn) break; // Pas de = parametre sans valeur. On quitte
            strncpy(req_param[*req_n_param],p,pn-p); //TODO verifier la longueur
            req_param[*req_n_param][pn-p]=0;
            p=pn+1;
            pn=strchr(p,'&');
            strncpy(req_param_value[*req_n_param],p,pn-p);
            req_param_value[*req_n_param][pn-p]=0;
            p=pn+1;
            (*req_n_param)++;
        }while (*req_n_param<32); // Max 32 paramètres pas chaine.
    }else
        strncpy(page_name,req,strlen(req)+1);
}
```

# Annexe 4 : Fonction de remplacement

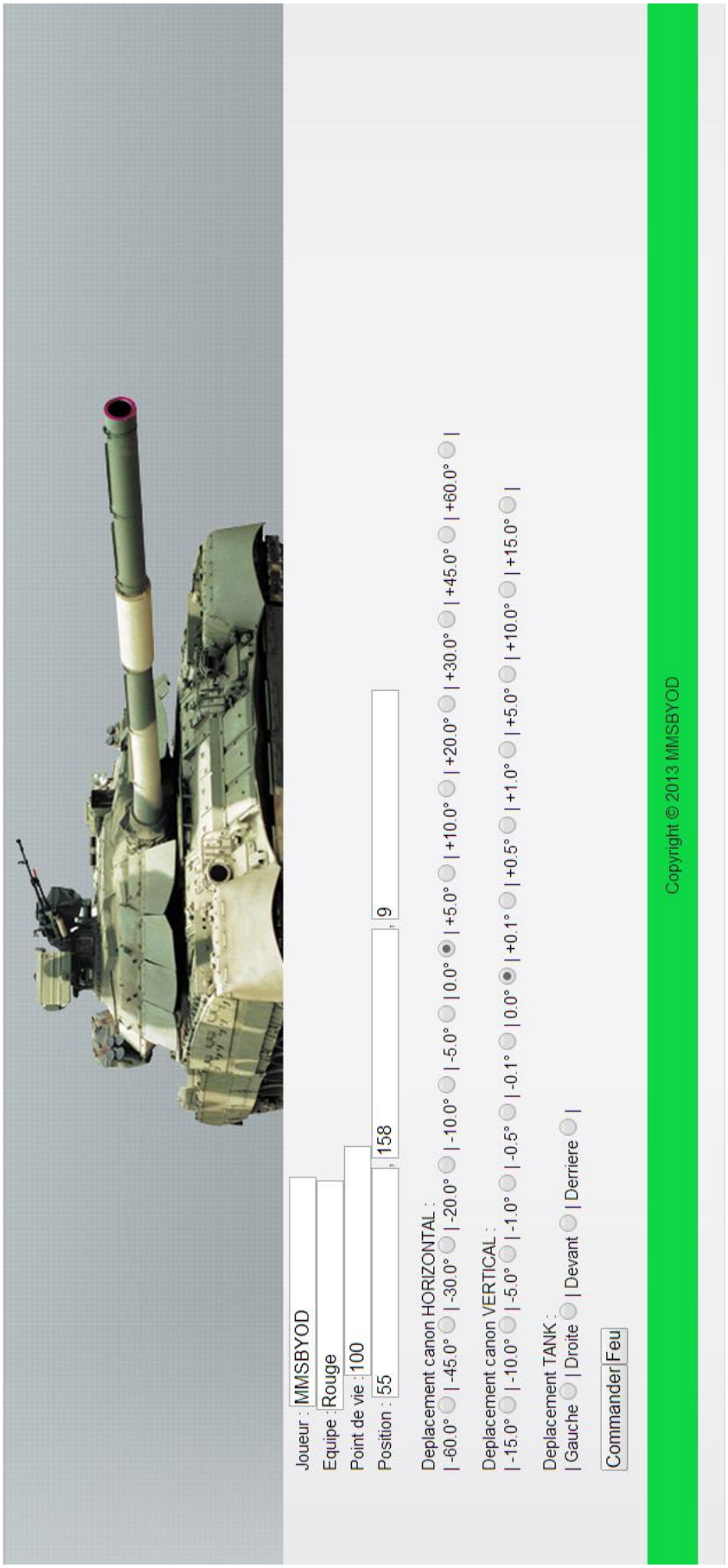
```
699  int str_replace(char * source, char * find, char * change){
700      char *pos,*last_pos, *orig;
701      char tmp_str[RAYDIUM_WEB_BUFSIZE];
702      int find_len, change_len, n_find;
703          strcpy(tmp_str,source);
704          orig=source;
705          find_len=strlen(find);
706          change_len=strlen(change);
707          n_find=0;
708          last_pos=tmp_str;
709          while ((pos=strstr(last_pos,find))){ // Trouve le motif
710              n_find++;
711              if(orig==source){// Première occurence
712                  orig+=pos-tmp_str; // Première itération on saute les premieres caractères qui ne sont pas modifiés.
713              }else{ // Iteration suivante ... On est plus au début du fichier
714                  if ((orig+(pos-last_pos))>(source+RAYDIUM_WEB_BUFSIZE))
715                      break; // On evite un depassement de la taille du buffer
716                  strncpy(orig,last_pos,pos-last_pos); // On copie les caractère entre la fin de la dernière occurence et le debut de la nouvelle
717                  orig+=pos-last_pos; // Nouvelle origine de travail
718              }
719              if ((orig+change_len)>(source+RAYDIUM_WEB_BUFSIZE))
720                  break; // On evite un depassement de la taille du buffer
721              strncpy(orig,change,change_len);
722              orig+=change_len; // Ajout des caractère changés.
723              last_pos=pos+find_len; // Nouvelle position de référence dans la chaine de référence
724          }
725          if(pos==NULL){ //Sortie Normale
726              if(orig==source) // Rien n'a été trouvé, on laisse tel quel
727                  return 0;
728              pos=tmp_str+strlen(tmp_str); // Dernier caractère
729              if ((orig+(pos-last_pos))>(source+RAYDIUM_WEB_BUFSIZE))
730                  return -1; // On evite un depassement de la taille du buffer
731              strncpy(orig,last_pos,pos-last_pos);
732              orig+=pos-last_pos;
733              *orig='\0'; //Termine la chaine
734              return n_find;
735          }
736          *orig='\0'; //Termine la chaine au mieux.
737          return -1; // Sortie sur depassement de la taille du buffer.
738  }
```

# Annexe 5 : Page d'accueil du serveur Web

# Annexe 6 : Page d'initialisation d'un joueur sur le serveur Web



# Annexe 7 : Page de jeu sur le serveur Web



# Annexe 8 :
# Formulaire d'envoi de données de la page HTML du serveur

```
<form target="_self" method="GET" action="jeu.htmlp" name="Jeu"> Joueur :

        <input readonly="readonly" value="{{Nom}}" name="Nom" type="text" />
        <input readonly="readonly" value="{{Id}}" name="Id" type="hidden" /><br>
        Equipe :<input readonly="readonly" value="{{Equipe}}" name="Equipe" type="text" /><br>
        Point rouge :<input readonly="readonly" value="{{PointR}}" name="PointRouge" type="text"
/> // Point bleue :<input readonly="readonly" value="{{PointB}}" name="PointBleue" type="text"
/><br>
        Point de vie :<input readonly="readonly" value="{{Vie}}" name="Vie" type="text" /><br>
        Position : <input readonly="readonly" value="{{PosX}}" name="PositionX" type="text" />,
<input readonly="readonly" value="{{PosY}}" name="PositionY" type="text" />, <input
readonly="readonly" value="{{PosZ}}" name="PositionZ" type="text" /><br><br>
        Deplacement canon HORIZONTAL :<br>
        | -60.0&deg; <input value="-60" name="AzimH" type="radio" />
        | -45.0&deg; <input value="-45" name="AzimH" type="radio" />
        | -30.0&deg; <input value="-30" name="AzimH" type="radio" />
        | -20.0&deg; <input value="-20" name="AzimH" type="radio" />
        | -10.0&deg; <input value="-10" name="AzimH" type="radio" />
        | -5.0&deg; <input value="-5" name="AzimH" type="radio" />
        | 0.0&deg; <input checked="checked" value="0.0" name="AzimH" type="radio" />
        | +5.0&deg; <input value="5" name="AzimH" type="radio" />
        | +10.0&deg; <input value="10" name="AzimH" type="radio" />
        | +20.0&deg; <input value="20" name="AzimH" type="radio" />
        | +30.0&deg; <input value="30" name="AzimH" type="radio" />
        | +45.0&deg; <input value="45" name="AzimH" type="radio" />
        | +60.0&deg; <input value="60" name="AzimH" type="radio" /> |<br><br>
        Deplacement canon VERTICAL :<br>
        | -15.0&deg; <input value="-15" name="AzimV" type="radio" />
        | -10.0&deg; <input value="-10" name="AzimV" type="radio" />
        | -5.0&deg; <input value="-5" name="AzimV" type="radio" />
        | -1.0&deg; <input value="-1" name="AzimV" type="radio" />
        | -0.5&deg; <input value="-0.5" name="AzimV" type="radio" />
        | -0.1&deg; <input value="-0.1" name="AzimV" type="radio" />
        | 0.0&deg; <input checked="checked" value="0.0" name="AzimV" type="radio" />
        | +0.1&deg; <input value="0.1" name="AzimV" type="radio" />
        | +0.5&deg; <input value="0.5" name="AzimV" type="radio" />
        | +1.0&deg; <input value="1" name="AzimV" type="radio" />
        | +5.0&deg; <input value="5" name="AzimV" type="radio" />
        | +10.0&deg; <input value="10" name="AzimV" type="radio" />
        | +15.0&deg; <input value="15" name="AzimV" type="radio" /> |<br><br>
        Deplacement TANK :<br>
        | Gauche <input value="8" name="Dep" type="radio" />
        | Droite <input value="2" name="Dep" type="radio" />
        | Devant <input value="1" name="Dep" type="radio" />
        | Derriere <input value="4" name="Dep" type="radio" /> |<br><br>
        <input value="Jeu" name="Etat" type="hidden" />
        <input value="Commander" name="Action" type="submit" /><input value="Feu"
name="Action" type="submit" /><br><br>
</form>
```

# Annexe 9 :
# Fonction pour récupérer le numéro d'un joueur en fonction de son pseudo

```
279 /**
280  * @brief Trouve le numéro de joueur
281  * @param char * name : nom du joueur
282  * @return int : numéro de joueur, -1 si non trouver
283  */
284 int find_joueur(char * name) {
285     int i;
286     for(i=0; i < MAX_PLAYERS; i++) {
287         if(strcmp(player[i].name, name) == 0)
288             return i;
289     }
290     return -1;
291 }
```

# Annexe 10 : Fonctions utilisées pour l'ajout d'un joueur au serveur

```
312  void player_add_num(char * name,int p, char * equipe){
313  char tmp[256];
314  dReal tx,ty,tz,rx,ry,rz;
315
316      simul_create(p);
317      strcpy(player[p].name,name);
318      player[p].active=1;
319      player[p].life=100;
320      raydium_log("Connection du joueur %s.", name);
321      camera_cible=p;
322      camera=suivi;
323      if(strcmp(equipe, "Rouge") == 0) {
324          player[p].equipe = 0;
325          sprintf(tmp,"spawn_%d",p+1);
326      }
327      else if(strcmp(equipe, "Bleue") == 0) {
328          player[p].equipe == 1;
329          sprintf(tmp,"spawn_%d",p+10);
330      }
331
332      _raydium_pos_rot_from_file("simple_scene.tri.pos",tmp,&tx,&ty,&tz,&rx,&ry,&rz);
333      raydium_ode_object_move_3f(player[p].tank_object,tx,ty,tz);
334  }
```

```
342  int player_add(char * name, char *equipe){
343  int p;
344
345
346      p=player_find_free();
347      if (p==-1){
348          raydium_log("ERROR: No more slot, cannot connect.");
349          return -1;
350      }
351      else {
352          N_player++;
353          player_add_num(name, p, equipe);
354      }
355      return p;
356  }
```

# Annexe 11 : Aire-de-jeu complète