Rapport détaillé : BomberSow Project

Développeurs :

- Brieulle Ludovic : Implémentation Système, Graphisme
- Norindr Ananda : Implémentation Moteur physique, Basic Graphical User Interface
- Raffre Jonathan : Implémentation Réseau, Implémentation Orientée Objet

I – But du projet

Notre projet vise à créer un jeu « simple » graphiquement, le gameplay consistant en un jeu de tir rapide, mêlé à une philosophie de jeu de plate-forme, et jouable en multi-joueurs, le tout développé dans un langage imposé, le **C**. Nous souhaitons de plus permettre aux joueurs de créer leurs maps (terrains) de jeu grâce à un éditeur de map.

II – Cahier des charges – Description du gameplay

Dans ce jeu, le joueur pourra commencer rapidement et simplement une partie avec au maximum 8 de ses amis. Grâce à un système de serveur intégré au jeu, il pourra créer ou rejoindre une partie crée par ses amis en quelques clics, en pouvant utiliser le pseudonyme qui lui convient (un système de discussion lui permettra même de discuter en direct avec les autres joueurs).
Deux modes de jeu seront à sa disposition:

- Deathmatch (DM) ou chacun pour soi. Dans ce mode de joueur doit simplement éliminer le plus de nombres de joueurs possibles pendant le temps imparti.
- Team DeathMatch (TDM). Exactement le même concept que le DM sauf que cette fois-ci le combat se passe en équipe.

Pour atteindre ces objectifs, le joueur a à sa disposition 7 armes, dont deux spéciales, et un JetPack dans certains niveaux. Voici une liste de ses accessoires.

Armes Normales

Ce sont des armes excellant chacune dans son domaine avec un temps de réapparition modéré, tout comme leur puissance.

- Pied de biche ou Crowbar. Inspiré de la très célèbre arme de Half-Life, c'est une arme de base que le joueur possède en commençant la partie ou en réapparaissant après une mort. Très rapide et infligeant des dommages décents, l'inconvénient de cette arme est qu'elle n'est utilisable uniquement en combat rapprocher.
- Shotgun. Fusil à pompes classiques, il utilise un tir à dispersion très puissant. Cependant le temps de rechargement est relativement élevé et la portée limitée.
- MachineGun. Mitraillette avec un débit très important mais avec des dommages relativement faible.
- Grenade. Grenade classique, pratique pour atteindre les endroits hors de portée des autres armes, dommage important sur une surface assez grande. Le contrôle des rebonds et de la trajectoire est cependant plutôt compliqué.
- RocketLauncher. Lance-roquette portable, de gros dommage sur une grande surface, malheureusement le temps de rechargement et l'anticipation pour toucher les joueurs demandent une certaine accoutumance.

Armes spéciales :

Les armes spéciales sont beaucoup plus puissantes que les armes normales, cependant leur temps de respawn et beaucoup plus importants et leur utilisation très limitée, en temps ou en munitions.

- Sniper. Un fusil de haute précision envoyant un projectile à très grande vitesse. Une seule balle suffit à tuer un adversaire.
- Lasergun. Une arme évoluée produisant un laser d'une certaine longueur faisant un grand nombre de dégâts par seconde.

Accessoires :

Les accessoires sont des objets mis à disposition du joueur pour faciliter son affrontement contre les autres joueurs.

- Bulles de soins. Permet de récupérer un nombre de point de vie conséquent afin d'allonger la survie du joueur.
- Jetpack. Engin permettant au joueur de se déplacer comme un oiseau dans le niveau. Il sera très utile pour les niveaux en haute altitudes.

Pour accéder à tous les recoins d'un niveau, le joueur a la possibilité de se déplacer de droite à gauche, sauter et effectuer un double saut qui se produit en l'air une fois qu'on appuie à nouveau sur la toucher « sauter ». Un double saut ne peut se faire qu'une fois pendant un saut complet.

Le joueur a aussi la possibilité de créer ses propres niveaux, les enregistrer et les partager avec ses amis. Tout ça grâce à l'éditeur de niveau programmé par nos soins.

IV – Conception

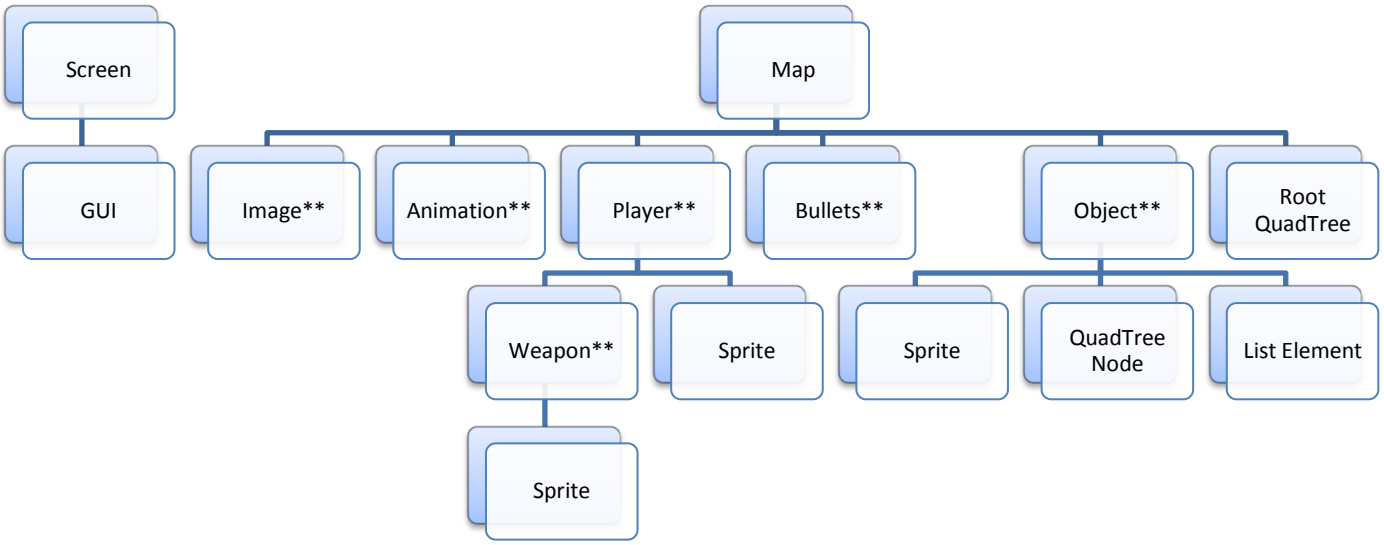
Pour la conception de ce jeu, nous utiliserons une philosophie de programmation orientée objet malgré le fait que le C ne soit pas prévu pour faciliter cette approche, cependant ceci est l'approche la plus approprié pour un tel projet. Ainsi nous allons considérer les structures crées comme des objets, et nommer les fonctions en rapport avec ces objets avec la convention « nom-de-l'objet_nom-de-la-fonction », ainsi que nous fixer une règle stricte que nous allons essayer de respecter dans la mesure du possible et du temps imparti : Ne pas accéder dans une fonction concernant un objet aux propriétés d'un type d'objet différent directement, mais par une fonction dédiée.

Pour nous faciliter la conception, nous allons utiliser principalement une librairie appelée SFML (Simple & Fast Multimedia Library), plus particulièrement sa version C nommée CSFML, ainsi que d'autres librairies nommées GLEW (The OpenGL Extension Wrangler Library) et halloc pour le débogage (modification de malloc détectant les fuites mémoires possibles).

Nous avons crée une surcouche à la CSFML pour pouvoir gérer les entrées de texte et des boutons, cependant nous ne traiterons pas d'elle dans ce rapport, cette surcouche ne faisant pas partie du projet lui-même. Elle est désignée par l'objet Gui dans le code.

Note : Tout type ou fonction commençant par « sf » fait partie de la SFML, celles commençant par « gl » de la GLEW.

Organigramme de dépendances objet : (Il ne prend en compte que les composants crées)



Notes
*** : Tableau***

Affectation et utilité de chaque objet :

- Map : Objet père, englobe tous les objets touchant au gameplay
  - Image** : Tableau d'objets Image, contenant les images de la map
  - Animation** : Tableau d'objets Animation contenant les animations de la map
  - Player** : tableau d'objets Player contenant tous les joueurs de la map
    - Weapon** : Tableau d'objets Weapon contenant toutes les armes du jeu
      - Sprite : Image des munitions de chaque objet Weapon
    - Sprite : Image du joueur
  - Bullet** : Tableau d'objets Bullet contenant les munitions tirées par les joueurs
  - Object** : Tableau d'objets Object (Plateformes, munitions, etc.)
    - Sprite : Image de l'Object
    - QuadTree Node : Nœud contenant l'Object en question
    - List Element : Élément de la liste contenue dans le QuadTree concernant l'Object en question.
  - Root Quadtree : Racine de l'arbre quaternaire de la map, pour la gestion physique.
- Screen : Objet père, englobe ce qui concerne l'interface utilisateur
  - GUI : Interface utilisateur (Boites de texte, boutons)

Fonctions principales génériques (existantes pour tous les objets) :

- Create : Création et initialisation de l'objet
- Destroy : Destruction de l'objet, quelques variantes existent suivant l'objet
- Set : Modifie l'attribut donné par le nom de la fonction de l'objet
- Get : Renvoie l'attribut donné par le nom de la fonction de l'objet
- Get*X*From*Y* : Renvoie X à partir de la donnée Y envoyée en paramètre
- Add/Del(ete) : Ajout/Suppression de l'objet donnée en second argument dans la structure donnée en premier argument

Structure et fonctions de chaque objet :

Partie Jeu

Map :

```
typedef struct MAP
{
    unsigned int mapId;             // ID de la map (transmission réseau)
    unsigned int max_players;       // Nombre maxi de joueurs
    unsigned int cpt_players_rev;   // Compteur courant pour les player_id
    sfSprite* background;           // Arrière-plan

    Image* images;                  // Images de la map
    Animation** animations;         // Tableau pour les animations
    unsigned int nb_anim;           // Nombre d'animations

    Object** objects_list;          // Tableau des objets de la map
    unsigned int nb_objects;        // Nombre d'objets sur la map

    Player** players_list;          // Liste des joueurs de la map
    unsigned int nb_players;        // Nombre de joueurs connectés sur la map

    BulletList* bullets;            // Liste des balles tirées

    PacketList* game_packets2send;  // Liste des paquets de jeu à envoyer

    bool chat_started;              // Salon de discussion démarré ?
    bool game_started;              // Partie démarrée ?

    sfSelectorTCP* tcp_selector;    // Sélecteur de sockets TCP pour
discussion/connexion/déconnexion
    sfSocketUDP* game_socket;       // Socket de jeu
    unsigned short game_port;       // Port de jeu

    sfClock* clock;                 // Timer d'actualisation
    float clock_time;               // Temps clock

    struct QUAD_TREE* quad_tree;    // struct QUAD_TREE pour la gestion de collisions

} Map;
```

```
Map* map_Create(unsigned int, unsigned int, Config*);
void map_Destroy();
void map_AddObject(Map*, Object*);
void map_DelObject(Map*, unsigned int);
void map_AddPlayer(Map*, Player*);
void map_DelPlayer(Map*, unsigned int);
void map_AddBullet(Map*, Bullet*);
void map_DelBullet(Map*, Bullet*);
void map_UpdateDisconnectedPlayers(void*);
Player* map_GetPlayerFromID(Map*, unsigned int);
unsigned int map_GetPlayerIDFromName(Map*, char*);
void map_SetGamePort(Map*, unsigned int);
void map_SetCptCurrPlayers(Map*, unsigned int);
void map_Draw(sfRenderWindow*, Map*);
```

`void map_UpdateDisconnectedPlayers(void*)` : Cette fonction sert à gérer la mise à jour des joueurs déconnectés, de façon à ne pas supprimer directement sans vérifications d'accès les joueurs qui se déconnectent.

`void map_Draw(sfRenderWindow*, Map*)` : Cette fonction dessine la map sur l'écran, avec ses objects, bullets, et players.

Cette structure sert à gérer tout ce qui sera affiché à l'écran, ainsi que certains aspects internes tels que le nombre maximum et courant de joueurs, le socket depuis lequel envoyer les données de jeu, le sélecteur de sockets TCP pour les données de connexion/déconnexion et discussion, le port sur lequel le jeu écoute, l'arbre de collision, le temps depuis la dernière actualisation, l'état du jeu.

Image :

```
typedef struct IMAGE
{
    sfImage **image_tab;        // Tableau stockant les images d'une map
    int image_nombre;           // Nombre d'images du tableau


}Image;

Image* image_Create();
void image_Destroy(Image*);

void image_Loader(Image*, char**, int);     // Loader d'image
sfImage* image_Get(Image*, int);            // Récupérer une sfImage à partir de son ID
```

`void image_Loader(Image*, char**, int)` : Cette fonction charge une image supplémentaire dans l'objet Image passé en paramètre.

Cette structure a été crée pour nous faciliter la gestion des images de la map et l'ajout d'images.

Animation :

```
typedef struct ANIMATION
{
    sfSprite *sprite;           // Image convertie en sfSprite de l'animation

    // Coordonnées de la première case
    int x;
    int y;

    // Coordonnée de dessin
    int x_c;
    int y_c;

    // Taille d'une case
    int image_hauteur;
    int image_largeur;

    int nombre_image;
    int cur_image;              // Position dans l'animation
    int play;                   // Nombre de lectures à effectuer
    float fps;                  // Temps d'attente entre chaque image (en s)

    sfClock *clock;             // Timer
} Animation;
```

Note concernant x et y : Les animations sont sous forme de « bande dessinée », c'est-à-dire que toutes les images de l'animation sont côte à côte dans une seule image, et le défilement géré en interne.

```
Animation* animation_Create(sfImage*, int, int, int, int, int, int, int, float);
void animation_Destroy(Animation*);
void animation_Play(Animation*, int);
void animation_Draw(Animation*, sfRenderWindow*);
void animation_SetPosition(Animation*, int, int);
```

`void animation_Play(Animation*, int)` : Change l'état de l'animation pour qu'elle soit prête à défiler.

`void animation_Draw(Animation*, sfRenderWindow*)` : Dessine l'animation et décale d'une image pour le dessin suivant.

Étant donné que la SFML ne gère pas les animations GIF, nous avons implémenté un système d'animation nous même, comprenant l'animation sous forme de bande dessinée, ainsi que le temps entre chaque image, et la dimension de l'animation.

La map stocke les animations, mais elles ne sont pas dessinées à partir de la map, les animations dessinées sont celles qui sont attachées aux Bullet, Player et Object.

Player :

```
typedef struct PLAYER
{
    sfString *name;                 // Nom du joueur
    char* char_name;                // Nom du joueur en char*
    unsigned int player_id;         // ID joueur
    sfIPAddress* player_ip;         // IP joueur
    unsigned int life;              // Vie restante

    Weapon **weapons;               // Armes du joueur
    unsigned int nb_weapons;        // Nombre d'armes
    unsigned int current_weapon;    // Arme courante

    float coord_x, coord_y;         // Emplacement sur la map
    float m_coord_x, m_coord_y;     // Coordonnées souris

    float speed_x, speed_y;         // Vitesse

    jump_t jump;                    // Type de saut en cours
    bool jetpack_mode;              // Mode JetPack ?

    unsigned int frags;             // Nombre de tués
    unsigned int killed;            // Nombre de morts

    sfSocketTCP* listen_socket;     // Socket d'écoute
    sfBool connected;               // Booléen de présence sur le serveur
    sfBool ready;                   // Prêt à jouer

    Sprite* sprite;

    struct QUAD_TREE* quad_node;    // Noeud du struct QUAD_TREE de la map qui contient
le player
    struct LIST_ELEMENT* list_node;

    bool gravity;

} Player;

Player* player_Create(char*, unsigned int);
void player_Destroy(Player*);
void player_Displace(Player*, Direction, float, Config*);
void player_SwitchWeapon(Player*, int);
void player_CollectWeapon(Player*, int);
void player_WeaponShoot(Map*, Player*, float, float);
void player_SetPosition(Player*, float, float);
void player_Draw(sfRenderWindow*, Player*);
```

`void player_Displace(Player*, Direction, float, Config*)` : Cette fonction déplace le joueur dans la direction donnée, d'une distance calculée à partir de la vitesse par seconde définie dans la structure de configuration, et du temps passé entre les deux images dessinées.

`void player_SwitchWeapon(Player*, int)` : Change l'arme du joueur

`void player_CollectWeapon(Player*, int)` : Ajoute l'arme donnée à l'inventaire du joueur

`void player_WeaponShoot(Map*, Player*, float, float)` : Tire avec l'arme courante et crée des Bullet à gérer.

Cette structure est la structure principale du joueur, elle contient ce qui concerne le joueur : son nom, ses coordonnées, ses coordonnées souris (si c'est le joueur local, le serveur n'a pas ces coordonnées), ses armes, son arme actuelle, sa vie, son IP, son état, son nœud dans l'arbre de collision, etc.

En mode serveur, cette structure n'est que partiellement remplie, étant donné que le serveur n'opère que des vérifications de routine et la transmission de l'état des joueurs.

Bullet :

```
typedef struct BULLET
{
    struct BULLET* prev;            // Liste D-Chain
    struct BULLET* next;

    unsigned int owner;             // Tireur (utile pour le self-damage)
    unsigned int bullet_type;       // Type de balles
    unsigned int damage;            // Dommages infligés
    unsigned int trajectory;        // Type de trajectoire (0 = Rectiligne, 1 =
Parabole (Grenade), 2 = Spread (Shotgun)
    unsigned int range;

    float coord_x;                  // Coordonnées
    float coord_y;

    float speed_x;
    float speed_y;

    Sprite* draw_image;             // Image de la balle (Si Balle invisible, Sprite
Transparent)

    struct QUAD_TREE* quad_node;    // Noeud du struct QUAD_TREE de la map qui contient
le bullet
    struct LIST_ELEMENT* list_node;

    bool gravity;
    float acceleration;

} Bullet;

// Conteneur pour la liste de bullets
typedef struct BULLET_LIST
{
    Bullet* head;
    Bullet* tail;
    unsigned int nb_bullets;
} BulletList;

Bullet* bullet_Create(unsigned int, unsigned int);
void bullet_Destroy(Bullet*);
void bullet_DestroyList(Bullet**);
void bullet_DeleteFromList(Bullet*);
void bullet_SetNext(Bullet*, Bullet*);
void bullet_SetPrev(Bullet*, Bullet*);
Bullet* bullet_GetNext(Bullet*);
Bullet* bullet_GetPrev(Bullet*);
void bullet_Draw(sfRenderWindow*, Bullet*);
void bullet_DrawList(sfRenderWindow*, Bullet*);
void bullet_SetPosition(Bullet*, float, float);
void bullet_SetSpeed(Bullet*, float, float);
BulletList* BulletList_Create();
void BulletList_Destroy(BulletList* ptr);
void BulletList_AddBullet(BulletList* ptr, Bullet* ptr2);
void BulletList_DeleteBullet(BulletList* ptr, Bullet* ptr2);
Bullet* BulletList_GetHead(BulletList* ptr);
Bullet* BulletList_GetTail(BulletList* ptr);
unsigned int BulletList_GetNbBullets(BulletList* ptr);
```

Note : Cette structure a été codée sous forme de liste doublement chainée au lieu d'un tableau pour faciliter la suppression d'éléments dans la liste et éviter la réorganisation inhérente aux tableaux.

`void bullet_Draw(sfRenderWindow*, Bullet*)` : Dessine le bullet envoyé en argument.

`void bullet_DrawList(sfRenderWindow*, Bullet*)` : Dessine une liste de bullets

Object :

```
typedef enum { PLATFORM, PLATFORM_DYNA, TRAP, WEAPON, AMMO } ObjectType;

typedef struct OBJECT
{
    unsigned int objectID;          // ID de l'object (transmission réseau)
    ObjectType type;                // Type d'objet (0 = Plate-forme fixe, 1 = Plate-
forme dyna, 2 = Piège, 3 = Arme, 4 = Ammo)

    Sprite* sprite;                 // Sprite de l'objet

    float start_coord_x;            // Coordonnées de l'objet
    float start_coord_y;

    float curr_coord_x;             // Coordonnées courantes de l'objet (serviront au
dessin et seront mises à jour pour le déplacement)
    float curr_coord_y;

    float dest_coord_x;             // Coordonnées d'arrivée de l'objet (si dynamique)
    float dest_coord_y;

    unsigned int speed;             // Vitesse de mouvement (Plate-forme mobile &
Pièges)
    sfClock* clock_mouvement;       // Clock pour les mouvements.

    unsigned int weapon_id;         // ID de l'arme liée au dessin (si type <= 2)
    unsigned int nb_ammo;           // Nombre de munitions ajoutées par le pack

    sfBool spawned;                 // Affiché ou pas

    bool gravity;                  // Subit la gravité ?

    struct QUAD_TREE* quad_node;    // Noeud du struct QUAD_TREE de la map qui contient
l'objet
    struct LIST_ELEMENT* list_node;

} Object;


Object* object_Create(unsigned int);
void object_Destroy(Object*);
void object_LoadImg(Object*, sfImage*, Animation*);
void object_Draw(sfRenderWindow*, Object*);
void object_SetPosition(Object*, float, float);
```

`void object_LoadImg(Object*, sfImage*, Animation*);` : Associe un sprite à un object

Cette structure est la structure qui sert à gérer les objets du décor : ces coordonnées, son nœud dans l'arbre de collision, ses mouvements éventuels (pour les plate formes mobiles), s'il est associé ou non à une arme (l'objet disparait au contact d'un joueur en lui conférant une arme ou des munitions).

Partie physique

Quad_tree :

```
typedef enum QUAD_POS {NW, NE, SW, SE} Quad_pos;  //position d'un noeuds


typedef struct QUAD_TREE {

    List* bullet;
    List* object;
    List* player;

    sfIntRect rect;     //rect du noeud du quad_tree

    int max_object;     //nombre maximal d'objet que peut contenir le noeud
    int max_depth;      //profondeur maximal du quad_tree
    int depth;          //profondeur actuelle

    struct QUAD_TREE* noeuds[4];    //pointeur vers les 4 fils du noeuds
    struct QUAD_TREE* parent;       //pointeur vers le parent du noeuds
    struct QUAD_TREE* first;        //pointeurs vers le premier element de l'arbre

} QuadTree;

QuadTree* quadtree_Create();
void quadtree_Destroy(QuadTree*);
void quadtree_Generate(QuadTree*, Map*);
void quadtree_Add(QuadTree*, void*, int);
void quadtree_Delete_Node(QuadTree*);
void quadtree_Check_Node(QuadTree*, bool*);
void quadtree_Delete_Elt(void*, int);
void quadtree_Update(void*, int);
void quadtree_Print(QuadTree*);
void quadtree_Draw(sfRenderWindow*, QuadTree*);
```

`void quadtree_Generate(QuadTree*, Map*)` : Permet de créer un QuadTree à partir d'une struct map.

`void quadtree_Add(QuadTree*, void*, int)` : Permet d'ajouter un objet/player/bullet à un QuadTree. Void* est le pointeur vers la struct et int le type de la struct (bullet, object, player).

`void quadtree_Delete_Node(QuadTree*)` : Permet de supprimer un nœud du QuadTree

`void quadtree_Check_Node(QuadTree*, bool*)` : Permet de parcourir le QuadTree à partir d'un nœud donnée afin de savoir si ces noeuds son vide. (Pour savoir si on peut supprimer un nœud ou non).

`void quadtree_Delete_Elt(void*, int)` : Permet de supprimer un élément du QuadTree

Cette structure est l'arbre à collision du jeu, elle permet une détection plus rapide des collisions qu'une recherche qui testerai chaque objet avec tout les autres (on ne cherche des collisions qu'entre objet proche).

Chaque nœud contient une liste des objets qu'il contient et éventuellement 4 fils qui à leur tour peuvent contenir 4 autres fils (d'ou le nom QuadTree) ce qui permet un découpage de l'écran en 4 zone à chaque itération.

Collision :

```
typedef struct COLLISION {

    Physics_type type;  //type de l'objet en collision

    Object* object;
    Player* player;
    Bullet* bullet;

} Collision;

Collision* collision_Create();
void collision_Destroy(Collision*);
Collision* collision_Detection_Object(void*, int);
void collision_Detection_ObjectArb(void*, int, QuadTree*, Collision*);
```

`Collision* collision_Detection_Object(void*, int);` : permet de détecter si un objet est en collision avec un autre en parcourant les listes d'objet contenu dans le nœud associé à l'objet

`void collision_Detection_ObjectArb(void*, int, QuadTree*, Collision*);` : Identique à la précédente mais détecte les collisions en parcourant l'arbre à partir du nœud associé à un objet

Cette structure permet de déterminer si un objet est rentré en collision avec un autre demandé. Elle est renvoyée par les fonctions `collision_Detection_Object` et `collision_Detection_ObjectArb` qui détermine s'il y a collision et renvoient NULL si aucune collision n'est trouvée.

Gravitysystem :

```
void gravitysystem_PlayerUpdate(Map*, Player*, Config*);
void gravitysystem_WorldUpdate(Map*, Config*);
```

`void gravitysystem_PlayerUpdate(Map*, Player*, Config*);` : Met à jour les coordonnées d'un joueur en fonction de la gravité qu'il subit (chute, saut, …).

`void gravitysystem_WorldUpdate(Map*, Config*);` : Met à jour les coordonnées de tout les objets qui subissent la gravité.

Les fonctions de gravité mettent à jour les coordonnées des objets qui subissent la gravité de la map. Elles stoppent la chute en cas de collision, gèrent l'accélération de la chute et servent aussi à gérer le saut des joueurs.

Particle :

```
typedef struct PARTICLE
{
    sfShape* shape;     //forme, couleur, coordonnée de la particule
    float speed_x;      //vitesse
    float speed_y;
}Particle;

Particle* particle_Create();
Particle* particle_CreateBlood();
void particle_SetPosition(Particle*, float, float);
void particle_Draw(sfRenderWindow*, Particle*);
void particle_Destroy(Particle*);
```

`Particle* particle_CreateBlood()` : Permet de créer une particule de sang.

Cette structure sert à créer des particules qui seront gérer par la structure Particle_Table par la suite.

Particle_Table :

```
typedef struct PARTICLE_TABLE
{
    int nbr_particle;       //nombre actuel de particule
    int nbr_max;            //nombre maximum de particule
    int indice_courant;     //indice actuel du tableau
    Particle** particle;    //tableau de particule
} Particle_Table;

Particle_Table* particle_table_Create();
void particle_table_Destroy(Particle_Table*);
void particle_table_AddParticle(Particle_Table*, Particle*);
void particle_table_Draw(sfRenderWindow*, Particle_Table*);
```

`void particle_table_AddParticle(Particle_Table*, Particle*);` : Cette fonction ajoute une particule au tableau de particule de la struct Particle_Table, si la limite de particule est dépassée, elle remplace l'élément du tableau le plus vieux par la nouvelle particule.

Cette structure sert à gérer l'ensemble des particules du jeu et à les afficher, elle possède un maximum de particule qui précisera quand détruire et remplacé les particules les plus vieilles.

Partie Réseau

```
// Permet de regrouper les données client à envoyer au thread
typedef struct CLIENT_DATA
{
    Map* map;

    ChatMessagesList* messages;

    Player* player;
    char* name;
    sfIPAddress ip;

    int port;
    Config* config;

    bool server_close;
} ClientData;

ClientData* clientdata_Create(char*, char*, unsigned int, Config*, unsigned int);
void clientdata_Destroy(ClientData*);
```

Cette structure nous permet d'envoyer au type sfThread de la SFML les données nécessaires pour la fonction client, sfThread_Create (la fonction de création de thread de la SFML) ne prenant que deux arguments, l'adresse de la fonction à exécuter et un pointeur à passer à la fonction à exécuter.

```
typedef enum PACKET_TYPE { CHAT_PACKET, PLAYER_PACKET, OBJECT_PACKET, BULLET_PACKET,
CONNECT_PACKET, ACCEPTED, REFUSED, DISCONNECT_PACKET, SERVER_CLOSING } PacketType;

// Structure stockant le paquet + son type
typedef struct PACKET
{
    PacketType code;
    sfPacket* packet;
} Packet;

// Structure stockant un tableau de paquets
typedef struct PACKET_LIST
{
    Packet** packets;               // Paquets à envoyer
    unsigned int nb_packets;        // Nombre de paquets

} PacketList;
```

Ces deux structures permettent de stocker les paquets de données à envoyer via le réseau avec leurs codes.

Partie Affichage :

Screen :

```
typedef enum { OPT_FONT, GUI_FONT, ALT_GUI_FONT } ScreenFontType;

typedef struct SCREEN
{
    sfImage** base_images;      // Images sources
    unsigned int nb_img;        // Nombre d'images source
    sfSprite** sprites;         // Sprites de l'écran
    unsigned int nb_spr;        // Nombre de sprites

    sfString** texts;           // Textes
    unsigned int nb_text;       // Nombre de textes
    sfFont* opt_font;           // Police des options
    // Intervalle contenant le menu
    unsigned int min_menu;
    unsigned int max_menu;

    Gui* gui;                   // Structure stockant les boutons et textbox
    sfFont* gui_font;           // Police de la GUI
    sfFont* alt_gui_font;       // Police alternative de la GUI

    sfMusic* music;             // Musique
} Screen;

Screen* screen_Create();
void screen_Destroy(Screen*);
void screen_LoadFont(Screen*, ScreenFontType, char*);
void screen_LoadText(Screen*, char*, sfColor, int, sfStringStyle, float, float);
void screen_HighlightText(Screen*, unsigned int, sfColor);
void screen_SetMenuInterval(Screen*, unsigned int, unsigned int);
void screen_LoadMusic(Screen*, char*, sfBool);
void screen_PlayMusic(Screen*);
void screen_StopMusic(Screen*);
void screen_LoadImage(Screen*, char*);
void screen_AddTextbox(Screen*, int, int, int, int, int, sfImage*, sfColor,
Widget_textbox_type, void*, sfColor, char*, sfColor, int);
Gui* screen_GetGUI(Screen*);
Widget_textbox* screen_GetTextbox(Screen*, unsigned int);
void screen_SetActiveTextbox(Screen*, int);
void screen_SetInactiveTextbox(Screen*, int);
void screen_Draw(Screen*, sfRenderWindow*);
```

`void screen_AddTextbox(Screen*, int, int, int, int, int, sfImage*, sfColor, Widget_textbox_type, void*, sfColor, char*, sfColor, int)` : Ajoute une boite de texte à l'interface.

```
void screen_SetActiveTextbox(Screen*, int);
void screen_SetInactiveTextbox(Screen*, int);
```

Rend active/inactive la boite de texte donnée.

V – Améliorations

Pour améliorer le gameplay, on pourrait ajouter un système de pièges aléatoires, pour que le joueur ne puisse pas se reposer sur ses lauriers le long de la partie par exemple, ou d'autres modes de jeu tels « Mort subite », où un tir tuerait le joueur. Si l'on pousse un peu plus sur l'idée de pièges, on pourrait améliorer le moteur physique et changer l'axe sur lequel la gravité s'applique. On pourrait aussi ajouter un système de « mods » pour modifier les statistiques des armes, la vitesse du jeu, la vitesse de déplacement, etc.

D'autres idées seraient l'ajout d'options pour changer la résolution du jeu, les commandes, activer ou non la musique du jeu. Sur le plan graphique, on pourrait améliorer les graphismes (Stickman est peut être un peu basique).

Au niveau du code, de grosses améliorations seraient possibles sur la vitesse de traitement en parallélisant plusieurs fonctions indépendantes telles que la gestion de la gravité, des déplacements, des trajectoires de Bullet, etc. La meilleure amélioration serait de passer au C++, mais ce n'est pas possible avec nos contraintes…

Au niveau sécurité, niveau réseau il n'y en a pas de connues, étant donné qu'on ignore tout paquet non reconnu, et que le reste des failles est inhérent à l'OS. Cependant, il serait possible d'améliorer la partie acquisition de texte lors des discussions, en ajoutant le support des caractères accentués, et en filtrant tout caractère « invisible ».

La création d'un jeu en C pur sans POO et sans fuites étant un challenge, ce projet est la preuve que c'est possible, non sans difficultés qui relèvent de la clarté et de la praticité du code. Les principales difficultés à surmonter étant le réseau, la physique et la gestion mémoire, ce projet a été un petit défi qui, pour nous programmeurs en herbe et expérimentés, a été relevé avec succès !