Symkit, du projet

Ancarola Raffaele et Cincotti Armando

28 mai 2018

Table des matières

Résumé

Dans le cadre du cours d'informatique de la section de Physique, pour l'année 2018 on se trouve à devoir développer un programme permettant de simuler numeriquement, et visualiser graphiquement, un sistème de plusieurs oscillateur harmoniques et d'oscillateurs couplés. Le document qui suit sert à illustrer la conceptualisation de notre projet couche par couche, jusqu'à l'illustration complète de ce qu'est notre programme à l'heure actuelle et de ses fonctionnalités. Dans l'idéation du programme, tout en suivant les cosignes principales, nous avons essayé de garder nos idées originales, pour rendre notre travail plus personnel possible, en deviant parfois dès consignes secondaires données par le professeurs tout le long de l'année pour nous guider dans la création du programme. Dans la suite, on essayera d'expliquer au mieux quand est-ce que notre programme devie dès consignes données, et pourquoi. Deux sont les buts que nous nous sommes donnés dans le developpement du projet : Le principal étant la bonne réusite de notre travail dans le cadre du cours, et donc dans le but d'obtenir une bonne évaluation. Le deuxième, pas moins important pour nous, étant la création d'un util numérique pour simulations physiques plus générales, étant donné qu'un tel programme pourrait nous être très utile tout au long de notre formation de physiciens.

Le projet principal, qu'on a appelé Symkit (pour Symulation Kit), est composé de quatre sous-parties, qui gèrent les aspects purement matématiques, graphiques ou du plotting de graphes liés aux systèmes physiques simulés, et finalement l'implementation d'une interface utilisateur. Symkit compte en plus deux subdirs Alpha et Beta, qui font le lien entres les trois partie du programme. Ils contiennent en faite le main qui crée et lance les simulations numeriques et leurs raprèsentations graphiques. Alpha comprenant nos premiers tests sur le complexe du programme, et Beta representant le résultat final après toute correction et amélioration.

Chapitre 1

Symath

1.1 Vectors

Liste des classes :
— **Vector**
— **SVector**<std::size_t N>

1.1.1 Vector

Cette classe raprèsente un vecteur à dimension quelquonque, utile pour la gestion d'un ou plusieurs paramètre. La dimension dépend souvent du nombre de degrès de liberté de l'oscillateur où le vecteur est utilisé. Par exemple pour un pendule qui bouje le long d'un plan fixe dont on peut modéliser le mouvement en fonction de la seule variation d'un angle, on utilisera un Vector de dimension 1 ayant en cordonnée l'angle que le pendule crée avec la verticale.

Pour cette classe sont definis :
— Un constructeur par defaut l'initialisant à la valeur 0, un **Vector** par defaut sera donc de dimension 1 ceci parce que les oscillateurs harmoniques décrits ont souvent un seul degré de liberté.
— Des constructeur pour initialiser un **Vector** de plusieurs manières (par vector, initializer list etc.).
— Des utils de looping pour traverser les differents paramètre (utilisés parfois dans les autres fonctions implementées).
— Fonctions pour la modification de la *dimension du Vector*.
— Plusieurs operateurs dont :
  — Un accesseur aux paramètre stockés, *operator*[], permettant aussi de les modifiers s'il le faut.
  — Des operateurs implementant somme interne et produit externe canoniques d'un espace vectoriel R(n). Ces ci sont construit pour permettre la concatenation d'operations sur des Vector.
  — Operateur implementant produit scalaire et vectoriel (si la dimension des Vector est compatible, sinon on lance un erreur).

  - Des operateurs de comparaison d'ordre, qui effectue la comparaison sur le module du/des Vector.
  - Des operateurs de comparaison de similitude, pour savoir si deux Vector sont égaux ou non.
  - Un operateur "-" retournant l'inverse additif du Vector, et "~" l'inversant.
- Deux fonction retournant le module du vecteur ou son carré (pour éviter une operation de plus).
- Une fonction returnant le vecteur unitaire du Vector concerné. Cette fonction est très utile pour caractériser des axes de restriction de mouvement. En effet, pour evaluer la projection d'un vecteur sur un certain axe de mouvement, pour simplifier les calculs il suffit de calculer le produit scalaire du vecteur pour le vecteur unitaire décrivant l'axe.

1.1.2 SVector

Celle-ci est une classe qui n'a pas été demandé par le prof, mais qu'on a crée pour distinguer les vecteurs pour la gestion de paramètres numériques caractérisant les systèmes physiques simulés, dès vecteurs utilisés pour placer des objet dans un environnement graphique tridimensionnel. **SVector** est une *classe template*, pour qu'on puisse la definir pour différentes valeurs de T. T representant une dimension fixe pour un SVector. Cette classe décrit donc des vecteurs à dimension fixe, et elle implemente essensiellement les même fonctions présentes dans la classe Vector sauf pour celles qui en modifient la dimension. En effet, des vecteurs destinés à "se déplacer" dans un espace à dimension finie fixe, n'auront jamais à modifier leur dimesion. Cette classe a été créée pour être utilisé dans la partie du programme qui gère la graphique. Des **SVector<3>** (de dimension 3) sont donc utilisés plus loin dans le programme pour décrire par exemple la position de caméra et objets graphiques.
Pour les deux types de Vecteurs a été défini en fin, un overload de l'*operator«* pour qu'on puisse afficher facilement la liste de valeurs d'un vecteur. Cet operateur est utilisé par exemple pour afficher les valeurs de position et vitesse d'un objet physique en évolution, pour obtenir ainsi une simulation en mode texte.
N.B. Un objet graphique est placé dans l'environnement 3D en fonction d'un *SVector<3>*, mais l'objet physique lié à un certain objet graphique, est décrit au travers d'un *Vector* contenant sa vitesse et un Vector contenant sa position, lesquels peuvent évoluer en fonction d'équations spécifiés plus loins dans le programme. Il y pourtant l'ecception des objets physiques décrit par des **NewtonDescriptors** (voir paragraphe suivant), lequels sont décrit au travers de *SVectors<3>*.

1.2 Descriptors

Un **Descriptor** est un objet qui permet de décrire un comportement évolutif dans le temps pour un objet physique quelquonque. La classe **Descriptor**

est une classe abstraite, contenant en effet une unique méthode virtuelle pure *update()*, utilisée pour "mettre à jour" l'objet, en le faisant évoluer en fonction de son type (oscillateur ou masse ponctuelle newtonienne pour ce qui a été fait à l'heure actuelle). Cette méthode est donc redefinie dans les classes héritant Desciptor.

Voici des specifications pre-définies de cette classe :

— **NewtonDescriptor** : décrit un objet dans un éspace tridimensionnel

— **Oscillateur** : décrit l'évolution d'un oscillateur

1.2.1 Oscillateur

Un **Oscillateur** est un descriptor décrivant le mouvement d'un oscillateur harmonique ou d'un oscillateur couplé. Celle-ci est une classe abstraite. Attributs :

— Deux Vectors contenant *positions* et *vitesses* des masses composantes l'oscillateur décrit. Ceux-ci peuvent être modifiés pour que l'objet décrit puisse évoluer.

— Un **integral_operation** qui est un type enum (EULER_CROMER ou NEWMARK) qui sert à spécifier au travers de quel intrgrateur la méthode *update()* mettra à jours les valeurs de position et vitesses de l'oscillateur.

Méthodes :

— **equation()**, une méthode virtuelle pure qui défini, dans chaque sous-classe d'oscillateur, l'équation décrivant le mouvement de l'oscillateur.

— **update()**, redefinie ici pour prendre en argument un pas de temps et modifier avec un certain *integrateur*, les valeurs de vitesse et position de l'osccilateur. Ceci est fait en fonction de la methode *equation()*. Le comment la methode equation est utilisée est spécifié lors de l'explication des integrateurs.

— Des constructeurs appropriés, exploitant chaqu'une manière différente de construir un Vector.

— Accesseurs et Manipulateurs pour accèder ou modifier les attributs de la classe.

1.2.2 OscillateurSimple

Sous-classe abstraite d'Oscillateur décrivant les oscillateurs comportant une seule masse. Cette sous-classe ajoute à la classe Oscillateur une méthode virtuelle pure **cartesiennes()**. Cette dernière a été implementée pour convertir la position de la masse décrite en un *SVector<3>*. En utilisant des cordonnées cartesiennes dans l'environnement graphique, on en a besoin pour qu'on puisse utiliser un vecteur position compatible avec l'implementation graphique. Vu qu'un **OscillateurSimple** décrit le mouvement d'une seule masse, ça vaut la peine, d'après nous, d'accèder ainsi à la position de l'objet.

1.2.3 NewtonDescriptor

Un **NewtonDescriptor** est un descriptor décrivant l'évolution d'une masse pontuelle sujette à des forces, dans un espace euclidian suivant les principes de la mécanique newtonienne. Attributs :

— Masse de l'objet.
— Trois SVector<3>, contenant *position*, *vitesse* et *forces* appliquée à l'objet.

Méthodes :

— Constructeurs appropriés et un destructeur.
— **update()**, qui modifie position et vitesse de l'objets par l'integrateur d'*Euler Cromer*, en foncion du vecteur force resultant appliquée à l'objet et en fonction de sa masse.

Il nous parait important de spécifier que la classe **NetonDescriptor** n'as pas étée demandée pour le projet, et que restent des corrections à faire vu qu'on s'est intéressés plus à ce qui était démandé, pour bien réussir dans notre évaluation finale. On a quand même crée un tel objet dans le main pour en illustrer l'idée d'implementation possible. Pour un **Oscillateur** a été implementé un overload de l'*operator«*, permettant d'imprimer sur un ostream comme cout, les vecteurs position et vitesse. Cet affichage, à chaque pas de temps, sera l'implentation pour une simulation en mode texte.

1.3 Integrateurs

— Vector **integrateEulerCromer**<typename>
— Vector **integrateNewmark**<typename, class Owner>

Celle des integrateurs est une des implementations qui a le plus dévié par rapport aux indications donnés. Les indications voulaient qu'on crée une classe integrateur qui traite la méthode d'integration d'un oscillateur différemment en fonction du type d'integrateur pris, et que ce soit l'integrateur à prendre en attribut un oscillateur pour le modifier. Dans notre conception, à chaque méthode d'integration correspond une *fonction* qui prend en paramètre(par référence), un vecteur position et vitesse et en modifie les valeurs en fonction d'un pas de temps et d'un vecteur acceleration. Ce dernier correspond à la valeur de retour de la fonction **equation()** pour les oscillateurs, et au vecteur forces pour un NewtonDescriptor. C'est donc, le **Desciptor** qui appele la méthode d'intégration choisi au travers de la méthode **update()**, en modifiant ainsi ses propres attributs de vitesse et position. Les méthodes d'integration sont insérées dans **integrator.h** (et .tpp), dans le **namespace symkit**, ainsi les **Descriptors** y ont accès au travers de ces deux ligne de code : *"#include skerror.h" "using namespace symkit"* (pour faciliter l'appel des méthodes). De plus les méthodes sont définies pour un type **template**, pour qu'elle puissent fonctionner pour des *Vectors*, si les appelle un Oscillateur, ou aussi pour des *SVectors*, si à les appeler est un NewtonDescriptor. Grace à cette implementation, on évite de créer une classe de plus à chaque fois qu'on veut implementer une nouvelle méthode

d'integration.

1.4 Symkit Error, géstion des érreures

SKerror est une structure contenant données, informations et commentaires utiles pour gérer un erreur pendant l'éxecution du programme. Se trouvant dans **namespace symkit**, à l'interieur de **skerror.h**, toutes les classes peuvent y avoir accès, et l'utiliser facilement à travers les commande *"#include "skerror.h""* et *"using namespace symkit"*. Lorsque, dans une certaine fonction le code est suceptible de générer une erreur, un **SKerror** est lancé, et au moment du catch un message d'erreur est affiché sur l'ostream cerr, grace à l'overload de l'*operator«* pour les **SKerror**. Eventuellement le programme s'arrête si l'erreur est fatale. Les informations contenues dans un SKerror sont :

— Type d'erreur (de type enum).
— Nom de la fonction qu'a généré l'erreur.
— Nom de la classe qui contient cette fonction.
— Fatalité ou non de l'erreur (de type bool).

Chapitre 2

Symgraph

2.1 Scene

La classe **Scene** ouvre une fenêtre *OpenGL* et manage le monde de dessin 3D. Elle implémente un mécanisme qui enregistre les composantes de type **SKActor** et les distingue parmi leur rôle (évoluable, dessinable, etc). Les rôles seront spécifiés dans les prochaines section. C'est dans cette classe qu'on gére le *fonctionnement/evolution* des objets graphiques, ajoutés à la scène, en temps de compilation, et en runtime, grace au système des **listeners** qui traduisent des input clavier (et eventuellement souris, à implementer), en commandes de gestion en temps réel. La scene peut hériter d'une classe listener, si elle gère les input d'un certain type. **Scene** implémente une fonction pour la gestion de la runtime (voir **Scene : :timerEvent** en **symgraph/scene.cpp**) laquelle définie l'ordre de mise à jour des éléments travaillant dans et sur la scène, ordre à respecter à chaque pas de temps, en tenant compte des input externes, si il y en a. Le *pas de temps* est propre à la scène, comme si les objets se trouvaient dans un espace avec un temps rélatif identique pour toutes les particules (ce qui est sensé d'après nous). **Scene** est strictement liée à une classe **Camera** pour la gestion de la vision du monde 3D de symulation. **Scene** gère aussi la simulation mode texte : si l'on veut, en appuyant sur un numero il est possible d'afficher les valeurs d'évolution d'une ou plusiseures particules. Ainsi faisant la simulation mode texte devient optionnelle, mais elle est de toute façon implementée.

Cette classe est celle qui correspond, à peu-prè, au support à dessind demandé par les consignes du projets. Pourtant nous l'avons implementée comme étant une scène, prenant des objets graphiques, bien différents des Descriptors mais liées à ceux-la pour évoluer dans la scène. Scene est un **SceneWrapper** (Classe décrite plus loin) donc elle contient plusieurs objets graphiques, des actors, qu'elle fait évoluer et qu'elle dessine chaqu'un de sa manière grace au mécanisme de résolution dynamique des liens. Donc, comme pour un support à dessin elle gère le dessin, mais la méthode de déssin reste propre aux objets déssinnés

2.2 Camera

Camera est une classe qui contient les infomations et les methodes pour la gestion d'une camera virtuelle dans le monde graphique 3D. La classe **Scene** peut facilement obtenir de cette classe une *QMatrix4X4* pour la création de la scène 3D au travers de la fonction **getMatrix()** contenue dans Camera.

2.3 Specifics (Specs)

Les superclasses qui déterminent l'identité d'un objet destiné à être enrégistré dans **Scene** sont à la base du système polimorphique de *Symgraph*. En faite, chaque classe joue un rôle différent, par une ou deux méthodes, dans le déroulement d'un *frame* pendant la simulation. Voici les classes en ordre d'appel :

1. **Evolvable** : Classe des objets graphiques pouvant *évoluer*, elle contient une fonction d'évolution *evolve(float)*.
2. **Describable** : Classe des objet pouvant *évoluer* selon un *Desciptor*, elle contient donc une reference au descriptor du quel cette classe dépend.
3. **Positionnable** : Classe des objets peuvant être *positionnés*.
4. **Orietable** : Classe permettant d'appliquer une *orientation* à l'objet.
5. **Scalable** : Classe permettant de changer la *grandeur* des objets graphiques.
6. **Colorable** : Classe permettant de changer la *couleur* des objets graphiques.
7. **Renderable** : Classe des objets *déssinables*.

De plus, pour permettre de pouvoir abiliter/désabiliter ces fonctions pour une certaine classe qui hérite ces caractéristiques, dans le fichier *specs.h* est définit un système à *bitfield* qui manage cette option.
N.B. Ces Specifics s'appliquent à des objets graphiques, comme par exemples des particules (voir plus bas), on ne déssinne donc pas dans des oscillateur dans la scène, mais plutôt des particules liées aux oscillateurs, étant ces derniers des objets physiques décrivant, dans notre conception, l'évolution d'une particule ou d'un système de particules.

2.3.1 Quelques mots de plus sur Specs.h

Si on regarde le contenu du fichier, on voie qu'il n'y a que de définitions de macro. En particulier, la classe **SKActor** construit son mécanisme de management des *flags* par l'utilisation de cette dernières.

L'idée qui est derrière est celle, pour chache acteur, de pouvoir manager ce qui est active e ce qui ne l'est pas en n'utilisant qu'un *byte* de mémoire. Puisque les identités specs sont exactement 8, alors on a exactement 1 bit réservé à chaque identité, pour à la fin composer un nombre de type *unsigned char*, c'est à dire un byte. La définition des bits resérvés est la suivante :

```
/* 8-bit type definition for shape flags */
typedef unsigned char specs_t;

#define S_EVOLVE      0x1   // first bit   00000001
#define S_RENDER      0x2   // second bit  00000010
#define S_FILL_MODE   0x4   // third bit   00000100
#define S_POSITION    0x8   // fourth bit  00001000
#define S_ORIENTATION 0x10  // fifth bit   00010000
#define S_SCALING     0x20  // sixth bit   00100000
#define S_COLOR       0x40  // seventh bit 01000000
#define S_DESCRIPTOR  0x80  // eighth bit  10000000
```

Pour comprendre comment ce système est interfacé dans les acteurs, il faut d'abord considérer que la variable qui contient les flags est définite à l'interieure de la classe **SKActor**, elle a un nom fixe et son accéssibilité est *protected*. Pour *nom fixe*, on entend qu'il y a aussi une macro qui définit son nom. On va voir que les définitions de specs.h ne se limitent pas à la déclaration de la *bit map* ou de la variable de stockage, mais aussi de l'accès à ce mécanisme pour garantir de l'agréabilité en l'utilisant.

On peut, à ce point là, regarder la définition de **SKActor** :

```
    /*
     * Generate specs accessor methods
     *
     * These methods allow to
     * access the SPECS_VAR by a "set" and a "get" method
     *
     * In particular, the set method allows to set all flags in one
     * using the numerical macros defined in "specs.h"
     * For example:
     *
     * setSpecs(S_EVOLVE | S_RENDER | S_POSITION | S_COLOR);
     */
    SPECS_ACCESS

protected:

    /*
     * Generate the specs flags field
     *
     * This macro is expanded to
     *
     * specs_t SPECS_VAR;
     *
     * This field must be protected in order to pass
     * the flags bit-wise structure to the children classes,
     * so they can generate the accessors methods defined
     * in "specs.h"
```

```
28      */
29     SPECS_FIELD
```

La macro **SPECS_FIELD** s'expande directement dans la définition de la variable de stockage de la bit map, où son nom est forcement donnés par **SPECS_VAR** :

```
1  /*
2   * define the variable to be added on the private or
3   * protected side of a class declaration
4  */
5  #define SPECS_VAR specs_flags
6
7  /*
8   * Field to be added inside the class in order to make
9   * work all specs generated methods
10  */
11 #define SPECS_FIELD specs_t SPECS_VAR;
```

D'autre côté, la macro **SPECS_ACCESS** engendre les accésseurs minimalistes à cette variable, qui permettent la convérsion entre des booleans et des *specs_t* et qui permettent aussi de l'affecter entièrement.

```
1  /*
2   * Accessors to set the specs flags in one time
3   */
4  #define SPECS_ACCESS                                    \
5                                                          \
6      void setSpecs(specs_t value)                        \
7      {                                                   \
8          SPECS_VAR = value;                              \
9      }                                                   \
10                                                         \
11     specs_t getSpecs() const                            \
12     {                                                   \
13         return SPECS_VAR;                               \
14     }                                                   \
15                                                         \
16     void toggleSpecs(specs_t value)                     \
17     {                                                   \
18         SPECS_VAR ^= value;                             \
19     }                                                   \
20                                                         \
21     void setSpecsState(specs_t value, bool state)       \
22     {                                                   \
23         if (state)                                      \
24             SPECS_VAR |= value;                         \
25         else                                            \
26             SPECS_VAR &= ~value;                        \
```

```
27      }
```

Ici, on voie la raison pour laquelle le nom de la variable de stockage est fixe : pour que elle soit utilisable dans les autres macro et les implémentations se réfèrent toujours à elle, son nom doit forcement être constant. Même si les méthodes générés par les macro specs marchent sous ces conditions strictes, l'utilisateur ne va jamais percevoir des difficulté. En fait, ce mécanisme est strictement limité à la costruction de la classe **SKActor** et on va voir comment peut-on en beneficier par héritage.

Application à une classe Specs

Potentiellement, toute classe peut hériter d'une classe specs, mais, par construction, chaque classe specs ne possède aucune géstion de l'activation ou de la déactivation sa propre fonctionnalité. Prendons par exemple **Evolvable** :

```
1  class Evolvable
2  {
3  public:
4
5      virtual ~Evolvable() {}
6
7      virtual void evolve(float dt) = 0;
8
9      virtual bool isEvolving() const = 0;
10 };
```

La peculiarité de cette classe est de posseder la méthode *evolve*, qui peut être définite dans plusieurs façons. Ce qui nous intéresse est le fait que, quand la scene essaye de faire appèl à cette méthode, elle execute d'abord un control supplémentaire ; c'est ici que la méthode *isEvolving* est appellée et elle retourne le status d'actif ou inactif. Mais, en fait, cette méthode est virtuelle pure et il faut toujours le définir dans les sous-classes ; c'est ici qui le mécanisme de management des specs flags entre en jeu. Quand un acteur hérite une classe specs comme **Evolvable**, on ajoute aussi (dans la séction *public*) la macro **SPECS_EVOLVE**.

```
1  class AnActor : public SKActor, public Evolvable
2  {
3      /* Update from Evolvable */
4      virtual void update(float) override;
5
6      /* Methodes pour acceder et modifier la flag de S_EVOLVE */
7      SPECS_EVOLVE
8  }
```

Cette macro engendre les méthodes manipulateurs de la variable de stockage specs qui a été héritée par **SKActor**. En fait ces manipulateurs agissent direc-

tement sur la variable, qui est d'accès *protected*, il est donc nécéssaire hériter la classe SKActor pour pouvoir utiliser ce mécanisme correctement. Compris entre ces méthodes, il y a la définition de *isEvolving*. Le rôle de cette macro, est donc aussi celui de définir la méthode virtuelle qui avait été déclarée dans la superclasse **Evolvable**.

On peut, toute de suite, regarder la définition des manipulateurs :

```
/*
 * Define abstractly the methods to be generated into
 * the public side a class declaration
 */
#define SPECS_METHODS(specs_Name, specs_bit)      \
                                                  \
    virtual bool is##specs_Name() const override  \
    {                                             \
        return (SPECS_VAR & specs_bit) != 0;      \
    }                                             \
                                                  \
    void toggle##specs_Name()                     \
    {                                             \
        SPECS_VAR ^= specs_bit;                   \
    }                                             \
                                                  \
    void set##specs_Name(bool value)              \
    {                                             \
        if (value)                                \
            SPECS_VAR |= specs_bit;               \
        else                                      \
            SPECS_VAR &= ~specs_bit;              \
    }
```

Qui va se concrétiser dans les *SPECS* macro :

```
/*
 * Define all generation methods
 */

// expand this macro into a class inheriting Evolvable
#define SPECS_EVOLVE SPECS_METHODS(Evolving, S_EVOLVE)

// expand this macro into a class inheriting Renderable
#define SPECS_RENDER SPECS_METHODS(Rendering, S_RENDER)

// expand this macro into a class inheriting Renderable
#define SPECS_FILL_MODE SPECS_METHODS(FillMode, S_FILL_MODE)

// expand this macro into a class inheriting Positionable
#define SPECS_POSITION SPECS_METHODS(Positioning, S_POSITION)

```

```
17 // expand this macro into a class inheriting Orientable
18 #define SPECS_ORIENTATION SPECS_METHODS(Orientating, S_ORIENTATION)
19
20 // expand this macro into a class inheriting Scalable
21 #define SPECS_SCALING SPECS_METHODS(Scaling, S_SCALING)
22
23 // expand this macro into a class inheriting Colorable
24 #define SPECS_COLOR SPECS_METHODS(Coloring, S_COLOR)
25
26 // expand this macro into a class inheriting Descriptable
27 #define SPECS_DESCRIPTOR SPECS_METHODS(RunningDescriptor, S_DESCRIPTOR)
```

2.4 SceneWrapper

SceneWrapper est la super classe d'un objet de classe **Scene**. Cette classe contient 4 liste de pointeurs à 4 types d'objets graphiques differents :

- **SkActor**
- **Evolvable**
- **Describable**
- **Renderable**

Ainsi une instance de scene pourra travailler sur ces 4 types différents d'objets spécifiquement. **SceneWrapper** hérite directement de **Evolvable** et de **Renderable**, donc elle contient une méthode *evolve()* et *render()*, qui consiste à faire évoluer et déssinner les *renderables* et *dessinnables* qu'elle gère, simplement grace au mécanisme de la résolution dynamique des liens, en appelant *evolve()* et *render()* pour chaqu'un des objets. Il est vrai que des objets contenus comme les evolvables et describalbe peuvent être les deux en même temps, mais en enregistrant leurs adresses dans des listes différentes lorsqu'on veut dessinner tout les dessinnables, ou utilisers tout les actors, ou plus, il suffit d'appeler la fonction pour les membre d'une des liste, ainsi on gagne en complexité temporelle car pour des objet qui sont forcement tous spécialisés dans l'action qu'on demande, il n'y a pas besoin de faire tout un tas de contrôles néccessaires autrement.

2.5 Symkit Actors

La classe **SKActor** est la classe base de tout ce qu'on peut enregistrer dans une **Scene**. Chaque instance de cette classe contiendra une *Qstring* fonctionnant de ID, et un *bitfield* permettant d'activer ou desactiver, à l'initialisation de l'acteur, ses spécificités (comme décrit dans *specs.h*). La classe se refère à *specs.h* pour obtenir les variable et méthodes nécessaires pour le fonctionnement à bitfield du système de specifics (voir commentaires dans *specs.h*).
Les sous-classes définies dans symgraph se divisent dans les suivantes catégories :

- Formes (Shape)
  - Décorations

  - Particules
- Systèmes (ParticleSystem)
  - Systèmes interactifs
  - SySystèmes d'oscillateurs

2.5.1 Formes et décorations

La classe **Shape** hérite **SKActor**, **Renderable** et **Colorable**. Son but est de dessiner un *modèle graphique*. Précisement elle contient une référence à un *modèle graphique* qui va être utilisée pendant l'ésecution de *render*. Il faut donc toujours donner une référence d'un modèle à une instance de *Shape*, sinon il va rien dessiner.

En particulier, cette classe est à la base de **Decoration**, dont l'objectif est de dessiner un modèle en donnant une position, une orientation et une rapport de volume dans l'espace tri-dimensionnel, sans avoir aucun comportement evolutif. L'autre spécialization pré-définie de **Shape** concèrne les formes qui vont avoir un comportement évolutive et qui sont aussi déstinées à simuler de la physique, on parle de la classe **Particle**.

2.5.2 Particules et oscillateurs

Maintenant, on arrive à interfacer la physique au graphisme. En faite, la classe fille de **Shape** qui s'occupe de cet but est la classe **Particle**, qui n'est rien d'autre qu'une forme qui hérite **Descriptor** et **Positionable**. Noter que dans cette classe rien est encore défini parce qu'elle est abstraite. Les deux spécifications de cette classe sont **NewtonParticle** et **OscillatorParticle**. La différence principale entre les deux est le type de **Descriptor** qu'elles utilisent, en fait, la première contient une **NewtonDescriptor** et la deuxième un **Oscillateur**. Les deux définissent les méthodes virtuelles de la superclasse en fonction de la nature du **Descriptor** et, de plus, elles ajoutent des accesseurs et des modificateurs selon le besoin. En particulier les deux classes contiennent une méthode *position()* qui permet d'obtenir la position évoluante pour dessinner le model dans le monde graphique. On a fournit de plus à la classe **OscillatorParticle**, un offset et des méthodes permettant de deplacer l'origine autour de la quelle la particule évolue.

2.5.3 Systèmes de particules

Pour ce type d'objet existent dans notre programme plusieures implementations, dont certaines implementées pour être utilisées en dehors du cadre du projets, et pour élargir le domaine de simulations possibles de notre programme.

Particlesystem

ParticleSystem est une classe qui gère plusieurs **Particles** permettant de les regrouper dans un domaine. Elle possède donc une liste de pointeurs à des

particules, et des méthodes pour en ajouter ou en enlever. Cette classe hérite de **Evolvable** et **Renderable** et définit donc les méthodes pour faire évoluer et déssinner les particules stockés dans cette classe. Un **ParticleSystem** possède aussi un offset, et hérite de **Positionnable** la méthode *position()* qui retourne ce dernier paramètre, pour qu'on puisse déssinner chaque particule en fonction de la position du système. Chaque particule évoluera autour de l'offset du système.

InteractingSystem

InteractingSystem hérite de **ParticleSystem** et spécifie comment les particules du système interagissent entre elles à deux à deux par la méthode *interaction(Particle *, Particle *)*. Cette classe a été conçue pour faire évoluer des Particules de type **NewtonParticle**, elle sort donc du cadre du projet, mais elle nous sera utile en futur, pour permettre de developper des interaction physiques plus générales entre particules. Elle redefinie la méthode *evolve()* pour considérer l'interaction entre particules.

Système Pendule-Couplé

Une classe **ActorGroup** hérite directement de **SceneWrapper** et sert comme classe de base pour un système de type *pendule-couplé*. Un tel système est representé par la classe **SymSystem**, qui hérite entre autre de **Describable** et **Positionable** (pour implementer le même système d'offset que le Particlesystem). Il prend comme descriptor des *Descriptor* pour oscillateurs couplés, pour obtenir de ce dernier les informations pour dessinner et faire évoluer deux particules. Dans le subdir *Beta*, pour finir, pour chaque sous-classe de **Symsystem** sera définie une *Shape positionnable*, qui prend en attribut un numero d'identification (0 ou 1 pour deux oscillateurs couplés entre eux) et qui retourne, par la méthode *position()* la position de la shape en fonction de son id.

Pour simplifier la compréhénsion du fonctionnement de la classe, de suite un exemple détaillé est illustré.
Ex.
Dans *Beta* existe la classe *PendCouple*, qui hérite de *SymSystem*, et *PenduleDesc* qui est un Descriptor de type *Oscillateur* décrivant l'évolution d'une couple de pendules. On defini maintenant une Shape *PendSphere*, qui hérite de *Positionnable*. PendSphere prend en attribut un *pointeur à un PenduleDesc*, et un int constant fonctionnant de numero d'*identification*. Si le id est 0, la méthode *position()* retourne la position de la première particule décrite par PenduleDesc, sinon elle retourne la position de la deuxième particule décrite. *PendCouple* prend en attribut un *pointeur à PenduleDesc*, et *deux pointeurs à deux PendSphere* initialisées avec 0 et 1 comme id. PendCouple initialise les PendSphere auquel ses pointeurs pointent, en leur donnant en réference PenduleDesc, ainsi il pourra les dessinner chacune en fonction de leur position.

2.6 Modèles graphiques

On a dit plus haut qu'une forme prends une référence à un modèle de dessin et pendant l'appel de *render* elle le dessine. Précisement, derrière à la concéption de modèle, il y a les appèls à la *OpenGL*.
La classe **SKModel** est définie par un *Vertex Buffer Object* et un *Index Buffer Object*. Les deux sont des conténiteurs de donnés statiquement insérés, qui une fois passés à la mémoire, ils restent en modalité de lecture jusqu'à quand on ait dit de les libérer. Ce mécanisme permet d'optimiser les appèls du procésseur en gagnant aussi beaucoup sur le temps de dessin.
La *Vertex Buffer Object*, en particulier, contient les coordonnés des vertex en format de groups de trois *float*, le *Index Buffer Object* contient les références en format de nombres entiérs de où trouver les vertex et avec quel ordre les positionner.

2.7 Clavier et souris

KeyListener et **MouseListener** sont des classes abstraites décrivant des objets déstinés à récolter des input, respectivement venant du clavier et de la sourie, pour éxecuter une certaine commande à chaque input prédéfini. Dans une sous-classe de *Scene*, qui hérite **KeyListener** ou **MouseListener**, on définit en effet les commandes à executer sous l'influence d'un certain nombre d'input. Un *Listener* contient pour finir une méthode pour être activé/desactivé, et une méthodes pour controller s'il est activé ou pas.

Chapitre 3

Symplot

3.1 Données

3.1.1 Plot_data

Pour representer des données caractéristiques d'un graphe elle existe la struct template **plot_data** dans le programme. Cette struct prend en attribut un *argument* de type indéfini, qui correspond à la préimage d'un point *image* du graphe. Un double img en attribut représent cette *image*.
Pour l'*Espace des Phases*, on défini un *"typedef plot_data<double> Point2D"*, qui represente un point bidimensionnel de l'espace.

3.1.2 PlotStream

PlotStream est une classe *template* representante un stream de **plot_data**, une liste de points stoqués pour qu'on puisse les dessinner et les garder sur le dessin du graph de l'Espace des Phases. En attribut elle prend un *vector<plot_data>* (Pour le projet on crée un **PlotStream** de **Point2D**). Comme méthode elle implemente des *manipulateurs*, *accesseurs* et *itérateurs* pour travailler sur le stream. Elle implemente enfin la surcharge de l'*operator«* pour pouvoir ajouter des **plot_data** à la liste contenue en attribut.

3.1.3 SKAxis

SKAxys est une struct contenant les information utiles pour la constructions d'un axe dans un graph : l'*échelle*, l'*offset* et la *couleur*. Cette struct contient aussi, outre qu'un constructeur, deux méthodes pour en modifier l'offset et l'échelle.

3.2 Les Graphes

3.2.1 SKplot

SKPlot est la classe base pour le dessin de graphes dans le programme. Elle prend en attribut :

— Un pointeur sur un *QGLShaderProgram* pour le dessin du graph.
— Deux *SKAxys* pour le dessin des axes.
— Un boolean qui spécifie si les *ShaderProgram* ont été chargées.
— D'autres attribut utiles au dessin du graph par les *Shaders*.

Les Méthodes :

— Un constructeur et un destructeur.
— **initializeGL()**, fournie dans le tutoriel pour Qt, laquelle charge et compile les shaders
— **PaintData** et **PaintAxis**, méthodes virtuelles pures, pour le dessin des axes et des points à travers un ShaderProgram fourni en attribut.
— **PaintGL**, qui appelle *PaintData* et *PaintAxis*.
— **ResizeGL** fournie dans le tutoriel Qt.
— Accesseurs et Manipulateurs des différents attributs de la classe.

3.2.2 SKPlot2D

Dans *skplot2d.h* on défini une struct **ColoredFuncion**, qui prend en attribut un **PlotStream<double>** et une couleur. Une couleur est une liste de 4 float fonctionnants selon le protocole rouge, vert, bleu, alpha, où alpha correspond au facteur de *transparence*. Cette struct represente les points d'une fonction colorée.
SKPlot2D hérite directement de **SKPlot**, prend en attribut des *ColoredFunctions*, et définit *PaintData()* pour le dessin et coloriage de ces dernières. Elle définie aussi une fonction *keyPressEvent()* pour la géstion des input clavier. Cette classe est l'implementation de l'Espace des Phases 2D.

Chapitre 4

Symviewer

Cette partie concèrne l'interface grafique dediée à l'utilisateur. Son utilité est de pouvoir changer les donnés de la symulation et, en fait, elle s'interface à *Symath*, *Symgraph* et aussi *Symplot* pour obtenir accès aux *descriptors* et à les propriétés des *acteurs*. Cette bibliothèque n'utilise que les instruments de l'interface *Qt*.

4.1 Viewer window et Panneax

La fenêtre principal est charactérisée par un tableau **QListWidget** des objets présents et, à droite les panneax **SKViewerPanel** répresentatifs corréspondants. Le mécanisme de *mise à jour* des donnés dans la fenêtre est effectué par l'appel de la fonction virtuelle *updatePanel* de tous les panneaux ajoutés. Par l'appel de la méthode *updatePanels* on active ce mécanisme et on mis à jour par conséquent les panneaux.

4.1.1 ActorPanel

Ce type particulier de panneau permet de menager les variables rélatives à un **SKActor**. Il contient un panneau **SpecsChecker** pour contrôler les *specs flags*, d'un tel façon qu'on peut les abiliter ou déabiliter manuelment. De plus, si le acteur est un **Describable** et il contient un déscriptor de type **Oscillateur**, alors il va apparaitre aussi un panneau supplémentaire pour pouvoir changer l'intégrateur.

SpecsChecker

Quand un acteur est passé comme argument, la méthode *setActor* engendre les boutons *check box* en fonction de son identité. Il y aura un bouton pour chaque *classe specs* identifiée.

4.1.2 SystemPanel

Cette sous-classe de ActorPanel permet d'impiler plusiers **ActorPanel** et il prend par argument un **SymSystem**. Les panneaux ajoutés sont gerés par un **QTabWidget** qui mostre un seuls à la fois. Il contient aussi un modificateur de vecteur pour l'*offset* du système.

4.1.3 Éspace des phases

Le panneau de l'éspace des phases contient un **SKPlot2D** pour dessiner et un **Oscillateur** pour mettre à jour par rapport aux valeurs de position et vitesse stockés dans le descriptor. Ces valuers sont ajouté, à chaque appel de *updatePanel()*, dans le buffer du déssin de la courbe. L'éspace des phases contient aussi un panneau qui permet de gerer le buffer de dessin par trois boutons :

- **start** : abiliter le déssin de la courbe
- **stop** : arrêter le déssin de la courbe
- **clear** : néttoyer la courbe

4.2 Visualisateurs et modificateurs des valeurs

Pour pouvoir visualiser et modifier les valeurs on a crée trois structures de panneaux pour le faire :

- **ValueVisual** : il utilise un **QLabel** et peut afficher un valeur double
- **Valuedit** : mouni d'un **QLineEdit** permet de modifier aussi la valeur, il implémente donc un *signal* qui est émis quand l'utilisateur modifie la valeur par un double. C'est possibile de le *connécter* par des **QObject** éxtérieures.
- **VectorEdit** : c'est un éxténsion du précedent, où les QLineEdit sont déplacés en horizontal ou en vertical et un peut quand même les connécter.

Chapitre 5

Conclusion

Ce projet contient tout ce qui est nécéssaire pour faire des simulations en 3D. En fait, on a dévéloppé aussi deux tests *alpha* et *beta* pour montrer ce qu'on peut faire en utilisant ces bibliothèques. Alpha consiste dans une scène 3D et beta est un éxtension du précendent, mais il contient aussi un fenêtre pour modifier les valeurs et les constantes physiques des oscillateurs. Le code contient beaucoup d'instruments qui n'était pas demandés par la consigne du projet, comme par éxemple la *NewtonSphere* qui peut subir une force de répulsion par l'utilisateur. Mais ces instruments seront plus utiles dans le futur quand on aura besoin de simuler en 3D sans devoir se référer à des bibliothéques éxterieures. Ces trois bibliothèque, en fait, ont comme but celui de pouvoir être utilisées pour simuler des systèmes de façon complète, avec un graphisme de base optimisé et un'instrument tout à fait basilaire pour créer des modificateurs de valeurs.