Rapport de projet « Casse-Briques »

Nicolas Jonville et Victorien Boussuge

Présentation de l'interface

Notre programme est composé d'une fenêtre de jeu et d'un menu. Une partie est automatiquement lancée au démarrage, et l'utilisateur peut donc tout de suite commencer à jouer.

Fenêtre de jeu

Nous avons en haut le nombre balles restantes (3 au maximum), le score et le niveau dans lequel nous sommes.

Chaque niveau est représenté dans la partie centrale, et est composé de murs, d'un palet et de briques. L'organisation des briques est déterminée aléatoirement à chaque niveau en fonction d'une liste de configurations : l'utilisateur peut créer ses propres niveaux en éditant le fichier des niveaux présent dans son répertoire.

Contrôles

L'utilisateur peut contrôler le palet de trois façons différentes : à l'aide du clavier, de sa souris ou d'une webcam (si l'option est cochée dans les paramètres). Appuyer sur la touche espace permet de charger une balle ou d'envoyer une balle déjà chargée. Le joueur peut mettre en pause le jeu en appuyant sur la touche P.

Si le joueur le souhaite, il peut déplacer le palet en filmant les mouvements de sa main avec une webcam. Cette option fonctionne très bien si les mouvements sont lents.

Il est possible de jouer avec plusieurs balles en même temps : les points obtenus en détruisant une brique sont ainsi augmentés.

Menu

Nous avons un menu qui offre quatre propositions : entamer une nouvelle partie, accéder à la fenêtre des joueurs, aller dans les paramètres ou quitter le jeu. Les scores des joueurs sont enregistrés à la fermeture du jeu.

Paramètres

Appuyer sur paramètre nous renvoie sur une boite de dialogue où l'on peut choisir la largeur du palet, l'activation de la caméra ou non. Si cette dernière est activée, la fenêtre principale s'agrandit pour la laisser s'afficher.

Joueurs

Cette fenêtre permet la création, la suppression ou la sélection d'un joueur. Sélectionner un joueur au lancement du programme est nécessaire pour enregistrer les scores.

Cette fenêtre affiche aussi les meilleurs scores de chaque joueur, ainsi que les dix meilleurs scores tous joueurs confondus.

Présentation des classes

Il s'agit ici d'exposer rapidement l'organisation générale du projet, et pour rentrer davantage dans les détails, il suffira de se référer aux fichiers d'en-têtes présents dans ce rapport. Un diagramme de ces classes est présent dans le dossier du projet.

Notre application se décompose en plusieurs classes, que l'on peut classer dans différentes catégories :

La première catégorie de classe permet de faire apparaitre les objets dans OpenGL et de traiter le jeu : « Balle », « Brique », « Palet », « Mur » et « CasseBriques ». On utilise une classe abstraite « Bloc » pour encapsuler certains de ces éléments.

- La classe « CasseBrique » est celle qui hérite de QGLWidget. Elle a pour tâche de créer l'interface du jeu en organisant les éléments entre eux. Pour cela, elle fait appel à ces différents éléments (Balle, Palet, …) et les fait s'afficher (module d'affichage), puis s'occupe de vérifier en permanence s'il y a des interactions entre ceux-ci (module de jeu). Les modules de jeu et d'affichage tournent en parallèle et mettent à jour la partie à chaque débordement de leur compteur respectif. L'organisation des briques est quant à elle déterminée aléatoirement à partir d'un fichier de niveaux que l'utilisateur peut modifier pour créer les siens. Par ailleurs, les contrôles clavier/souris sont implémentés directement dans cette classe.
- La classe « Balle » s'occupe de l'affichage des balles et de leur déplacement. Il est à noter que plus il y a de balles dans l'espace de jeu, plus le score obtenu en touchant une brique est important (il est plus difficile d'évoluer avec trois balles en même temps qu'avec une seule…)
- La classe « Bloc » n'a pas de fonction, mais permet d'organiser de manière cohérente les éléments représentant un bloc, et d'en simplifier la gestion (à l'aide du polymorphisme par exemple). Toutes les classes suivantes de cette catégorie héritent de « Bloc ».
- La classe « Mur » implémente la gestion des murs : la réaction des balles rentrant en collision avec ceux-ci, ainsi que leur affichage. La gestion des collisions est déterminée par le type de mur qui a été spécifié à la création de celui-ci (mur gauche, ou droit, par exemple).
- La classe « Brique » permet l'affichage d'une brique, et sa collision avec une balle. Pour vérifier les collisions on repère si une balle touche un des côtés, ou un des coins, et on s'intéresse aussi à son angle d'arrivée. L'ensemble des briques est stocké dans un std ::vector au sein de la classe CasseBriques.
- La classe « Palet » permet la gestion simplifiée du palet, et garantie le non-dépassement de celui-ci en-dehors de l'espace de jeu. Elle gère au même titre que les autres l'affichage et les collisions. Cependant, contrairement aux autres objets, le palet peut bouger puisqu'il nous est possible de le contrôler. S'il se déplace trop rapidement (ie s'il se déplace trop entre deux rafraîchissements du module de jeu), les collisions peuvent être faussées : la balle se retrouve au milieu du palet par exemple. Cette classe permet donc d'éviter au maximum ce problème, en mémorisant des états de passé puis en les prenant en compte pour calculer la collision actuelle.

La deuxième catégorie regroupe les classes permettant la gestion des données des joueurs : « Joueur », « ListesJoueurs ». Elles garantissent la sauvegarde et la restauration de ces données à l'aide d'un fichier, et s'occupent de les organiser entre elles pour les rendre accessibles depuis d'autres catégories de classes.

- La classe « Joueur » permet la gestion bas niveau des différents joueurs. Elle implémente la syntaxe à utiliser pour lire ou écrire les données d'un joueur dans un fichier. Elle permet de trier les scores et de sauvegarder le nom du joueur associé.
- La classe « ListeJoueur » permet quant à elle d'organiser les données des différents joueurs entre eux à l'aide de la classe « Joueur ». Elle permet de définir un joueur courant, qui indique qui est en train de jouer (et donc à qui attribuer le score), et permet le tri des données de tous les joueurs confondus pour qu'ils soient facilement exploitables pour un affichage par exemple.

La troisième catégorie de classe regroupe les fenêtres : « ParametresDialog », « MainWindow, « JoueursDialog ». Ce sont avec celles-ci que l'utilisateur interagit. Elles permettent l'affichage du jeu, des scores, la modification de paramètres, la gestion des joueurs… Tout ceci de manière directement accessible pour le joueur.

- La classe « ParametresDialog » permet l'ouverture d'une fenêtre qui est accessible via le menu de la fenêtre principale, en choisissant « Paramètres ». La fenêtre affichée agrège pour le moment deux options : le choix de la taille du palet (qui influe sur le score), et l'activation du déplacement du palet avec les mouvements de la main. Si cette dernière option est activée, la fenêtre principale s'agrandit pour laisser apparaître l'aperçu de la webcam ; et lorsqu'elle est décochée l'aperçu disparaît et la fenêtre reprend sa taille initiale.
- La classe « JoueursDialog » qui crée une fenêtre de gestion des joueurs accessible via le menu principal. Elle lit ou agit avec la classe « ListeJoueurs » pour ajouter des joueurs, en supprimer, en sélectionner un (pour y enregistrer ses scores), et pour afficher les meilleurs scores d'un joueur ainsi que de tous les joueurs confondus.
- La classe « MainWindow ». Elle caractérise la fenêtre principale où apparaît un menu et le widget de jeu OpenGL.

Une classe à part permet de gérer les interactions avec la webcam : « Camera ». Elle dépend de MainWindow, mais son affichage est déterminé par la fenêtre des paramètres (ParametresDialog) et elle est utilisée dans le jeu (CasseBriques). Son rôle est d'afficher l'aperçu de la webcam, et de calculer la translation du palet. Pour effectuer ce calcul elle enregistre en mémoire un nombre fixe de translations (les vingt dernières par exemple), qui est remis à jour à chaque nouvelle capture. C'est l'évolution de leur moyenne qui permet un mouvement régulier du palet et un effet de freinage.

Etat de finalisation de l'application

Toutes les fonctionnalités obligatoires et optionnelles du cahier des charges sont implémentées dans notre application :

- Déplacement du palet commandé par le déplacement de la main à partir de la WebCam

- Rebond de la boule sur les murs, sur le palet et sur les briques.
- Destruction des briques (disparition) lorsqu'elles sont touchées par la boule.
- Rebondissement sur le palet et contrôle de la direction de la boule en fonction du point d'impact sur le palet.
- Décompte des boules utilisées et contrôle de la fin de partie.
- Génération aléatoire d'un nouveau niveau avec une vitesse supérieure de la boule par rapport au niveau précédent.
- Calcul des points.
- Choix de la taille du palet par le joueur.
- Sauvegarde du score/nom de joueur et affichage du classement.

Mais nous avons aussi réalisé :

- Gestion des joueurs plus poussée : ajout, suppression et sélection d'un joueur, affichage de ses meilleurs scores ainsi que des meilleurs scores tous joueurs confondus.
- Possibilité de jouer avec plusieurs balles en même temps
- Déplacement du palet avec la souris et le clavier
- Un menu permettant d'accéder aux options et à une fenêtre de gestion des joueurs
- La possibilité de relancer une nouvelle partie en plein jeu
- Un créateur de niveau : le programme lit automatiquement les niveaux créés dans un fichier (pour la syntaxe, cf le Readme.txt)
- La possibilité de mettre pause au jeu
- Un effet de freinage du palet.

Les bogues restants :

- Il arrive de temps en temps que lorsque la balle touche deux briques en même temps, elle soit déviée de la mauvaise façon. Cela est dû au fait que la déviation n'est calculée que sur une des deux briques. Pour remédier à ce bogue, il faudrait faire en sorte que si la balle touche deux briques, alors sa direction doit être modifiée selon X s'il s'agit d'un rebond sur un côté vertical, et selon Y sinon.
- Lorsque l'on augmente la taille du palet, celui-ci peut s'agrandir au-delà du mur. Il faudrait rajouter une condition pour empêcher cela.

Nous avons beaucoup testé notre application, et il ne semble pas y avoir d'autres bogues. Nous avons en effet passé un temps considérable sur la résolution de légers défauts, dans le but d'améliorer l'expérience de jeu.

Fichiers d'entête

Classe « Balle » Boussuge Victorien

```
class Balle
{
public:

    /*******************************/
    /* Constructeurs & Destructeur */
    /*******************************/
```

```
    /* Constructeur pouvant servir au debug (inutile actuellement).
     * Il permet de créer une balle à un endroit précis, et qui se
     * déplace dans une direction précise */
    Balle(float x, float y, float dirX, float dirY);

    /* Constructeur utilisé dans le programme.
     * palet : le palet sur lequel doit se situer la balle au démarrage
     * niveau : niveau en cours pour déterminer la vitesse */
    Balle(Palet *palet, int niveau);

    // Destructeur
    ~Balle();

    /*********************/
    /* Setters & Getters */
    /*********************/

    /* Il y a beaucoup de setters et de getters car cette classe est
     * celle qui est le plus en relation avec les autres classes */

    /* Direction de la balle : toutes les directions sont situées
     * entre 0 et 1. Elles sont déterminées par des cosinus/sinus */
    void setDirection(const float x, const float y)
{m_direction[0]=x;m_direction[1]=y;}
    float getDirectionX() const {return m_direction[0];}
    float getDirectionY() const {return m_direction[1];}

    /* Centre de la balle : utile pour détecter les collisions (get)
     * ou pour déplacer la balle si le palet vient la percuter
     * sur le côté par exemple (set) */
    float getCentreX() const {return m_positionCentre[0];}
    float getCentreY() const {return m_positionCentre[1];}
    void setCentreX(float x) {m_positionCentre[0] = x;}
    void setCentreY(float y) {m_positionCentre[1] = y;}

    // Rayon de la balle : utile pour détecter les collisions
    float getRayon() const {return m_rayon;}

    /* Etat de la balle : est-elle sur le palet ? Sert à savoir
     * si l'on doit déplacer la balle avec le palet par exemple */
    bool getEstSurPalet() const {return m_estSurPalet;}
    void setEstSurPalet(bool etat) {m_estSurPalet = etat;}

    /***************/
    /* Déplacement */
    /***************/

    /* En appelant cette méthode, la balle se déplace automatiquement
     * en fonction de sa direction, qui est mise à jour à chaque
     * collision */
    void deplacer();

    /* Permet d'envoyer la balle lorsqu'elle se situe encore sur le palet,
     * et qu'elle ne s'est donc jamais déplacée */
    void envoyerBalle();

    /* Affichage de la balle : on ne considère ici que l'affichage pur
     * et dur. Cette méthode est appelée par le Display() principal */
    void Display();
```

```
private:

    /*************/
    /* Attributs */
    /*************/

    GLUquadric *m_sphere;
    float m_rayon;
    float m_couleur[3];
    float m_vitesse;
    float m_direction[2];
    float m_positionCentre[2];
    bool m_estSurPalet; // La balle se trouve-t-elle sur le palet ?
};
```

Classe « Bloc » Boussuge Victorien

```
class Bloc
{

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    // Ne sert qu'à fixer la couleur sur blanc
    Bloc();

    /* Le destructeur n'est pas implémenté mais peut être à
     * considérer dans le cas de futures évolutions */
    virtual ~Bloc();

    /*****************************/
    /* Méthodes virtuelles pures */
    /*****************************/

    /* Permet l'affichage de l'objet héritant. Il ne s'agit ici que des
     * fonctions d'OpenGL permettant l'affichage */
    virtual void Display() = 0;

    /* Méthode qui indique si une balle (passée en paramètre) est
     * actuellement en collision avec l'objet. Un booléen est renvoyé
     * comme réponse */
    virtual bool collision(Balle* &balle) = 0;

    /* S'il y a collision (cf la méthode précédente), il s'agit ici de
     * repérer son type, puis de la traiter ! C'est-à-dire qu'en fonction
     * de l'objet héritant (mur, brique ou palet), on va modifier la
     * trajectoire d'une façon particulière. Cette méthode ne se trouve pas
     * dans la classe Balle, justement car on traite les collisions de
     * manière différente en fonction des objets considérés. */
    virtual void traiterCollision(Balle* &balle) = 0;

protected:

    /*************/
    /* Attributs */
    /*************/
```

```
    /* Contient tous les points du bloc, sous la forme
     * x1,y1,x2,y2,x3,y3,x4,y4. */
    float m_points[4][2];

    float m_couleurs[3];
    float m_largeur; // Largeur du bloc, ie son côté le plus grand
    float m_hauteur; // Hauteur du bloc, ie son côté le plus petit

};
```

Classe « Brique » Boussuge Victorien

```
class Brique : public Bloc
{

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* Ce constructeur permet de créer une brique très simplement !
     * On n'a en effet besoin que de la position de son coin, et de
     * sa largeur. */
    Brique(float x, float y, float largeur);

    // Destructeur
    virtual ~Brique();

    /*************************************/
    /* Méthodes virtuelles pures héritées */
    /*************************************/

    /* Pour plus de détails, voir les commentaires du header de la classe
     * Bloc */

    virtual void Display();
    virtual bool collision(Balle* &balle);

    /* Les collisions sont détaillées : on s'intéresse au rebond sur les
     * bords, mais aussi à la réaction de la balle en fonction de son angle
     * d'arrivée sur chaque coin. */
    virtual void traiterCollision(Balle* &balle);

private:

    /*************/
    /* Attributs */
    /*************/

    float m_position[2]; // Position du coin supérieur gauche de la brique

};
```

Classe « Camera » Jonville Nicolas

```
class Camera : public QLabel
{
    Q_OBJECT

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* Pas besoin de passer le jeu (CasseBriques) en paramètre, c'est ce
     * dernier qui récupèrera la valeur de la translation à l'aide d'un
     * get */
    Camera();
    ~Camera();

    /* Permet de créer la caméra, de créer le rectangle dans lequel on
     * récupèrera la translation, et d'initialiser les variables
     * nécessaires au fonctionnement */
    void initialiserCamera();

    /* Méthodes qui permettent de mettre en pause la caméra. La cadence des
     * prises de vues est déterminée par un timer m_timerCam, d'où la
     * nécessité de ces méthodes */
    void stop();
    void start();

    /* Permet d'effectuer une prise de vue, et de calculer la translation
     * de la main. Pour cette dernière, on réeffectue à chaque prise de vue
     * une moyenne des 20 dernières translations calculées (à l'aide d'une
     * std::queue) pour "lisser" le déplacement du palet. Cette moyenne est
     * ensuite stockée et récupérée par le jeu pour faire bouger le
     * palet. */
    void capturerImage();

    /* Fonctions qui permettent de définir ou de récupérer l'état
     * d'activité de la webcam (si elle est cochée dans les paramètres ou
     * pas, par exemple) */
    bool getActive() const {return m_active;}
    void setActive(const bool active);

    /* Récupèrer la translation calculée avec capturerImage(). On divise
     * ici par 20 pour obtenir la moyenne des 20 dernières translations
     * (cf la méthode capturerImage) */
    float getTranslation() const {return m_translation/20.0f;}

private:

    /*************/
    /* Attributs */
    /*************/

    bool m_active; // Indique l'état d'activité de la caméra
    QTimer m_timerCam; // Cadence des prises de vues

    // Stocke les translations consécutives de la MAIN
    std::queue<float> m_buffer;

    // Stocke la translation du PALET
    float m_translation;
```

```
    /* Attributs qui permettent de créer la fenêtre de la webcam, et de
     * dessiner le rectangle vert apparaîssant dessus. */
    cv::VideoCapture cap;
    cv::Mat frame1, frame2, frameRect1, frameRect2;
    cv::Mat resultImage;
    cv::Rect workingRect;
    cv::Rect templateRect;
    cv::Point workingCenter;
};
```

Classe « CasseBriques » Jonville Nicolas

```
class CasseBriques : public QGLWidget
{
    Q_OBJECT

public:

    /*****************************/
    /* Constructeur & Destructeur */
    /*****************************/

    /* Le jeu utilise éventuellement une caméra pour diriger le palet, et
     * attribue des scores aux joueurs. Initialisation des variables d'état
     * et des timers pour le module de jeu et pour le module
     * d'affichage. */
    CasseBriques(Camera* camera, ListeJoueurs *joueurs,
                 QWidget * parent = nullptr);

    // Destructeur
    ~CasseBriques();

    /************************/
    /* Actions sur le palet */
    /************************/

    /* Permet de recupérer ou de fixer la largeur de palet. Ce
     * setter/getter est utile pour permettre à l'utilisateur de fixer
     * lui-même la donnée, via le menu. */
    void setLargeurPalet(float largeur);
    float getLargeurPalet() const {return m_largeurPalet;}

    /* Déplacer le palet sur l'axe des abscisse, en le plaçant au point x.
     * Cette méthode prend en compte la présence du éventuelle balle sur le
     * palet, et la déplace en conséquence. */
    void deplacerPalet(float x);

    /*****************/
    /* Module de jeu */
    /*****************/

    /* Initialisation du jeu, c'est-à-dire des variables qui vont
     * influencer l'expérience de jeu (nombre de balles, réinitialisation
     * des éléments du jeu, ...) */
    void initialiserJeu();

    /* Stoppe la partie. Sert à mettre sur pause par exemple, mais est
     * aussi utile lorsqu'il s'agit de relancer une partie (on arrête le
```

```
     * jeu avant de relancer le jeu) */
    void stopJeu();

    /* Lance la partie. Sert par exemple à sortir du mode de pause, ou bien
     * pour démarrer la partie après avoir initialisé les différents
     * paramètres du jeu */
    void startJeu();

    /* Recommencer une partie depuis le début (utile lorsque l'on clique
     * sur "Nouvelle partie" dans le menu en particulier */
    void nouvellePartie();

private slots:

    /* Mise à jour des variables du jeu. C'est le coeur du module de jeu :
     * cette méthode est appelée à chaque débordement du compteur
     * m_timerJeu. */
    void updateGame();

    /**********************/
    /* Module d'affichage */
    /**********************/

protected:

    /* Fonction d'initialisation de l'affichage, et donc des variables
     * permettant le bon fonctionnement du module d'affichage. Il s'agit de
     * l'équivalent de "initialiserJeu()" pour le module d'affichage. */
    void initializeGL();

    /* Fonction de redimensionnement, qui n'est pas utilisée ici. Nous
     * avons décidé d'attribuer à la fenêtre de jeu une taille fixe. */
    void resizeGL(int width, int height);

    /* Fonction d'affichage qui appelle les fonctions Display() de tous les
     * éléments du jeu, c'est-à-dire les briques, les balles, les murs et
     * le palet. Cette méthode est le cœur du module d'affichage,
     * puisqu'elle est appelée à chaque débordement du compteur
     * m_timerGL */
    void paintGL();

    /****************************/
    /* Gestion des interactions */
    /****************************/

    /* Fonctions de gestion des interactions : elles permettent ainsi à
     * l'utilisateur d'interagir avec le jeu. Par exemple le déplacement du
     * palet, l'envoi des balles, le mode pause, ... */
    void keyPressEvent(QKeyEvent * event);
    void mouseMoveEvent(QMouseEvent *event);

private:

    /********************/
    /* Méthodes privées */
    /********************/

    /* Permet de charger une configuration de briques à partir d'un
     * fichier. Elle est appelée à chaque changement de niveaux. Le fichier
     * utilisé peut être modifié par l'utilisateur pour créer ses propres
     * niveaux (cf Readme.txt pour plus de détails) */
```

```
void chargerNiveau();

/* Traitement des collisions pendant une partie, et fait appel à toutes
 * les méthodes de gestion des collisions des autres classes du jeu. */
void traitementCollisions();

/* Teste si la partie doit continuer ou non. Utile pour vérifier si le
 * jeu est en pause, et si le joueur a gagné ou perdu. */
void testJeuEnCours();

/*************/
/* Attributs */
/*************/

// Liste des joueurs : nécessaire pour rentrer les scores
ListeJoueurs* m_joueurs;

/* Caméra permettant le déplacement du palet à l'aide des mouvements de
 * la main */
Camera* m_camera;

// Timers des modules de jeu et d'affichage.
QTimer m_timerGL;
QTimer m_timerGame;

/* Eléments du jeu (les murs, le palet et les briques héritent de la
 * classe Bloc). On notera qu'on peut utiliser plusieurs balles en même
 * temps. */
Palet *m_palet;
std::vector<Mur *> m_murs;
std::vector<Balle *> m_balles;
std::vector<Brique *> m_briques;

/* Attributs de configuration d'une partie : ces paramètres sont
 * déterminants pour savoir si un joueur a gagné ou perdu (nombre de
 * balle), et permettent la gestion des scores (qui dépend aussi du
 * nombre de balles en jeu) */
unsigned int m_nombreBallesInitial;
unsigned int m_nombreBallesRestantes;
unsigned int m_nombreBallesEnCours;
double m_score;
int m_niveau;

/* Attributs de configuration de l'espace de jeu : ils sont utilisés
 * par le module de jeu pour configurer une partie à partir du fichier
 * des niveaux (celui permettant de créer ses propres
 * configurations) */
int m_briquesParLigne;
int m_briquesParColonne;
float m_largeurBrique;
float m_espaceEntreBriquesLigne;
float m_espaceEntreBriquesColonne;
float m_largeurPalet; // Modifié par l'utilisateur dans le menu

/* Ces attributs sont utiles en interne pour vérifier la condition de
 * certaines réalisations */

/* Indique s'il y a déjà eu une collision sur une brique dans une
 * exécution paintGL. Permet d'éviter un double inversement de
 * direction de la balle. */
bool m_collision;
```

```
    /* Indique si une des balles présentes en jeu est sur le palet (ie n'a
     * pas encore été lancée) */
    bool m_balleSurPalet;

    bool m_perdu; // Indique si le joueur a perdu
    bool m_gagne; //Indique si le joueur a gagné
    bool m_pause; // Le jeu est-il en pause ?

};
```

Classe « Joueur » Jonville Nicolas

```
class Joueur
{

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* On peut choisir de créer un joueur avec ou sans nom ! Ici, il n'y
     * a pas besoin de destructeur. */
    Joueur();
    Joueur(std::string nom);

    /************************/
    /* Gestion des données */
    /************************/

    /* Le chargement des joueurs et de leurs scores se déroule à
     * l'ouverture du jeu, dans le constructeur de la fenêtre principale.
     * On récupère en paramètre l'emplacement du fichier de sauvegarde,
     * déterminé dans MainWindow. */
    void charger(std::ifstream &is);

    /* Le programme sauvegarde les joueurs quand on quitte le jeu. La
     * méthode prend en paramètre un flux d'écriture déterminé dans
     * MainWindow */
    void sauver(std::ofstream &os);

    /*********************/
    /* Setters & Getters */
    /*********************/

    std::string getNom() const {return m_nom;}

    /* Ici on utilise un algorithme de tri par insertion dans un tableau
     * trié dans l'ordre croissant. Cette méthode est appelée à chaque fois
     * qu'on souhaite enregistrer le score d'un joueur (donc à chaque fois
     * que le joueur perd). */
    void setScore(long score);

    /* Renvoie le i-ème score du tableau des meilleurs scores, qui est
     * classé dans l'ordre croissant. Le meilleur score du joueur est donc
     * le dixième élément. */
    long getMeilleursScores(int i) const {return m_meilleursScores[i];}
```

```
private:

    /*************/
    /* Attributs */
    /*************/

    std::string m_nom; // Nom du joueur
    long m_meilleursScores[10]; // Ses dix meilleurs scores

};
```

Classe « JoueursDialog » Jonville Nicolas

```
class JoueursDialog : public QDialog
{
    Q_OBJECT

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* Pour agir sur les joueurs, la fenêtre doit récupérer l'adresse d'une
     * liste de joueurs (ListeJoueurs). Le contructeur crée la fenêtre,
     * remplie le QTableWidget avec le nom des joueurs, et remplie la liste
     * des dix meilleurs scores tous joueurs confondus */
    explicit JoueursDialog(ListeJoueurs* joueurs, QWidget *parent = 0);

    // Destructeur
    ~JoueursDialog();

private slots:

    /****************/
    /* Slots privés */
    /****************/

    /* Permet de mettre à jour l'affichage des meilleurs scores du joueur
     * lorsqu'on le sélectionne dans la liste */
    void selectionChanged();

    /* Gèrent l'ajout et la suppression d'un joueur en intéragissant
     * directement avec la liste des joueurs passée en paramètre */
    void on_boutonSupprimerJoueur_clicked();
    void on_boutonAjouterJoueur_clicked();

    /* Pour enregistrer le score pour un joueur particulier, il faut
     * préalablement le sélectionner, en cliquant sur le bouton
     * "Sélectionner le joueur". Ce slot s'occupe de fixer le joueur
     * sélectionné comme étant le joueur courant, c'est-à-dire celui pour
     * lequel les scores viendront s'ajouter. */
    void on_boutonSelectionnerJoueur_clicked();

private:

    /*************/
    /* Attributs */
    /*************/
```

```
    Ui::JoueursDialog *ui; // Fenêtre en elle-même
    ListeJoueurs *m_joueurs; // Liste des joueurs
};
```

Classe « ListeJoueurs » Jonville Nicolas

```
class ListeJoueurs : public std::list<Joueur>
{

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    ListeJoueurs();
    ~ListeJoueurs();

    /*********************/
    /* Setters & Getters */
    /*********************/

    /* Lorsque l'on sélectionne un joueur sur la fenêtre des joueurs,
     * ces méthodes sont appelées. Si on précise un joueur en paramètre,
     * c'est qu'un joueur a été sélectionné, et si on ne précise rien,
     * c'est qu'on ne définit aucun joueur comme étant le joueur
     * courant. */
    void setJoueurCourant() {m_joueurCourant = NULL;}
    void setJoueurCourant(Joueur &j);

    /* Permet d'attribuer un nouveau score au joueur sélectionné (ie au
     * joueur courant). Le tri des scores de ce joueur est automatique à
     * l'aide de la méthode setScore() de la classe Joueur. */
    void setScore(long score);

    /* Cette méthode prend en paramètre une std::map regroupant des paires
     * P telles que P[i] = (score[i], nomJoueur) pour tous les joueurs. La
     * fonction s'occupe de rentrer dans cet objet tous les scores
     * existant. On utilise une std::map car à l'ajout d'une paire en son
     * sein, la map est automatiquement triée selon une clé (qui est ici le
     * score). L'affichage des scores tous joueurs confondus est donc très
     * simplifié et le tri n'a pas à être implémenté. */
    void getMeilleursScores(std::map<long, std::string> & meilleursScores);

    /* Permet de récupérer le joueur courant. Utile pour récupérer son nom
     * par exemple. */
    Joueur *getJoueurCourant() {return m_joueurCourant;}

    /************************/
    /* Gestion des données */
    /************************/

    /* Ces fonctions permettent de sauver ou charger les données des
     * joueurs (ie noms + scores). Elles font appel aux méthodes charger()
     * et sauver() de la classe Joueur. L'enregistrement et la lecture des
     * scores dans le fichier se fait à l'ouverture et à la fermeture du
     * programme. */
    void charger(std::ifstream &is);
```

```
    void sauver(std::ofstream &os);

private:

    /*************/
    /* Attributs */
    /*************/

    // Le joueur sélectionné, sur lequel enregistrer les scores
    Joueur *m_joueurCourant;

};
```

Classe « MainWindow » Jonville Nicolas

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:

    /*******************************/
    /* Constructeur et Destructeur */
    /*******************************/

    /* Construit la fenêtre principale et la configure en insérant le
     * widget OpenGL et le menu. */
    explicit MainWindow(QWidget *parent = nullptr);

    // Destructeur
    ~MainWindow();

    /***************************/
    /* Slots privés et publics */
    /***************************/

public slots:

    /* Est appelé par la fenêtre des Paramètres lorsque l'on coche "Activer
     * la caméra". Cette méthode permet d'agrandir la fenêtre principale
     * pour laisser apparaître la webcam juste à droite du widget OpenGL.
     * Elle permet aussi de diminuer la fenêtre à nouveau lorsque l'on
     * désactive l'option Webcam */
    void updateWidgetCentral();

private slots:

    /* Commencer une nouvelle partie quand nous appuyons sur l'option dans
     * le menu. */
    void slotNouvellePartie();

    // Quitter la partie quand nous appuyons sur l'option dans le menu
    void slotQuitter();

    // Afficher la fenêtre Paramètres en appuyant dans le menu
    void slotParametres();

    void slotJoueurs(); // Renvoie au menu des joueurs
```

```
private:

    /*************/
    /* Attributs */
    /*************/

    /* Les trois attributs suivants constituent le jeu en lui-même : des
     * joueurs, un casse-briques, et un contrôle (la caméra). */
    CasseBriques* m_casseBriques;
    Camera* m_camera;
    ListeJoueurs* m_joueurs;

    /* Ces deux attributs permettent de gérer l'organisation de l'espace
     * central de la fenêtre principale, qui peut contenir éventuellement
     * l'aperçu de la webcam en plus du casse-briques. */
    QHBoxLayout* m_layoutCentral;
    QWidget* m_widgetCentral;
};
```

Classe « Mur » Boussuge Victorien

```
class Mur : public Bloc
{

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* Le constructeur prend en paramètres les coordonnées des coins du
     * mur, le type de mur (gauche, haut, droite) ainsi que les coordonnées
     * limites pour la balle (celle qu'elle ne doit pas dépasser). */
    Mur(float p[][2], int type, float coord);

    // Destructeur
    virtual ~Mur();

    /*************************************/
    /* Méthodes virtuelles pures héritées */
    /*************************************/

    /* Pour plus de détails, voir les commentaires du header de la classe
     * Bloc */

    virtual void Display();
    virtual bool collision(Balle* &balle);
    virtual void traiterCollision(Balle* &balle);

private:

    /*************/
    /* Attributs */
    /*************/

    /* Représente le type de mur : 1 pour le mur de gauche, 2 pour
     * le mur de droite, et 3 pour le mur du haut. Cela est utile pour
     * gérer les collisions. */
    int m_type;
```

```
    // Valeur de x ou y que la balle ne doit pas dépasser
    float m_coord;

};
```

Classe « Palet » Boussuge Victorien

```
class Palet : public Bloc
{

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* Nous rentrons les valeurs du milieu du palet x et y, la largeur du
     * palet, sa hauteur, et les valeurs en abscisse xMin et xMax que le
     * palet ne doit pas dépasser (il ne doit pas traverser les murs) */
    Palet(float x, float y, float largeur, float hauteur,
          float xMin, float xMax);

    // Destructeur
    virtual ~Palet();

    /*************************************/
    /* Méthodes virtuelles pures héritées */
    /*************************************/

    /* Pour plus de détails, voir les commentaires du header de la classe
     * Bloc */
    virtual void Display();
    virtual bool collision(Balle* &balle);

    /* Le traitement des collisions est amélioré pour le palet, car celui
     * ci n'est plus fixe. Si le palet se déplace rapidement, la balle peut
     * se retrouver au-milieu de celui-ci entre deux mises à jour du module
     * de jeu, et le traitement des collisions peut alors être faussé. Pour
     * remédier à cela on se réfère à l'emplacement de la dernière
     * collision pour réajuster la position de la balle. On peut ainsi
     * déplacer le palet plus rapidement. Lorsque le palet percute la balle
     * sur le côté et continue d'avancer dans sa direction, celui-ci la
     * déplace. */
    virtual void traiterCollision(Balle* &balle);

    /**********************/
    /* Setters & Getters */
    /**********************/

    // Coordonnées du centre du palet
    float getCentreX() const {return m_position[0];}
    float getCentreY() const {return m_position[1];}
    void setCentreX(float positionCentreX);

    // Hauteur et largeur
    float getHauteur() const {return m_hauteur;}
    float getLargeur() const {return m_largeur;}
    void setLargeur(float largeur);
```

```
    // Vitesse du palet lors du déplacement (utile pour le clavier)
    float getVitesse() const {return m_vitesse;}

    /* Angle minimal de rebond quand la balle atterie sur un côté extrême
     * du palet */
    float getAngleMin() const {return m_angleMin;}

    /**********************/
    /* Mouvement du palet */
    /**********************/

    /* Permet de déplacer le palet en prenant en compte les murs. On
     * appelle cette fonction via la fonction deplacerPalet() du
     * CasseBriques à chaque fois qu'on utilise le clavier, la caméra ou la
     * souris pour déplacer le palet */
    void decaler(const float x, const float y);

private:

    /*************/
    /* Attributs */
    /*************/

    float m_vitesse; // Vitesse de la balle
    float m_position[2]; // Position du centre
    float m_xMin; // xMax que le palet ne doit pas dépasser
    float m_xMax; // xMin que le palet ne doit pas dépasser

    // L'angle minimal dont peut dévier la balle en touchant le palet
    float m_angleMin;

    /* Prise en compte de la dernière collision. 1 : à gauche du palet, 2
     * en haut du palet, 3 : à droite du palet, 0 : pas de collision */
    int m_codeDerniereCollision;

};
```

Classe « ParametresDialog » Jonville Nicolas

```
class ParametresDialog : public QDialog
{
    Q_OBJECT

public:

    /*******************************/
    /* Constructeur et destructeur */
    /*******************************/

    /* La fenêtre des paramètres doit récupérer des instances de Camera et
     * CasseBriques pour modifier des paramètres étant rattachés à ces
     * classes. */
    explicit ParametresDialog(CasseBriques* casseBriques, Camera* camera,
                              QWidget *parent = 0);

    // Destructeur
    ~ParametresDialog();
```

```
    /****************/
    /* Slots privés */
    /****************/

private slots:

    // Slot pour modifier la valeur de la largeur du palet
    void slotLargeurPalet();

    void slotActiverCamera(); // Activer ou non la caméra

    /************/
    /* Signaux */
    /************/

signals:

    /* Ce signal est appelé lorsque l'on coche ou décoche "Activer la
     * caméra". Celui-ci indique qu'il faut modifier la fenêtre principale
     * pour inclure l'aperçu de la webcam, ou bien le supprimer. */
    void updateWidgetCentral();

private:

    /*************/
    /* Attributs */
    /*************/

    Ui::ParametresDialog *ui; // Interface graphique

    // Utilisé pour modifier la longueur du palet
    CasseBriques* m_casseBriques;

    Camera* m_camera; // Utiliser ou non la caméra

};
```