Université d'Orléans — L3 Info
Informatique — 2016 - 2017

Travaux Dirigés sur Machine n°9 — Géométrie et factory— 3h non noté

Le but des ce TDM est d'implanter des classes représentant des formes géométriques ou des opérateurs sous forme de librairies dynamiques. Pour cela, il faut mettre en place une *factory* pour avoir un guichet unique pour la création de formes, et lire des formes (en post-fixé) et engendrer leur représentation graphique.

Le TDM Couvre les classes abstraites, les *template*, le pattern *factory* et le chargement dynamique de code.

Le TDM n'est pas noté et il n'y a pas de compte-rendu à rendre, néanmoins les questions posées font partie de l'exercice et aident à la compréhension. Avant de commencer le TDM, il est fortement conseiller de :

- réfléchir aux formules d'appartenance aux formes simples ainsi qu'aux opérateurs,
- travailler les questions pour le compte-rendu, en particulier le mécanisme de chargement dynamique de `test_dl.cpp` et `ok.cpp`,
- en cas de non utilisation du `Makefile` fourni ou d'un portable personnel, reporter les options de compilation et vérifier que `make ok` marche [1] et qu'il est possible d'afficher `test.ps` (PostScript).

Il est également conseillé de commencer la partie *factory* et la suite sans attendre que la première partie soit entièrement finie.

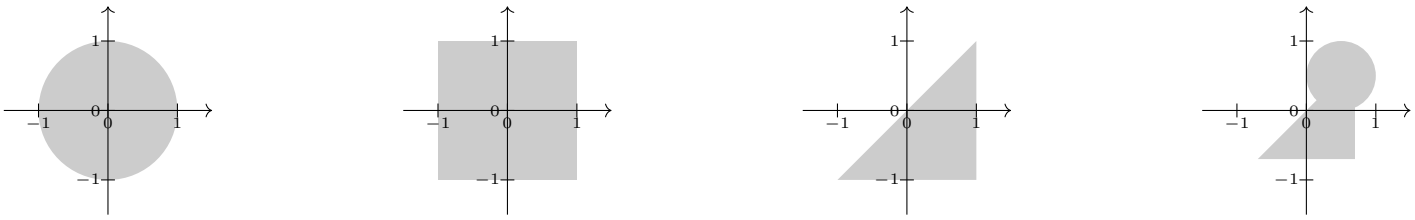
1 Formes géométriques

Elles sont représentées par des classes héritant directement d'une classe abstraite `Shape`. Cette dernière impose la définition d'une méthode pour dire si un point est ou non dans la forme.

La Figure 1(d) correspond à :

```
Shape_Union ( new Shape_Scale ( new Shape_Triangle () , .7 ) ,
    new Shape_Translate ( new Shape_Scale ( new Shape_Circle () , .5 ) , .5 , .5 ) );
```

ou en notation post-fixée : `triangle .7 scale circle .5 scale .5 .5 translate union`



(a) Disque (b) Carré (c) Triangle (d) Construction

FIGURE 1 – Figures de base.

Il faut compléter les définitions des classes fournies sans s'occuper des `create_instance` ni de l'opérateur `>>` vers `shape` (pour l'instant). Pour tester, `make TS` (et `MS` pour la mémoire) doit engendrer la sortie `test_shape.output`.

Formes de base. La classe `Shape_Circle` représente un disque centré sur l'origine et de rayon 1 (Fig. 1(a)). La classe `Shape_Square` correspond à un carré horizontal de coté 2 centré sur l'origine (Fig. 1(b)). La classe `Shape_Triangle` correspond à un triangle ayant pour sommets $(-1,-1)$, $(1,1)$ et $(1,-1)$ (Fig. 1(c)).

Modifications de forme. La forme `Shape_Scale` sert à faire une homothétie (de centre $(0,0)$) d'une autre forme. La classe `Shape_Translate` sert à faire une translation selon un vecteur. La classe `Shape_Rotate` sert à faire une rotation [2].

1 Expliquer dans le compte-rendu comment fonctionne la méthode `Shape_Scale::contains`. (Son destructeur assure la destruction de la forme translatée. Ceci devra être systématisé pour toutes les classes restantes.)

1. Ce sujet a été développé et testé sous Ubuntu.
2. Une rotation d'angle α correspond à faire $\begin{cases} x' = x.\cos(\alpha) - y.\sin(\alpha) \\ y' = x.\sin(\alpha) + y.\cos(\alpha) \end{cases}$.

Compositions binaires de formes. Il s'agit de pouvoir faire l'union, l'intersection, la différence de deux formes ou une forme moins une autre forme. Ces compositions ne se distinguent que par l'utilisation d'une fonction différente au niveau de contains. Ceci est réalisé par le *template* du fichier shape_binary.hpp plus des instantiations dans les différentes bibliothèques partagées. Tout ce qui fait le déploiement du patron est déjà en place (par exemple pour shape_union). Il faut écrire le *template* de shape_binary.hpp.

2 Expliquer dans le compte-rendu comment defines et *template*s marchent de concert.

2 Comprendre le chargement dynamique

3 Commenter le fichier test_ok.cpp pour expliquer les étapes du chargement dynamique. Expliquer les options de compilation utilisées pour cela dans le Makefile.

3 Mise en place d'une *factory*

Le but est d'obtenir, par exemple, une instance de la classe Scale_Circle en demandant un "circle" à la Factory_Shape (de même pour les autres formes de base). Il en est de même pour les opérateurs ci-dessus en passant les arguments nécessaires par une pile. Pour cela, la *factory* doit connaître le moyen de construire chacun d'eux.

Les informations pour la construction sont stockées dans un tableau associatif (std::map). La méthode register_shape permet d'ajouter une paire (nom,fonction) à ce tableau. Le nom est une chaîne de caractères. La fonction est un shape_creator qui prend en argument une pile (U_Sh_d_Stack) pour y lire des arguments.

Cette lecture d'arguments correspond à une approche post-fixée. Les arguments de la pile sont soit des Shape soit des double. Le module u_sh_d fournit un type adapté de pile (U_Sh_d_Stack).

La collecte des informations se fait avec le constructeur de la factory. celui-ci doit charger toutes les librairies dynamiques listées dans le fichier factory_shape.config. Son fonctionnement est similaire à celui de test_ok.cpp.

4 À la destruction de l'instance, tout ce qui a été chargé doit être libéré. Expliquer dans le **compte-rendu** pourquoi on ne peut le libérer avant.

Écrire tout le module factory_shape et des méthodes create_instance ayant besoin d'être redéfinies. Cela peut être testé avec make TF1 TF2 MF1 MK2.

4 Lecture d'une shape

La lecture se fait en mode post-fixé avec une pile : les arguments avant l'opérateur. Des fichiers d'exemple et de test sont fournis (fig*n*.shape). Si un double est lu, il est empilé pour servir d'argument. Si une chaîne est lue, une shape est créée (en puisant si nécessaire sur la pile) en utilisant la *factory*. La shape créée est empilée sur la pile. À la fin de la lecture du flux, il ne doit y avoir dans la pile qu'une shape. Celle-ci est retournée. La lecture est à tester avec make TR*n* make MR*n* où n est un numéro de fichier test.

5 Comment, au niveau de la lecture, distinguer les doubles des chaînes de caractères ?

5 Visualisation

Vu l'absence de primitive standard de dessin et l'utilisation de matériels hétérogènes, la visualisation se fait par la création d'un fichier PostScript en-capsulé (.eps). Il s'agit d'un système minimaliste :

- on crée un Export_Eps en indiquant le nom du fichier à créer, la taille en pixels ($[0..x\_max] \times [0..y\_max]$) et éventuellement la taille d'un pixel, puis
- on peut noircir un pixel (plot) et
- on peut faire noircir une Shape (plot).

L'affichage des formes se fait en parcourant tous les pixels et en demandant à chaque fois s'il est dans la forme [3]. Le code se trouve dans test_shape_eps. Il n'y a rien à écrire. Tester avec make E*n* permet de voir si tout marche bien. Il produit à chaque fois un fichier fig*n*.eps.

3. On fait autrement pour afficher des formes simples, néanmoins, cette idée de parcourir tous les pixels se retrouve, par exemple, dans les rendu 3-D à base de lancer de rayons.