Pierre Dénès, Diane Joly, Lucas Riou, Romain Pelletier

Projet IPI : Color Flood

Index

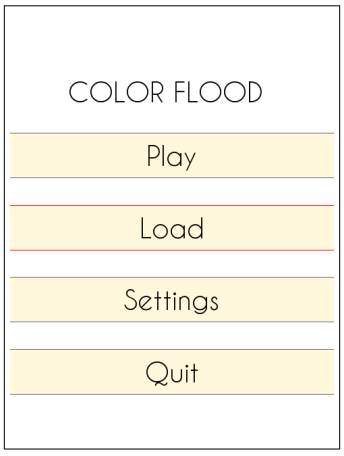


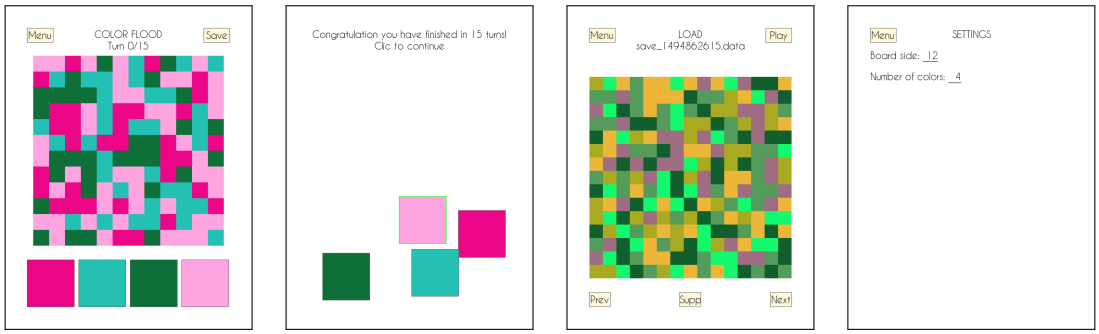
FIGURE 1 – Menu d'accueil du jeu

1 Introduction : Travail effectué

Le cahier des charges de ce projet demande de livrer un jeu fonctionnel, équipé d'une interface graphique ainsi que de deux solveurs. Le code doit être documenté, et des tests unitaires doivent être effectués sur les fonctions agissant sur les structures de données du jeu.

A la fin des 4 sprints prévuw, l'archive livrée contient bien le jeu avec toutes les fonctionnalités requises, cependant le code n'est pas entièrement documenté, et les tests unitaires sont également absents.

Ce rapport vise a expliquer comment ce travail a été produit, ainsi que les raisons des divers retards et manquements au cahier des charges.



(a) Ecran de jeu (b) Ecran de fin (c) Chargement (d) Réglages

FIGURE 2 – Les différents écrans du jeu

1.1 Menu

Le menu contient donne accès aux quelques fonctionnalités qui permettent de paramétrer et lancer une partie. Les boutons sont uniquement sélectionnables à la souris (cadre rouge quand survolé, cadre vert quand pressé, action déclenchée a la relâche)

1.2 Partie

Lors d'une partie, le joueurs peut jouer une couleur en la sélectionnant au bas de l'écran, revenir au menu (la partie est alors perdue), ou sauvegarder dans un fichier. Au lancement une solution est affichée dans la console

1.3 GameOver

L'écran de fin de partie est constitué d'un message en fonction de l'issue de la partie, et des boutons de couleurs qui rebondissent en fonction de leurs différentes composantes RGB.

1.4 Chargement

Cet écran affiche le nom de la sauvegarde, sa grille, ainsi que des boutons de navigation. (NB : le bouton Supp sensé supprimer la sauvegarde n'a pas été implémenté et est donc inactif)

1.5 Réglages

Il est possible ici de régler la taille et le nombre de couleur. Il serait intéressant d'implémenter la possibilité de choisir les composantes RGB de chacune car la génération aléatoire ne permet pas toujours de les distinguer a l'oeil nu.

2 Organisation du travail

2.1 Romain

Sa seule participation s'arrêtera a faire rapidement une présentation powerpoint. A ce propos l'intéressé dira :

> Vis à vis de la collaboration, nous avons eu quelques échecs. Pour ma part j'ai pêché par mon manque de temps de cerveau disponible à cause de mes projets extérieurs et des quelques problèmes que j'ai pu avoir. — *Romain Pelletier, 15 Mai 2017*

Personnellement ça me fait bien rire de dire ça à la fin (Pierre).

2.2 Lucas

Même si il s'est plus "impliqué" que l'individu précédent, son absence pour la présentation ainsi que le fait qu'il ne rédige pas lui même ce paragraphe parlent d'eux même.

Lucas avait commencé a se pencher sur les tests unitaires, qui n'ont au final jamais été faits, et un début de solveur qui ne mérite pas vraiment ce nom.

2.3 Diane

- Lot A & B : documentation du jeu
- Lot C : solveur, structure de pile
- Lot D : solveur (pas retenu, ne fonctionne pas) + documentation du solveur

J'ai essayé d'établir un lien et de poser des délais et une répartition des tâches avec les autres membres. On a finit par arrêter d'essayer de communiquer avec deux membres qui ne répondaient plus (ce qui n'a pas eu l'air de les inquiéter plus que ça).

Je me suis aussi occupée de la communication avec les profs à propos des difficultées rencontrées et des retards.

Ce que le projet m'a apporté : la maîtrise de doxygen, savoir comprendre le code écrit par quelqu'un d'autre, découvrir la gestion de projet où il faut savoir s'imposer pour etre écoutée.

2.4 Pierre

Après l'échec du premier sprint, où personne moi y compris, ne s'est manifesté pour prendre en charge le lancement du projet, j'ai rapidement effectué un squelette du code du jeu, avec des fonctions a remplir pour tout le monde. Cependant comme j'en avais besoin pour avancer plus loin, je me suis permis de les remplir moi-même.

Ensuite pour le second sprint, après un recadrement, l'idée était que je me charge de faire une interface graphique avec SDL, pendant que les autres personnes se consacraient à la documentation et aux tests unitaires, permettant à tout le monde d'être au point sur le fonctionnement du code du jeu.

Après la livraison de la première version de l'interface au lot C, je me suis penché sur un solveur bruteforce dont le fonctionnement est expliqué plus tard. A ce point seule Diane répondait encore sur la conversation Scoledge dédiée au projet, nous avons donc complété autant que possible le cahier des charges à deux.

J'ai ensuite complété l'interface avec le chargement des sauvegardes. Comme le solveur heuristique de Diane ne fonctionnait pas, nous avons aussi rapidement repris le code du solveur bruteforce pour le transformer en solveur heuristique basique et l'intégrer à l'interface.

2.5 Temps investi

Nom	Sprint A	Sprint B	Sprint C	Sprint D
Romain	0	0	0	0
Lucas	?	?	8 ?	0
Diane	5	5	15	8
Pierre	10	10	100	30

Il est difficile d'estimer le temps investi par chacun avec notre manque d'organisation. Le temps d'apprentissage de la SDL est aussi difficile a quantifier.

2.6 Sur la méthode agile

Le manque d'implication de chacun a rendu difficile, voire impossible l'utilisation de la méthode agile. Le seul rôle vraiment représenté aura été le dev pour pouvoir rendre un jeu avec toutes les fonctionnalités.

Cependant la non application de la méthode n'est pas sans enseignement puisqu'on constate le résultat : manque de productivité, retards sur les sprints, mauvaise communication. On retiendra l'utilité des méthodes agiles par l'échec dû à leur absence.

3 Algorithmes

3.1 Gestion interne du jeu

La structure du jeu contient deux grilles, une pour les couleurs, une pour les étiquettes des composantes connexes. L'algorithme intéressant du fonctionnement est le flood fill qui permet la coloration d'une composante connexe, ainsi que la propagation de l'étiquette associée

```
Data: Une grille et des coordonnées (x,y)
Result: Propagation de la couleur/étiquette dans la grille
Récupération de la couleur/étiquette en (x,y);
for chaque case voisine do
    if la case existe et répond aux conditions de propagation then
        appel récursif sur cette case;
        f
    end
end
```

Algorithm 1: Algorithe Flood Fill

La condition de propagation pour les couleurs est que la case appartienne a la même composante connexe et soit d'une couleur différente.

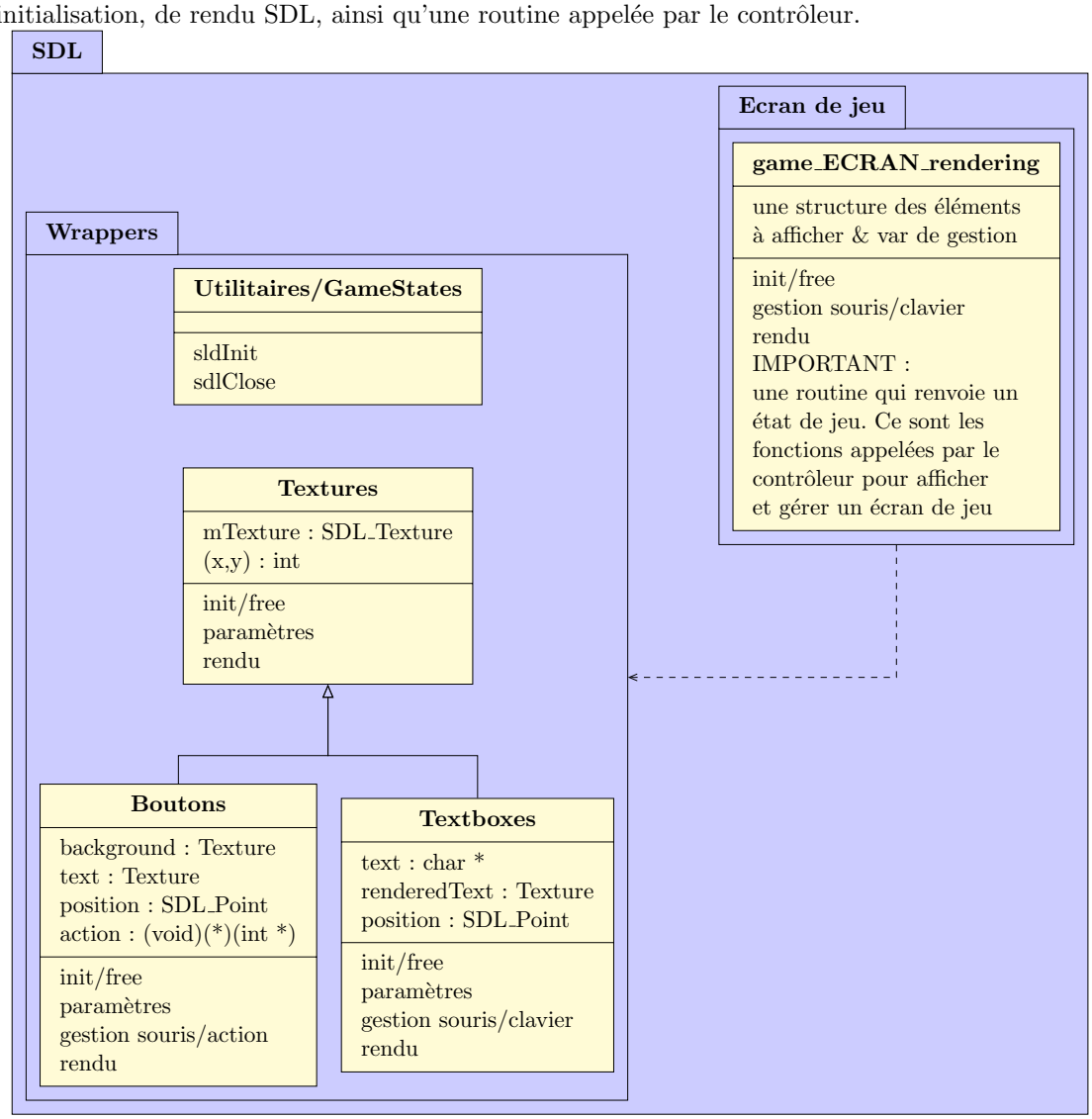
La condition de propagation pour les étiquettes est que la case soit de la même couleur et d'une composante connexe différente.

3.2 SDL

Il n'y a aucun algorithme notable dans ce module du code, mais son organisation est intéressante à expliquer. Il y a deux types de fichiers : des utilitaires/wrapper et des fichiers de rendering.

Les wrappers regroupent des fonctions SDL pour faciliter la réalisation des différents écrans et éviter en partie la répétition de code.

Les fichiers de rendering correspondent a un écran et contiennent une structure, des fonctions d'initialisation, de rendu SDL, ainsi qu'une routine appelée par le contrôleur.



L'utilisation de routines et de GameStates permet d'avoir un main très épuré :

```c
int main(void) {
  SDL_Window *gWindow = NULL;
  SDL_Renderer *gRenderer = NULL;
  /* default settings for a game */
  int boardSize = 12, cNb = 4;
  game *g = NULL;
  srand(time(NULL));

  if(!sdlInit(&gWindow, &gRenderer)) {
    fprintf(stderr, "Failed to initialize!\n");
  }
  else {
    GameState gs = GAMESTATE_MENU;
    SDL_Event e;

    while(gs != GAMESTATE_QUIT) {
      switch(gs) {
        case GAMESTATE_MENU:
          /* gs internaly changed but to stay coherent... */
          gs = menuRoutine(&gWindow, &gRenderer, &e, &gs);
          break;

        case GAMESTATE_PLAYING:
          /* if no game loaded from save, create a new one */
          if(g == NULL) g = gameInit(boardSize, cNb);
          gs = boardRoutine(g, &gWindow, &gRenderer, &e);
          gameFree(g);
          g = NULL;
          break;

        case GAMESTATE_SETTINGS:
          gs = settingsRoutine(&gWindow, &gRenderer, &e, &boardSize,
              &cNb);
          break;
        case GAMESTATE_LOAD:
          gs = previewRoutine(&g, &gWindow, &gRenderer, &e);
          break;

        default:
          gs = GAMESTATE_MENU;
          gs = menuRoutine(&gWindow, &gRenderer, &e, &gs);
      }
    }
  }
  sdlClose(&gWindow, &gRenderer);
  return 0;
}
```

3.3 Solveurs

3.3.1 Bruteforce

L'idée est de jouer toutes les combinaisons possibles pour trouver la meilleure possible. Pour éviter des appels trop fréquents a l'algorithme Flood Fill, les structures représentant la grille ont été modifiées. Au lieu de matrices, on utilise :

— Une matrice d'adjacence
— Un tableau de correspondance étiquette → couleur
— Un tableau de 0 et 1 pour chaque composante connexe, selon si elle est "coloriée" ou non

```
Data: Matrice d'adjacence
      Tableau de correspondance
      Tableau des composantes "coloriées"
      Profondeur maximale de récursion
      Pile des coups joués pour la solution courante
      Pile des coups de la meilleurs solution
      Couleur jouée
Result: Meilleure solution stockée dans la pile correspondante
if premier tour then
    for chaque composante voisine de la composante 1 do
        if sa couleur n'a pas encore été marquée && composante non coloriée then
            marquer la couleur;
        end
    end
else
    for chaque voisin à une composante coloriée do
        if le voisin n'est pas de la couleur jouée then
            marquer sa couleur;
        else
            colorier le voisin;
            for chaque voisin non colorié de la composante nouvellement coloriée do
                marquer sa couleur;
            end
        end
    end
end
ajouter à la pile courante de coup joué;
if gameover then
    copie de pile courante dans le pile de la meilleure solution;
    mise a jour de la profondeur max de récursion;
else
    if récursion ne dépasse pas le seuil de profondeur then
        for chaque couleur marquée do
            appel récursif du solveur;
        end
    end
end
```

Algorithm 2: Algorithe Solveur BruteForce

3.3.2 Heuristique

L'idée de départ était la suivante :

— Une fonction qui compte pour une case donnée le nombre de cases dans la même composante connexe

— Une fonction qui compte, pour chaque couleur, le nombre de cases capturées si la couleur est jouée

Le but était d'optimiser le solveur en jouant par préférences les couleurs qui ajoutent le plus de cases dans la composante connexe du joueur. Le solveur aurait ensuite pu être modifié pour trouver le plus de cases coloriées en n coups. Le problème du premier solveur était le temps pour déterminer une première solution, et la complexité exponentielle. Avec cette méthode on passerait d'un solveur lent sur une grille 10x10 a un solveur rapide et performant sur des grilles bien plus grandes.

Par manque de temps, cet algorithme n'a pu être implémenté, une solution moins performante a été implémentée pour la SDL. L'algorithme BruteForce a été repris en remplaçant le marquage des couleurs par un comptage du nombre de composantes voisines de chaque couleur. La récursion se fait alors uniquement sur la couleur avec le plus grand nombre de composantes connexes. Une solution est rapidement trouvée, mais elle comporte un grand nombre de coups en plus.

4 Améliorations

4.1 Code

Un point évident à changer est la documentation inexistante pour la SDL. Toujours pour la SDL, d'autres wrappers pourraient être mis en place, notemment pour les banner/footers utilisés plusieurs fois. Le code gagnerait aussi en propreté en définissant des normes sur l'ordre des paramètres, les noms, etc...

4.2 Algorithmes

L'algorithme en BruteForce pourrait être accéléré énormément en changeant la matrice d'adjacence. C'est une matrice creuse qu'on parcourt a la recherche de 1, il serait plus rapide de garder des listes des composantes voisines.

L'arbre des combinaisons possibles peut être élagué en jouant une couleur dès qu'elle colorie toutes les dernières composantes de cette couleur, et ignorer les autres couleurs dans la récursion.

Enfin, la complexité dépend beaucoup de la longueur de la première solution trouvée. En effet, l'algorithme cherche des solutions successives de taille n, n-1, n-2... Trouver une première solution de manière heuristique permettrait d'assurer un temps d'exécution plus constant et donc un temps moyen plus petit.

L'implémentation fonctionnelle de l'algorithme jouant la meilleurs combinaison de n coups permettrait peut-être d'obtenir une meilleure solution qu'en jouant le maximum de composantes connexes sur 1 coup, ou du moins de comparer les performances.

Avoir transformé la grille en graphe permettrait d'utiliser la théorie des graphe, ce qui est une piste qui mérite d'être explorée.

Tout un travail d'analyse des performances est aussi a effectuer.

4.3 Organisation

L'amélioration principale pour les prochains projet est l'organisation du travail. Avoir un vrai leader, un vrai planning, une vraie répartition des tâches. S'appuyer sur des outils comme Trello pour pouvoir suivre le travail de chacun. Communiquer, prendre des décisions, et travailler en équipe. Utiliser la méthode Agile simplement.

5 Conclusion

Bien que le projet ne ce soit pas déroulé comme prévu, de nombreux enseignements en sont tirés pour les futurs projets : l'importance des méthodes de travail en équipe premièrement, et de nombreuses compétences techniques avec l'apprentissage de la SDL, la structuration d'un projet, l'utilisation de git, la maitrise du C.