Rapport d'avancement du projet FriendPaint (9 janvier 2015)

Valentin BOCQUEL - Sébastien BOURDIN - Yoan LECOQ
Tuteur : Konstantin VERCHININE

Table des matières

Architecture globale du système

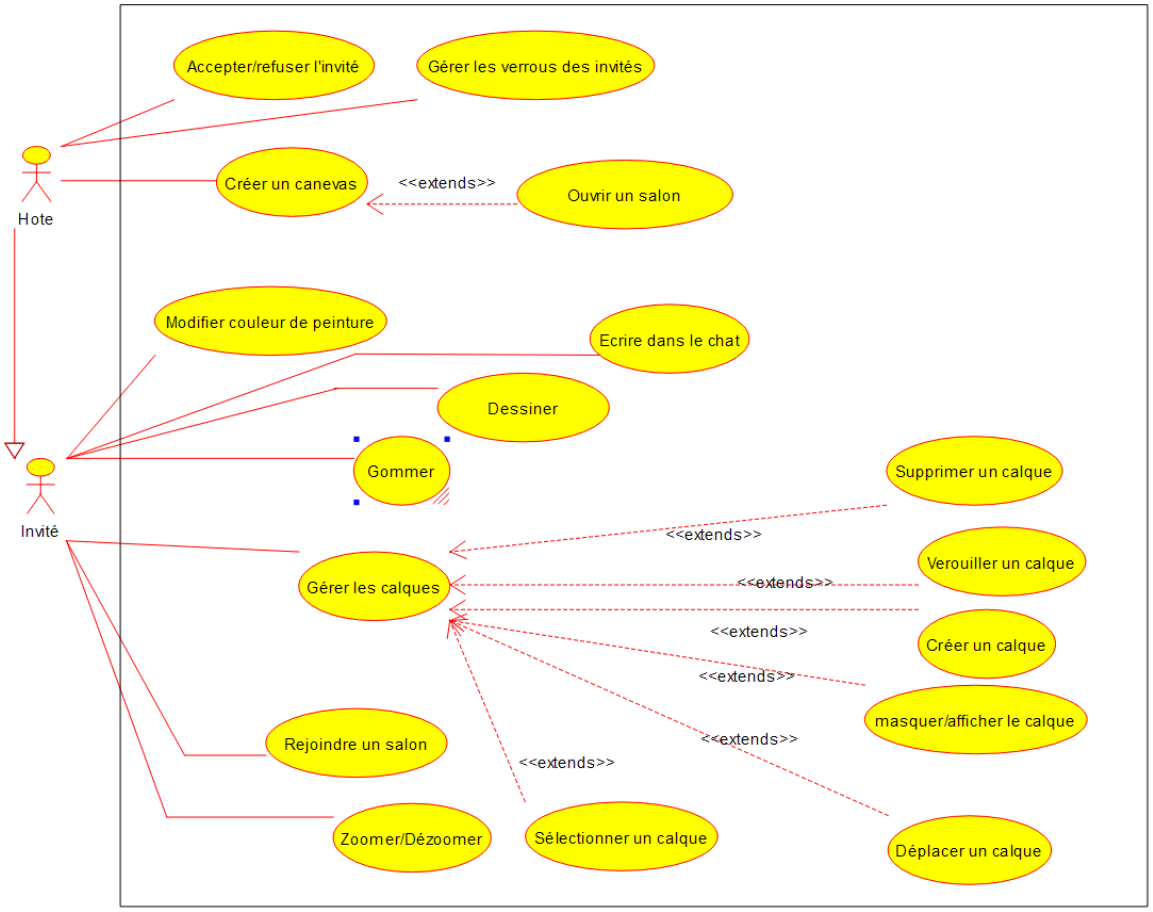
Notre système consiste en deux exécutables : FriendPaint et FriendPaintServer.
FriendPaint est un programme client qui se connecte à un (ou plusieurs) programme(s) serveur FriendPaintServer.

FriendPaint relie deux API que nous développons : GUI (interface graphique, entrées utilisateur) et PaintEngine (les fonctionnalités définies dans le cahier des charges).
Ces deux API sont conçues pour être indépendantes l'une de l'autre, pour deux raisons :
-> La séparation de la Vue et du Modèle (MVC) au sein de FriendPaint;
-> Elles seront individuellement réutilisables par d'autres programmes, à l'avenir.

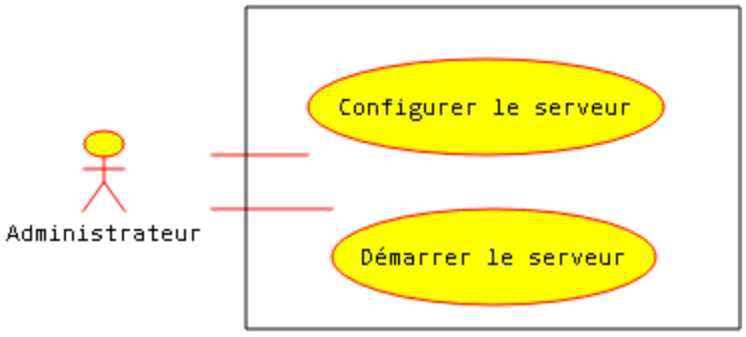
FriendPaintServer est simplement un programme exploitant l'API PaintEngine différemment du client.

Cas d’utilisation, séquences

Le diagramme de cas d’utilisation du client est inchangé. Cependant, ici vient s’ajouter le diagramme de cas d’utilisation de FriendPaintServer.

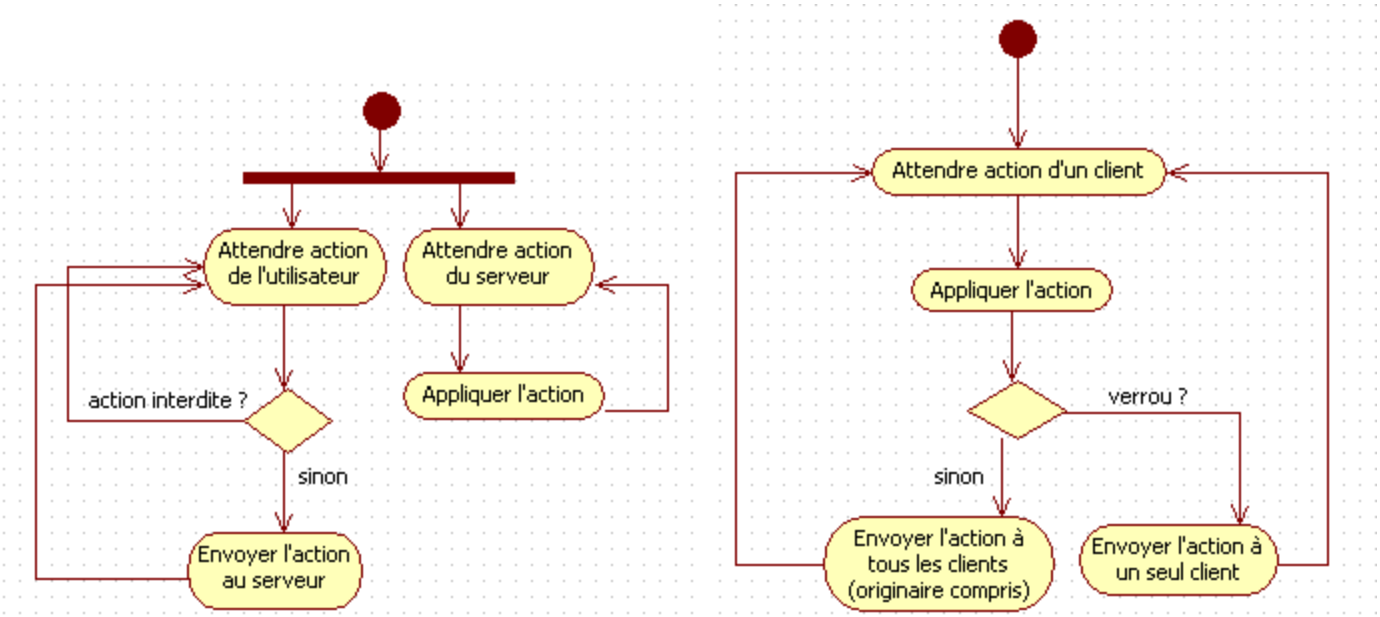


Ci-dessous, le diagramme de cas d’utilisation de FriendPaintServer.



Algorithmes, protocoles

Notre projet ne comporte pas d'algorithmes complexes, cependant nous explicitons le comportement du client et du serveur. Ci-dessous, un schéma de la boucle principale du client (gauche) et celle du serveur (droite).



Les actions sont, par exemple, la pose d'un verrou de calque pour un utilisateur, un tracé de crayon, etc.
Ce modèle applique la politique que nous avons choisi :
-> Le moins de charge possible pour le serveur;
-> Etat consistent du canevas pour chaque utilisateur.
Les deux inconvénients assumés sont :
-> Le serveur fait une confiance aveugle aux clients;
-> Il est possible que l'utilisateur ne voie pas son action appliquée immédiatement.

La définition de notre protocole de communication à travers le réseau est relativement longue et fait donc l'objet d'un fichier à part : "Protocole.txt"

Diagrammes de classe

NOTES :
Les diagrammes de classe, pour des raisons de lisibilité, sont joints au format PDF dans l'archive que nous rendons (PaintEngine.pdf et GUI.pdf).
Afin de faciliter leur compréhension, les paragraphes suivants fournissent des explications globales, puis spécifiques à chaque diagramme de classe.
Pour le confort de lecture, les méthodes "accesseurs" ont été omises de ces diagrammes, mais sont présentes dans le code source.
Nous y utilisons explicitement certains opérateurs du C++, notamment l'opérateur de référence ('&') et l'opérateur de portée ('::').
Les classes de la SFML sont beaucoup exploitées (préfixées par 'sf::') et leur utilité précise n'est parfois pas immédiatement apparente. Voici quelques clarifications :
-> sf::Uint8, sf::Uint32, etc. sont des types primitifs.
-> sf::Vector2<T> contient deux valeurs d'un type T. (Il existe les alias sf::Vector2u (unsigned) et sf::Vector2f (float)).
-> sf::FloatRect est alias pour sf::Rectangle<float>, soit un rectangle à coordonnées réelles.
-> sf::SocketSelector garde une liste de pointeurs vers des sf::Socket, et stocke dans une file leurs messages, dans l'ordre de réception. On peut ensuite traiter les requêtes une par une. Cette classe permet à un serveur de se passer du modèle "fork on accept".
-> sf::Image abstrait un tableau de pixels.
-> Une classe héritant de sf::Drawable peut être dessinée à l'écran.
-> La fenêtre génère régulièrement des sf::Event, qui encapsulent les entrées utilisateur et les actions sur la fenêtre.

Diagramme de classes de GUI (*GUI.pdf*)

GUI fait usage du design pattern "Composite", pour permettre au développeur de gérer par hiérarchie des Widgets et des conteneurs de widgets (WidgetContainer). Voici des clarifications pour les classes les plus complexes :
-> WidgetNode contient les mécaniques essentielles au fonctionnement des Widgets et conteneurs de Widgets.
Un WidgetNode comporte un rectangle dessinable (sf::RectangleShape), qui sert à contenir sa position, sa taille, et sa couleur.
Le développeur arrange son interface graphique en spécifiant des "**insets**" pour ses Widgets/conteneurs. C'est à dire "**combien d'espace laisser**, à gauche, au-dessus, à droite, et en-dessous par rapport à son **parent**".
Cette méthode rend très facile la création d'interfaces.
-> Slider est un ascenseur. Il connait le pourcentage d'avancement de sa poignée, et le convertit en valeur en utilisant une fonction que le développeur lui indique (calculator). Ainsi, par exemple, on peut créer un ascenseur générant des valeurs de 0 à 255.
-> TextContainer sert à contenir du texte intelligemment. Il est utilisé comme un flux de sortie standard; Le texte qui déborde est placé sur une nouvelle ligne, et seules les lignes de texte les plus récentes sont affichées. On peut y imprimer un texte coloré, et/ou italique.
-> CanvasWidget est un widget contenant une **vue** sur un canevas (ClientCanvas, qui est une classe de l'autre diagramme). Il attrape les évènements générés par la souris et applique les actions correspondantes sur le canevas.

Diagramme de classes de PaintEngine (*PaintEngine.pdf*)

Le diagramme de PaintEngine est plus facilement lisible dès que son principe est acquis.
Sa complexité est dûe au fait que le client et le serveur n'ont pas les mêmes besoins en termes de stockage de données.
Les classes propres au client sont vertes et cyan pour le serveur. Les classes jaunes servent à réunir des informations communes aux deux.

Les classes Server et Client n'utilisent pas le design pattern "Singleton" (volontairement), mais des attributs et méthodes statiques, le tout pour le confort de lecture. Ainsi, au lieu de d'écrire, par exemple :
`Client::getInstance().getVersion()`
nous écrirons :
`Client::getVersion()`

Rapport d'activité

Nous avons concentré nos efforts sur l'API GUI et la gestion de piles de calques, et ceux-ci sont terminés.
Une partie du temps a été prise par la définition du protocole, le choix de la politique de fonctionnement du client et du serveur, et par des mises à jour régulières des diagrammes de classes.
Nous considérons la phase d'analyse complètement terminée.
Nos efforts vont d'abord se consacrer à la réalisation d'un client sans fonctionnalités réseau. Ensuite nous développerons le serveur, en nous basant sur un serveur de "tchat" que nous avons développé auparavant, et modifierons notre client.
Actuellement, nous travaillons activement sur une démonstration pour notre tuteur.

Diagramme de Gantt actualisé