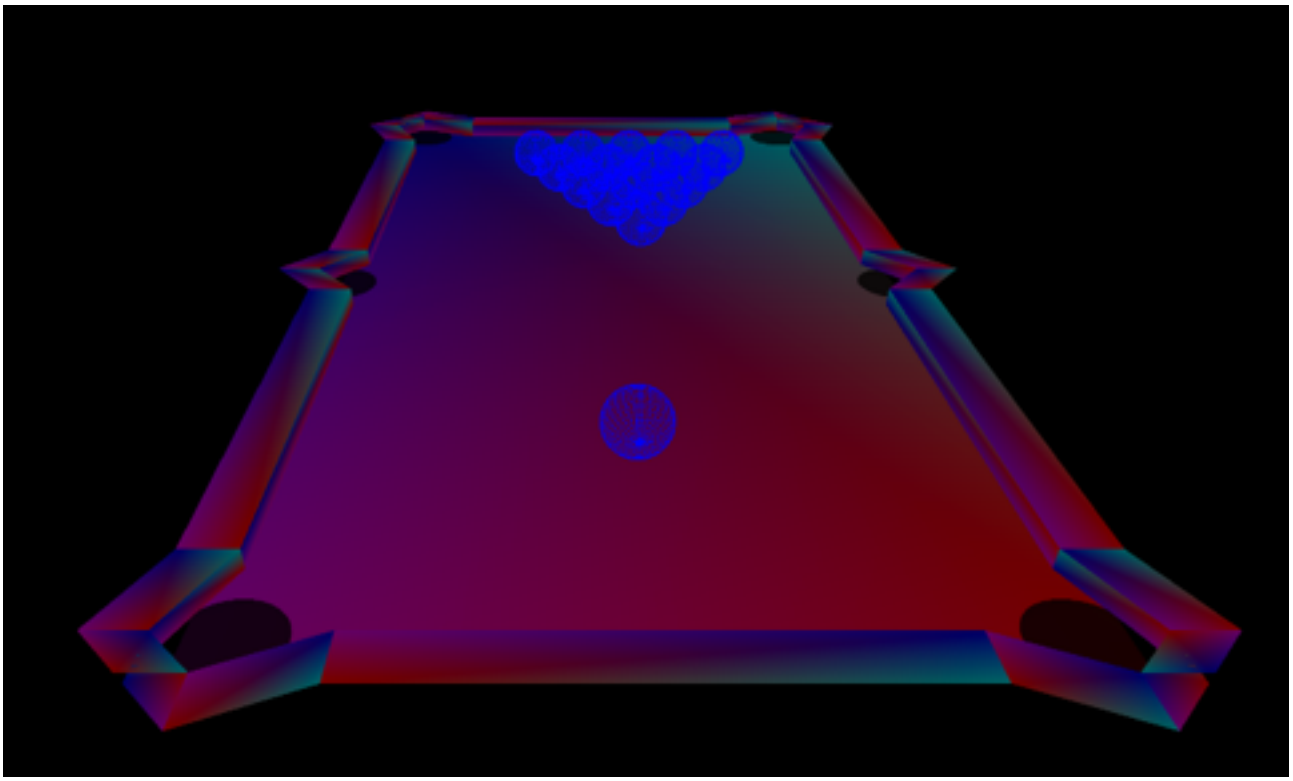
Conception Orientée Objet
Billard en C++

Harbulot Julien | **Snijders Zoé** | Mai 2015



SOMMAIRE

Les intégrateurs

Pour pouvoir faire évoluer nos objets physiques (les faire avancer, recalculer la vitesse, etc.) nous utilisons un intégrateur.

Les objets intégrables

Afin d'isoler la classe en charge de l'intégration des différentes spécificités de nos objets, nous avons créé une interface `Integrable`, qui propose les méthodes suivantes :

```
class Integrable{
public:
    virtual Vecteur etat() const = 0;
    virtual Vecteur derivee_etat() const = 0;
    virtual Vecteur derivee_seconde_etat(double temps) const = 0;

    virtual void set_etat(Vecteur const& nouveau) = 0;
    virtual void set_derivee_etat(Vecteur const& nouveau) = 0;

    virtual ~Integrable(){}
};
```

Les intégrateurs

Un intégrateur reçoit un `Integrable` et s'occupe de mettre à jour son état. En pratique, il recalcule la vitesse et la position en fonction de l'accélération de l'`Integrable`.

Afin de pouvoir utiliser différentes méthodes d'intégration de façon polymorphique, nous avons décidé de créer une interface `Integrateur` dont hériteront les différentes implémentations.

En héritent les classes suivantes : `Integrateur_Euler`, `Integrateur_Newmark` et `Integrateur_RungeKutta4`. Ces classes utilisent les méthodes d'intégration dont elles portent le nom.

Conclusion

Grâce au duo `Integrateur` / `Integrable`, ces classes pourront être réutilisées dans tout projet nécessitant un moteur physique. De plus, l'application est isolée du choix de la méthode d'intégration grâce à l'interface `Integrable`.

Les Objets physiques

Notre billard sera constitué de différents objets : des boules, des parois, des briques, des trous, etc. Afin d'isoler le contrôleur (i.e. la boucle d'événements) de ces différents objets, nous avons créé une interface `Objet`.

L'interface Objet

L'interface `Objet` offre des méthodes liées aux attributs physiques (vitesse, position, etc), ainsi que la méthode `point_le_plus_proche_de(Vecteur const& point)`, qui, étant donné un point extérieur, renvoie le point de l'objet le plus proche, ainsi que le vecteur normal au plan tangent à la surface de l'objet en ce point. Cette méthode sera utile pour traiter certaines collisions (la majorité !) de façon polymorphique.

Afin de pouvoir gérer correctement les collisions les plus spécifiques, nous avons décidé d'implémenter un pattern visiteur sur la hiérarchie des `Objet`. La classe `Objet` propose donc une méthode `void accept(ObjetVisiteur& v)`. Ce port d'entrée pour le dispatch visiteur nous sera également utile pour afficher les `Objet` sans avoir à surcharger les classes dérivées.

Enfin, un objet possède une vie. Dans le monde réel, cette vie pourrait correspondre à son état (neuf / cassé). Nous avons donc ajouté une méthode `bool en_vie() const` à la classe `Objet`. Dans ce projet, nous utiliserons la vie pour savoir quand un objet est tombé dans trou. Nous ajouterons également une `BouleTueuse` qui détruit les objets avec lesquels elle entre en collision.

Un Objet est Integrable

Un `Objet` possède une vitesse, une vitesse angulaire, une accélération et une accélération angulaire. Il s'agit de vecteurs à trois dimensions que la classe Objet concatène en vecteurs d'état (à 6 dimensions) utilisables par un intégrateur.

Cette opération permet d'isoler les classes filles de leur caractère intégrable : nous manipulons des vecteurs classiques à 3 dimensions, et non des vecteurs à 6 dimensions.

L'interface UniqId

Nous avons décidé qu'un objet possèderait un identifiant unique fixé à la compilation. Nous avons donc créé une classe `UniqId` munie d'une méthode `int uniq_id() const`. Les objets héritent de cette interface.

Cet identifiant unique n'est jamais utilisé dans le code de l'application, mais nous nous en servons pour identifier les objets plus simplement que par leur adresse lors du débogage. *Elle pourra donc être supprimée sans conséquence sur le comportement du moteur physique*.

Première couche d'implémentation

La classe `Objet` a été pensée comme une interface afin d'isoler les applications qui en dépendent des détails de son implémentation. Néanmoins les classes dérivées partagent beaucoup de code (position, vitesse, etc.) que nous avons décidé de factoriser dans deux classes : Les `ObjetMobile`s et les `ObjetImmobile`s.

Un `ObjetMobile` possède des attributs liés à son état physique : position, vitesse, accélération, etc.

Un `ObstacleImmobile` ne possède pas de vitesse et pas d'accélération.

Seconde couche d'implémentation

Voici les classes qui implémentent les spécificités de chaque objet :

La classe `Boule` représente une boule : elle dispose d'un rayon, peut calculer la vitesse relative en un point de sa surface et implémente la méthode `point_le_plus_proche_de(Vecteur const& point)`.

La classe `Paroi` représente un morceau de plan contre lequel les objets peuvent rebondir.
La classe `Brique` représente un parallélépipède. En interne elle est constituée de six parois.
La classe `Trou` représente un trou. Concrètement, il s'agit d'un disque plan pour lequel nous définirons des règles de collision particulières.

Les Boules spéciales

Nous avons créé des sous classes de la classe `Boule` afin d'enrichir les fonctionnalités de notre application : `BouleDeCouleur` et `BouleTueuse`. Nous parlons enfin d'une classe utilitaire importante : la télécommande.

Boule de couleur

La `BouleDeCouleur` possède un attribut supplémentaire du type `Couleur*`. Ce type n'est pas défini dans le coeur du billard afin que chaque interface graphique puisse le re-définir selon ses besoins.

Boule tueuse

La `BouleTueuse` détruit les boules avec lesquelles elle entre en contact. Pour cela, il a suffi de traiter une collision avec une boule tueuse comme une collision avec un trou.

Boule invincible

Il s'agit d'une boule qui change de couleur et qui ne meurt jamais.

Télécommande pour Boule

Enfin, signalons ici que nous avons implémenté dans la partie graphique une télécommande pour `Boule`. Il s'agit d'une classe capable d'intercepter les événements clavier et qui ajuste la vitesse de la boule télécommandée en conséquence.

L'utilisation de cette télécommande ne requiert aucune modification dans le coeur du billard (le moteur physique) et permet de créer un vrai jeu : l'utilisateur peut contrôler une boule, la faire se déplacer, sauter, etc.

Les collisions

La détection des collisions

Pour détecter les collisions, nous utilisons une classe `DetecteurDeCollision`. Cette classe utilise un `Integrateur` et détecte la prochaine collision à venir entre deux objets.

Cette collision à venir est retournée via une structure `Collision` qui contient un booléen permettant d'indiquer si une date à été trouvée, ainsi qu'un double correspondant à la date trouvée.

Comme les collisions `Boule` / `Boule` et les collisions `Boule` / `ObstacleImmobile` sont détectées différemment, nous utilisons le port d'entrée du *dispatch visiteur* grâce à la bibliothèque *Multiple Disptach Wrapper* que nous avons implémentée (voir chapitre dédié).

Pour utiliser la classe de détection, il faut appeler la méthode : `Collision detecter(Objet&, Objet&)`

Grâce à la bibliothèque, l'appel sera redirigé vers la méthode appropriée parmi les suivantes :

```
Collision operator() (Boule&, Boule&)
Collision operator() (Boule&, ObstacleImmobile&)
Collision operator() (ObstacleImmobile&, Boule&)
```

Remarque sur l'algorithme de détection des collisions

Nous avons apporté un soin particulier à la détection des collisions afin de ne pas détecter de collisions inexistantes : nous avons effectué des tests de validité réguliers.

La gestion des collisions

La classe `ExecuteurDeCollison` de gestion des collisions fonctionne de façon similaire à la classe de détection des collisions. Le port d'entrée est : `bool executer(Objet&, Objet&)`, et les méthodes de gestion sont :

```
bool operator() (Boule&, Boule&);
bool operator() (Boule&, ObstacleImmobile&);
bool operator() (Boule&, Sol   &);
bool operator() (Boule&, Trou   &);
bool operator() (Boule&, BouleMortelle&);
bool operator() (BouleMortelle& a, BouleMortelle& b);
```

Le booléen qui est retourné indique si la collision a pu être gérée. Par exemple, il est négatif lorsque les objets ne sont pas en contact, ou lorsqu'ils s'écartent déjà l'un de l'autre.

Le moteur physique : la classe Billard

La classe billard porte en elle l'état du monde physique. Afin d'isoler l'interface graphique (`Viewer`) et le contrôleur de ses spécificités, nous avons créé une interface.

L'interface Billard

```
class Billard {
public:
    virtual void evoluer(double dt) = 0;
    virtual void se_dessiner(Viewer& viewer) = 0;
};
```

Concrètement, voici un exemple de boucle d'évènement simple (nous n'utilisons pas cette boucle-ci dans l'application finale) :

```
inline void
controler(Billard& b, Viewer& v, double temps_a_faire, double dt)
{
    for(double temps_fait = 0; temps_fait <= temps_a_faire; temps_fait += dt)
    {
        b.evoluer(dt);
        b.se_dessiner(v);
    }
}
```

La classe BillardGeneral

Nous avons décidé de créer une classe `BillardGeneral` capable de représenter l'état de n'importe quel billard.

Cette classe possède une collection polymorphique de pointeurs intelligents uniques vers des `Objet`s, ainsi qu'une collection de pointeurs nus vers des `Boule`s.

Cette deuxième collection pourrait être une collection de pointeurs vers des `ObjetMobile`s mais nous n'avons pas jugé cette abstraction nécessaire *pour l'instant* du fait que les `Boule`s sont les seuls objets mobiles dont nous disposons.

Notre application a été conçue de sorte à ce que le billard puisse manipuler une collection d'`Objet`s sans avoir à distinguer entre les objets mobiles et les objets immobiles.
Pour cela, il faut encore ajouter un élément que nous n'avons pas eu le temps d'implémenter : le calcul de la force exercée par un `Objet` générique sur un autre. Nous l'avons fait dans le cas particulier du `Sol` et des `Boule`s : c'est pourquoi nous utilisons une collection de `Boule*`.

La responsabilité de la classe `BillardGeneral` est la suivante : faire avancer le temps pour les objets contenus. Il possède donc les deux méthodes suivantes qui sont `protected` et non virtuelles :

```
protected:
    void faire_avancer_le_temps(double dt_integrateur);
    void supprimer_les_objets_morts();
```

ainsi que la méthode publique virtuelle pure suivante :

```
public:
```

```
virtual void evoluer(double dt) = 0;
```

Bien entendu, ce ne sont pas les seules méthodes dont la classe `BillardGeneral` dispose, mais nous les indiquons ici car la signification est claire : les classes filles se chargent de l'algorithme d'évolution (détection des collisions / gestion des collisions) mais pas des détails liés à la gestion du temps. C'est la classe `BillardGeneral` qui intègre les objets et qui gère la mise à jour de son âge interne.

Remarque sur l'intégrateur

Comme nous utilisons un intégrateur via un pointeur vers l'interface `Integrateur`, il est possible de changer de méthode d'intégration pendant l'exécution de l'application. Pour cela, il suffit d'appeler le setter suivant : `void integrateur(unique_ptr<Integrateur> nouveau)`. Nous n'utilisons pas de référence afin d'indiquer que l'intégrateur fourni change de propriétaire et appartient au `BillardGeneral`.

L'implémentation de l'algorithme d'évolution

Nous avons créé les deux classes filles suivantes en charge d'implémenter la méthode `evoluer(double dt)` : `BillardAlgo1` et `BillardAlgo2`.

Le premier algorithme est classique et peu intéressant : nous ne gérons qu'une collision par date. Voici la partie intéressante de l'algorithme tel que nous l'avons écrit :

```
ProchaineCollision prochaine = prochaine_collision_avant_fin_du(petit_dt);

if (not prochaine.trouvee) {
	faire_avancer_le_temps(petit_dt);
}
else{
	faire_avancer_le_temps(petit_dt);
	gerer_collision(  move(prochaine)  );
	double petit_dt_restant = petit_dt - prochaine.dt_avant_collision;
	evoluer(petit_dt_restant);
}
```

Le second algorithme est plus évolué : d'une part nous gérons toutes les collisions qui ont lieu à la même date et d'autre part nous gérons les faux positifs : s'il n'y a pas vraiment eu collision, nous exécutons la collision suivante.

Voici la partie intéressante de l'algorithme telle qu'elle est écrite dans le code (pour réduire la taille du code dans ce fichier pdf, nous avons omis la majorité des commentaires et la gestion des erreurs) :

```
auto prochaines = prochaines_collisions_avant_fin_du(petit_dt);
    // Nous attendons un tableau trié par dates croissantes

    if (prochaines.empty())
    {
       faire_avancer_le_temps_securise(petit_dt);
    }
    else{
        double petit_dt_deja_parcouru = 0;
        bool au_moins_une_collision_valide = false;

        // La boucle suivante cherche à diminuer le petit dt restant
        // Tant que nous n'avons pas géré de vraie collision
        for(auto prochaine : prochaines)
        {
            double dt_avant_collision = prochaine.first - petit_dt_deja_parcouru;
            auto const& objets_en_collision = prochaine.second;
            faire_avancer_le_temps_securise(dt_avant_collision);
            petit_dt_deja_parcouru += dt_avant_collision;
            bool collision_valide = gerer_collision(objets_en_collision);

            if(collision_valide)
            {
                au_moins_une_collision_valide = true;
                break; //fin de la boucle car nous avons trouvé une vraie collision
            }
        }//for

        double petit_dt_restant = petit_dt - petit_dt_deja_parcouru;

        // Si nous n'avons pas réussi à diminuer le dt_restant
        // C'est qu'il existe des faux positifs dans dt_avant_collision = 0
        // Dans ce cas, nous intégrons.
        if( not au_moins_une_collision_valide and almost_equal(petit_dt_deja_parcouru, 0., 10) ){
            faire_avancer_le_temps_securise(petit_dt_restant);
        }
        else{
            evoluer(petit_dt_restant);
        }
    }
```

Conclusion

1. L'interface `Billard` permet d'isoler l'interface graphique et le contrôleur (i.e. la boucle d'évènement) des détails d'implémentation du billard.

2. Les pointeurs intelligents s'occupent de gérer la mémoire des objets.

3. La classe `BillardGeneral` s'occupe de gérer le temps et l'intégration.

4. Les classes filles implémentent différents algorithmes de détection et de gestion des collisions pour la méthode `evoluer(double dt)`.

L'interface graphique

L'interface Viewer

Comme à notre habitude (cf sections précédentes) nous avons isolé les détails d'implémentation des `Viewer` du reste de l'application grâce à une interface. Cette interface permet d'utiliser différents `Viewer` avec un même billard : Open GL (`OpenglViewer`), console texte (`TextViewer`), etc.

Voici son prototype :

```
class Viewer {
public:
    virtual void dessiner(Objet &) = 0;
};
```

Concernant son implémentation, nous avons décidé d'isoler les classes filles du mécanisme de dispatch que nous utilisons : une interface graphique ne devrait pas avoir à se servir d'une bibliothèque pour pouvoir dessiner l'état de notre billard.

Ainsi, nous avons créé une classe `ObjetViewer` qui implémente la méthode `void dessiner(Objet&)` et qui effectue le dispatch en utilisant la bibliothèque *Multiple Dispatch Wrapper* que nous avons créée, comme le font les classes de gestion des collisions.

```
class ObjectViewer : public Viewer{
public:
    void dessiner(Objet&) override final;

    virtual void operator()(Boule &) = 0;
    virtual void operator()(Paroi&) = 0;
    virtual void operator()(Sol   &) = 0;
    virtual void operator()(Brique&) = 0;
    virtual void operator()(Trou&) = 0;
    virtual void operator()(BouleDeCouleur&);
    virtual void operator()(BouleMortelle&);
};
```

Les implémentations des interfaces graphiques spécifiques à une technologie (Qt, OpenGl, etc.) peuvent donc se dispenser d'inclure la bibliothèque puisqu'il leur suffit d'hériter de la classe `ObjetViewer`.

Le cas du Viewer Qt/OpenGL

Nous avons réalisé une partie graphique qui utilise la bibliothèque Qt ainsi que la technologie OpenGL pour dessiner le billard en trois dimensions.

Nous avons identifié et distingué trois responsabilités :
1. la mise en place de la fenêtre,
2. le dessin de formes (carré, cercle, etc.) avec OpenGL,
3. et la représentation des objets du billard.

Pour chacune de ces responsabilités, nous avons créé une classe :
1. la classe `GLWidget` fournit un environnement OpenGL (la fenêtre),
2. la classe `VueOpenGl` s'occupe de dessiner des formes et propose entre autres les méthodes suivantes : `dessineSegment(…)`, `dessineCercle(…)`, `dessineSphere(…)`,
3. la classe `OpenGlViewer`, qui hérite de `ObjetViewer` et qui utilise la classe `VueOpenGl` afin de dessiner les formes spécifiques des objets du billard.

Une telle séparation présente de nombreux avantages. Par exemple, **il sera aisé de réutiliser la classe `VueOpenGL` dans n'importe qu'elle application graphique** puisque cette classe ne dépend pas du billard, de ses spécificités et de ses objets.

La fenêtre, promue contrôleur de l'application

Nous n'avons pas jugé utile de créer une classe contrôleur dédiée.
Dans notre application, c'est la fenêtre GLWidget qui tient ce rôle : elle contient une boucle d'évènements (via un Timer Qt) et s'occupe de faire évoluer le billard.

Elle intercepte de plus les événements clavier liés au déplacement de la caméra et met à jour la vue OpenGL en accord avec les commandes reçues.

Les bibliothèques

Lors de l'implémentation de notre application, nous avons jugé utile d'extraire certaines fonctionnalités dans des classes et des fichiers dédiés que nous avons placés dans le dossier `libraries/`.

Par exemple, la détection des collisions `Boule / Boule` requiert la résolution d'un trinôme du second degré : nous avons extrait les formules nécessaires à cette résolution dans des fonctions dédiées et nous avons créé le fichier `trinome_second_degre.h`.

Afin de clairement distinguer les bibliothèques que nous avons conçues des bibliothèques externes que nous utilisons, nous avons créé un sous dossier `3rd_party/` (parties tierces). Dans ce dossier figure la bibliothèque *Bandit Cpp* qui est une bibliothèque de tests unitaires écrite par Joakim Karlsson (https://github.com/joakimkarlsson/bandit) et mise à disposition sous les termes de la *MIT license*. Cette bibliothèque est présentée à l'adresse suivante : http://banditcpp.org/.

Ainsi, toutes les bibliothèques qui ne sont pas dans ce sous-dossier **sont de notre conception** :
- *cpp_sscanf* (utilisée dans le fichier de tests des affichages texte)
- *String Matcher* (utilisée dans le fichier de tests des affichages texte)
- *Multiple Dispatch Wrapper* (utilisée pour nos dispatch visiteur : classes de collision et viewer)
- raii.cpp

Ces trois bibliothèques sont sous licence libre *MIT License*.

Ci dessous, nous expliquons brièvement le fonctionnement de la bibliothèque *Multiple Dispatch Wrapper* sur laquelle repose une grande partie notre projet.

La bibliothèque *Multiple Dispatch Wrapper*

Cette bibliothèque, *que nous avons réalisée*, est disponible sur BitBucket (https://bitbucket.org/Gauss_/library-multiple-dispatch-in-cpp) et est brièvement présentée à l'adresse suivante : https://julienharbulot.wordpress.com/2014/10/09/multiple-dispatch-library/.

Dans cette partie nous expliquons comment son utilisation permet de simplifier le code de notre projet.

Notre projet repose sur la mise en oeuvre d'un dispatch visiteur sur la hiérarchie `Objet`.

Ce dispatch est utilisé à deux endroits :
- pour la gestion des collisions (dispatch double),
- pour l'affichage des objets (dispatch simple).

La bibliothèque factorise le code des visiteurs et nous permet d'effectuer ce dispatch sans avoir besoin d'hériter de l'interface `ObjetVisiteur`. Nous écrivons donc moins de lignes de code.

Mais l'utilisation de cette bibliothèque nous permet d'utiliser le port d'entrée du dispatch visiteur pour effectuer simplement un double dispatch : à partir de deux références vers des `Objet`, nous pouvons retrouver simplement deux références vers des classes filles. C'est ce mécanisme que nous utilisons dans nos classes de collision.

Sans l'utilisation de la bibliothèque, la mise en place de ce double dispatch aurait été fastidieuse et aurait exigé la pollution des classes filles qui héritent de `Objet` par des méthodes de visite. En effet, dans un double disptach classique, chaque objet est à la fois un *visiteur* et un *visitable*, ce que l'utilisation des wrappers permet d'éviter.