Alain Milliet #228408
Julien Perrenoud #227924
02.06.2013

FLAME MAKER : BONUS

Possibilité de déplacer le cadre

Description

Comme le titre l'indique, cette amélioration aura pour but d'ajouter à l'interface utilisateur la possibilité de déplacer et redimensionner le cadre de la fractale. Afin d'ajouter un peu de créativité à ce bonus, il sera également possible de déplacer directement le cadre en glissant avec la souris sur les composants graphique (à la manière de Google Map).

Une autre implémentation de ce bonus est la possibilité de dessiner le cadre sur les composants graphiques. Cela était à la base destiné à aider l'utilisateur à se représenter quelle partie du plan il « visait » avant de pouvoir exporter l'image.

Mise en œuvre Java

La première chose à faire est de rendre le cadre de fractale Observable et mutable. Pour ce qui est de la mutabilité, on mets en œuvre le patron de conception *Builder*, et on lui ajoute un *HashSet* d'*Observer* ainsi que des méthodes pour en ajouter/enlever. Ceci nous donne donc la classe *Rectangle.ObservableBuilder.*

Une fois cela fait, la première chose a faire et d'offrir aux composants graphiques la possibilité d'agir directement sur ce builder. On crée pour cela deux nouvelles classes,
NavigableFlameBuilderPreviewComponent et *NavigableAffineTransformationComponent*, prenant en argument les mêmes arguments que leurs homonymes si ce n'est que cette fois il nous faut un *Rectangle.ObservableBuilder* au lieu d'un simple *Rectangle.*

Dès lors, la gestion du cadre devient relativement simple. On ajoute un *MouseListener* sur chacun des deux composants à leur création, afin de savoir quand on clique (*onMousePressed*) et quand on relâche (*onMouseReleased*) la souris. Au relâchement, une fonction *translateCenter* calcule et applique en fonction du point de départ et d'arrivée la translation correspondante au trajet de la souris. A noter encore que toutes les implémentations communes aux deux classes sont faite dans la superclasse non-instantiable *AbstractNavigableComponent*, afin d'éviter la duplication de code.

Cela dit, mis à part le petit gadget qu'est la modification sur le composant graphique, on aimerait également ajouter un interface permettant de gérer les autres paramètres tels que la hauteur et la largeur. Pour cela, on crée simplement un nouveau *JPanel* avec *JTextFields* et *JButtons*. Les *JTextFields* possèdent chacun un *ActionListener* qui utilise les méthodes setWidth, setHeight ou *setCenter* du bâtisseur de rectangle, mais sont aussi tous observateurs de ce dernier, de sorte que lorsqu'on zoom ou se déplace sur les composants graphiques, leurs valeurs sont automatiquement mises à jours.

On ajoute également 4 boutons, instances de la classe imbriquée *ZoomButton*, et qui permettront de simplement en un clic de zoomer ou dé-zoomer d'un certain rapport (Ici le zoom se résume à agrandir ou rapetisser le rectangle, donc on est obligés de recalculer à chaque changement.

Pour ce qui est du dessin du cadre, on écrit la méthode *PaintFrame(Graphics g)* dans la superclasse *AbstractNavigableComponent* et on la réutilise dans les deux sous-classes, si et seulement si leur attribut *isFrameVisible* est *true.*

Remarques

Nous avons finalement décidé d'opter pour cette solution certes pas optimale du point de vue de la réutilisation du code (un peu de code dupliqué entre navigable et non-navigable) car les autres solution nous heurtaient à des problèmes bien plus complexes.

La mise en œuvre du patron stratégie aurait été une très bonne idée. Créer une classe *NavigableComponent* et l'utiliser pour décorer les deux composants graphiques était d'ailleurs la première idée. Néanmoins, le changement entre Rectangle et *Rectangle.ObservableBuilder*, ainsi que les problèmes de dessin d'un *JComponent* décoré nous ont vite persuadés d'utiliser autre chose, car il fallait alors soit recréer une instance du composant sous-jacent à chaque *repaint* et créait beaucoup de problèmes de cast, à moins de refaire beaucoup de classes et donc écrire plus de code qu'en simplement créant une nouvelle classe avec des propriétés similaires.

Nous avons également essayé de redéfinir la méthode *paintComponent* des *NavigableComponent* en créant un *AffineTransformationComponent* ou un *FlameBuilderPreviewComponent* et en le dessinant, mais il faut faire appel à des méthodes supplémentaires et pose certains problèmes de redimensionnement.

Modification de la densité

Description

Ce bonus est très certainement celui qui fut le plus simple à mettre en œuvre, cependant son utilité n'en est pas pour autant moindre. En effet, pouvoir changer la densité de la fractale est extrêmement utile car cela permet d'améliorer grandement la rapidité du programme lorsque l'utilisateur n'accorde pas beaucoup d'importance à la précision avec laquelle la fractale est dessinée.

Mise en œuvre

La mise en œuvre est assez similaire à celle de l'index de transformation courante, mais cette fois-ci l'attribut à observer est *density*.

On crée donc un interface *DensityObserver* et les méthodes d'ajout/suppression habituelles. Dès lors, afin de respecter le patron MVC, le changement de densité appelle la méthode *setDensity* qui, elle, notifiera les *DensityObserver* (= le composant graphique de la fractale) lors d'un changement.

Pour choisir une nouvelle valeur de densité, nous avons opté pour un *JSlider*, car ce composant permet d'avoir une valeur minimale et une valeur maximale définie à l'avance (on ne voudrait pas que l'utilisateur fasse planter le programme en entrant une densité de 10000).

Afin d'attribuer cette valeur, nous avons opté pour un simple *JButtton* qui récupère la valeur du *JSlider* et appelle la méthode *setDensity.* Il ne serait en effet pas très pratique de faire le changement directement avec un *ActionListener* sur le *JSlider* qui serait appelé lors du changement de valeur, car cela recalculerait la fractale à chaque valeur et prendrait alors un temps absolument énorme.

Alain Milliet #228408 02.06.2013
Julien Perrenoud #227924

Palette de couleur

Description

La palette de couleur permet de donner à l'utilisateur le choix des couleurs qui apparaîtront sur la fractale en cours. Notre palette permet dans un premier temps de sélectionner la couleur à modifier, de la modifier tout en gardant la couleur initiale dans la palette et ensuite de la remplacer par la nouvelle couleur créée. La liste créée permet aussi d'inverser les couleurs dans la palette ce qui permet de mieux comprendre où chaque couleur se place par rapport à son emplacement dans la palette.

Mise en œuvre

La mise en œuvre n'est pas des moindres car plusieurs nouvelles composantes constituent ce bonus. Il a d'abord fallu créer une liste où chacune des couleurs ont été stockée, pour cela nous avons créé une nouvelle classe imbriquée comme pour la liste des transformations mais cette fois-ci nommée *ColorsListModel* qui a les même propriétés que la classe *TransformationsListModel* mais avec une nouvelle méthode *invertColor* permettant comme son nom l'indique d'inverser deux éléments de la liste (ici des couleurs). Cette méthode n'est pas directement définie dans la classe imbriquée et appelle la méthode *invertColor* de *ObservablePalette*.

Nous avons décidé de créer cette nouvelle classe *ObservablePalette* pour éviter de devoir retoucher nos anciennes classes afin que le programme FlameMaker sans bonus ainsi que celui avec bonus puissent être tous les deux appelés sans modification de code. Notre classe observable pour la palette utilise une *interpolatedPalette* dans son constructeur ainsi que sa méthode *colorForIndex*. Elle contient surtout un *HashSet* d'observateurs qui sont appelés lors d'une modification de la liste.

La *JList<String>* de couleurs a exactement les mêmes propriétés que la liste de transformations. Les boutons d'ajout/de suppression sont eux aussi similaires aux boutons de la liste de transformations mais ils modifient également la couleur affichée dans les prévisualisation de l'image sélectionnée ainsi que le droit d'utilisation des boutons *Up* et *Down* également ajoutés lors de cette étape. Ces boutons utilisent la méthode *invertColor* décrite ci-dessus.

L'affichage de la palette (*PalettePreviewComponent*) a comme toutes composantes graphique deux méthodes simple, l'une relativement simple étant getPreferredSize() qui retourne la dimension désirée de la composante et l'autre *paintComponent(Graphics g0)* permettant de dessiner la composante. C'est dans le constructeur de cette classe que l'on ajoute un observateur à la palette pour que la prévisualisation de la palette se mette à jour lors d'une quelconque modification de la palette.

La dernière partie de l'amélioration est la possibilité de modifier une couleur appartenant déjà à la palette, nous avons utilisé ici trois *JSlider*, un par couleur primaire, ce qui permet de créer toutes les couleurs possibles. Ces *JSlider* mettent à jour le *colorPreviewComponent* qui est tout simplement un espace où la couleur qui est modifié apparaît avant son intégration dans la palette.
Ces deux composantes de prévisualisation permettent à l'utilisateur de voir la couleur qu'il va remplacer dans la palette et la couleur qu'il va y mettre à cette place (lorsque le bouton *replace* sera appuyé).

Remarque

Cette amélioration n'est pas compliquée à imaginer, malgré tout l'utilisation des observateurs est quelques peu périlleuse et il ne faut surtout pas oublier de mettre à jour toutes les composantes concernées lors d'une modification.

Ajout d'un menu basique

Description

Cette dernière implémentation n'a pour le moment pas énormément d'utilité si ce n'est un coté gadget, mais elle a principalement été faite par curiosité. Il s'agit en gros d'implémenter un menu principal à l'application.

Son utilité principale sera tout de même de pouvoir décider de quel composant graphique on veut disposer dans la fenêtre.

Mise en œuvre Java

La création d'un menu est quelque chose de finalement assez simple. On crée une instance de JMenuBar, auquel on ajoute des instances de Jmenu qui contiendront eux-mêmes des instances de JMenuItem.

Ensuite, décider de l'action a effectuer sur tel ou tel JMenuItem se fait tout naturellement avec un ActionListener. On peut par exemple Créer un JMenuItem « Exit » appelant la méthode System.exit(0).

On peut également ajouter des raccourcis aux JMenuItem en utilisant la méthode setAccelerator(KeyStroke.getKeyStroke(...)) ou on remplace les ... par les valeurs des touches du clavier (exemple : KeyEvent.VK_F & KeyEvent.CTRL_MASK pour simuler Ctrl + F).

Une fois tout cela compris, il est possible de faire un menu « Display » permettant de facile choisir quels composants graphiques on souhaite afficher. On crée pour cela une sous-classe de JCheckBoxMenuItem (MenuItem avec CheckBox, donc) qui, lorsqu'on clique dessus, appellera la méthode toggleVisibility(JComponent component), qui se charge de gérer la visiblité du composant choisi via la méthode setVisible(boolean).

Remarques

Ce Bonus ayant été le dernier à être fait, il y a beaucoup de choses que nous aurions aimé rajouté mais n'avons pas eu le temps.

De plus, aussi bien que gère le *BorderLayout* le redessin lors des *setVisible*, le *GridLayout* quant à lui ne permet pas de manière triviale de redessiner le composant en enlevant un des deux composants graphiques. C'est la raison pour laquelle l'appel de la méthode *toggleVisibility* ne fonctionne pas complètement sur les deux composants graphiques, qui laissent alors simplement un trou au lieu d'allouer l'espace restant.