Rapport de projet Sandwich

BAZIN Maxence
WALTHER Dimitri

Novembre 2021 - Janvier 2022

Table des matières

1 Introduction

Ce rapport vous présente notre travail de plusieurs semaines sur le projet nommé Sandwich. Le sujet porte sur les automates cellulaires et a été réalisé en langage C. Au fil des sections, nous vous présenterons les problématiques que nous avons rencontrées et quel a été le fil de notre réflexion.

Notre but lors de ce projet a été de reproduire certain des comportements du jeu pris comme exemple du sujet : Noita

Ce rapport vous présentera notre première approche des automates cellulaires, les modifications que nous y avons apportées pour reproduire les comportements physiques de certains matériaux, les approfondissements que nous avons réalisés puis enfin les outils que nous avons développés pour faciliter la mise en place de ce projet.

2 La multicoloritude

2.1 Problématique

Dans le cadre de l'*achievement* 1, une des premières problèmatiques que nous avons rencontré était la multicoloritude. La multicoloritude avait pour thème la création d'une règle utilisant plusieurs couleurs. Elle était nécessaire à la réalisation de l'*achievement* 1.

Nous avons un peu élargi notre réflexion en nous demandant plutôt comment représenter un type de cellule, et comment un type de cellule peut être représenté par plusieurs couleurs.

2.2 Les types de cellules

Toutes les couleurs sont représentées par des nombres allant de 0 à 16777215. Un entier en c peut contenir des nombres étant plus grand que 16777215. Nous avons décidé que les types de cellule seraient définis par une énumération de numéro qui commencerait à partir de la dernière couleur affichable soit 16777216. Voilà pour les considérations techniques.

Le monde est donc composé de couleurs concrètes, mais lors de leur utilisation et de leur interprétation dans les règles, elles sont converties en type de cellule. Par exemple, la couleur 11955 et 5654564 sont des couleurs convertibles en type "STONE" de numéro 16777218.

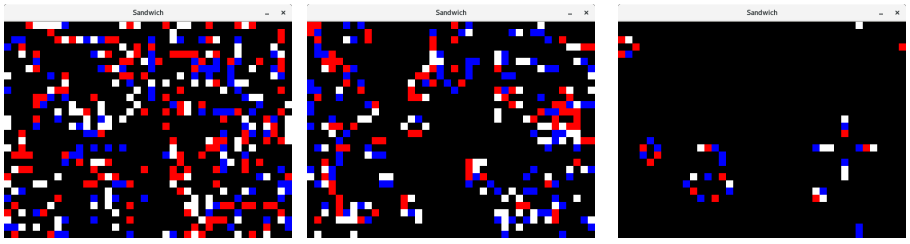


Figure 1: Jeu de la vie simulé en utilisant 3 couleurs

2.3 Le système de "last color"

Dans le cadre de la multicoloritude, un type spécial "last color", une convention, a été mis en place. Ce type n'est utile que pour les cellule qui bougent, dont le fonctionnement sera expliqué plus tard dans le rapport. Si une règle retourne la couleur "last color" quand on lui demande quelle sera la nouvelle couleur d'une cellule, alors la cellule une fois déplacée gardera la même couleur. Cette convention a son utilité quand un type est composé de plusieurs couleurs, mais qu'on voudrait que la cellule garde sa couleur.

Avec un système comme celui-ci, du bois ou des objets unis pourront être implémentés et ils conserveront leur couleur (aléatoire à l'initialisation) en se déplacent.

Figure 2: Un bloc de sable qui tombe

Figure 3: Le sable tombe en conservant ses couleurs

3 Comportement de sable

3.1 Problèmatique

Dans le jeu Noita, nous avons trouvé que le sable est un des éléments les plus intéressants à regarder comme à essayer de refaire.

Notre objectif a donc été de faire un système plus généralisé de particules qui se déplacent pour faire du sable. Un autre objectif était de trouver un moyen pour faire en sorte que des blocs de particules soient soumis à un déplacement. Afin de faire un bloc de sable qui tombe et pas seulement des grains de sable tombant petit à petit, en décalé, comme on peut le voir dans d'autres simulations.

3.2 Algorithmes

3.2.1 Modification d'une cellule à changement statique

Nous avons un premier parcours de toutes les cellules de notre monde pour regarder si des règles de modifications doivent s'appliquer. Une file de règles à appliquer sur des cellules est obtenue à l'issue de la boucle.

Ensuite, la file est défilée pour soit :

- Appliquer le changement directement si la règle concerne uniquement le type de la cellule. Pas de déplacement en somme, le changement est appliqué tout de suite.

- Effectuer ce qui suit si la règle concerne une particule.

3.2.2 Modification d'une cellule à changement dynamique

Si la règle concerne une particule en mouvement, alors des informations sont remplis pour être traité par la suite.

Sur chaque cellule, les deux informations à remplir sont :

- Une liste de changement que d'autres cellules veulent appliquer sur celle-ci.

- Une structure indiquant si la cellule veut se déplacer, et où elle veut le faire si c'est le cas.

Enfin une dernière boucle où chaque cellule est de nouveau parcourue. On regarde en premier lieu si elle est le départ d'un autre changement. Si la particule voulait se déplacer, alors ce serait son changement qu'on veut appliquer en priorité. On appelle récursivement l'algorithme sur la cellule partant de cette case.

Après avoir appelé ou non récursivement la modification à faire sur une autre cellule, on regarde si la cellule à modifier est vide. On ne veut pas perdre des cellules, on ne peut donc appliquer une modification que si la cellule est vide.

Il faut maintenant choisir la modification à faire sur la cellule, car pour rappel, nous avons une liste de changements à faire et non pas un changement unique. Un des changements sera tout simplement selectionné aléatoirement.

Au final si la modification est appliquée alors la cellule de départ de cette modification est rendue vide. C'est ce qui donne un sens aux récursions et fait que nous pouvons déplacer des blocs de particules.

Figure 4: Attention : Chute de sable

3.3 Du sable qui tombe par bloc

Pour avoir du sable qui tombe par bloc, il faut faire une recursion dans les règles pour savoir si la cellule d'en dessous tombe. Pour éviter de devoir faire des récursions inutiles, nous utilisons un cache

de sauvegarde. Le cache indique pour une cellule son état "elle tombe" ou pas. Les différents états sont "elle n'est pas encore calculé", "elle tombe" ou "elle ne tombe pas".

Le cache est donc utilisé pour sauvegarder la validation de la règle. On peut immédiatement savoir, si la cellule d'en dessous tombe ou pas, bien sûr si plus tôt dans l'exécution du programme, son état a été décidé.

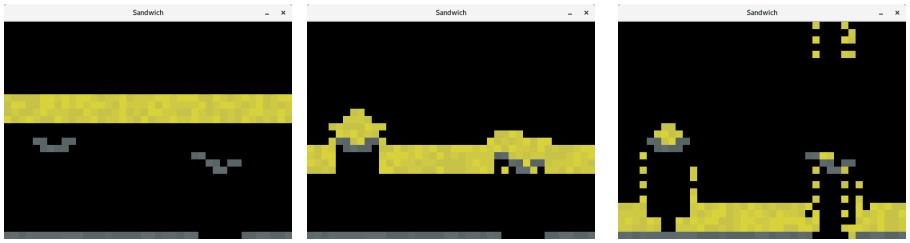


Figure 5: Le sable s'inflitre dans les trous en diagonal.

3.4 Exemple avec un test de compression

La meilleure façon de tester un système est de lui imposer un cas critique.

Soient 4 couleurs, orange, rouge, bleu et blanc. Nos cellules rouges se déplacent vers la gauche si la cellule est vide, nos cellules bleues vers la droite respectivement. Une cellule orange ne se déplace que s'il y a une cellule rouge à droite, respectivement une cellule bleue à gauche pour les cellules blanches.

Soit la configuration suivante :

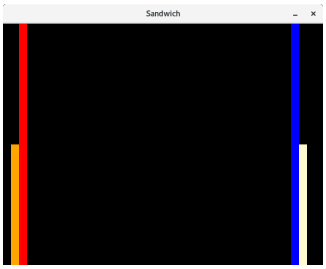


Figure 6: Configuation initiale du test.

Quand les deux colonnes sont sur le point de se rencontrer, nous voyons qu'il va y a voir une problème de compression en premier lieu. En second lieu, un problème de propagation de la résolution du conflit aux cellules oranges et blanches, pour que ces dernières n'avancent que si la cellule qu'elle suive a été celle choisie dans la résolution du conflit.

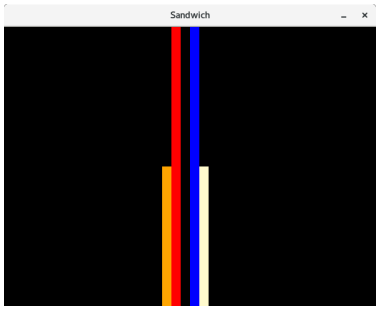


Figure 7: Un évènement dramatique va se produire.

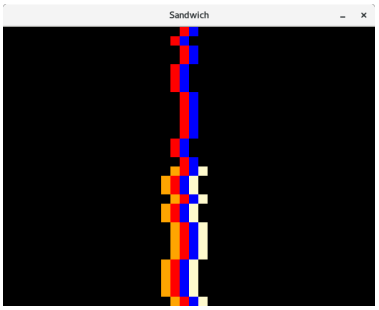


Figure 8: Mais la situation est gérée correctement.

4 Colonne complète en sables

4.1 Problématique

Pour réaliser l'*achievement* 5, nous avons remarqué qu'un cas pouvait casser notre simulation. Une colonne de sable, qui provoquerait une récursion infinie dans les tests des règles, et une deuxième récursion infinie dans l'application des modifications pour les particules.

4.2 Algorithmes

4.2.1 Résoudre les cycles dans les règles

La règle du sable teste récursivement si la cellule d'en dessous tombe, pour décider si elle-même tombe. S'il y a une colonne remplie de cellules qui peuvent tomber, il y a une boucle infinie. La solution choisie est l'utilisation du cache, initialisé en début de la simulation de l'image, pour repérer la récursion, la briser, et enregistrer le fait que toute la colonne tombe.

4.2.2 Résoudre les cycles dans le déplacement des particules

De la même façon que pour les boucles dans les tests de règles. Un cache est utilisé pour sauvegarder la récursion en cours. Quand une boucle est repérée, elle est brisée et toutes les particules sont déplacées.

Figure 9: Le sable rempli tout l'espace.

Figure 10: Il est quand même entièrement déplacé.

5 Evènements aléatoires

5.1 Problématique

Sans lien avec les *achievements*, nous voulions faire des évènements aléatoires en utilisant le cache à notre avantage.

5.2 Exemples

5.2.1 Explosions

Dans le cache, nous avons ajouté deux informations en plus de celles utilisées pour les cellules qui tombent. Pour chaque cellule :
Une probabilité moyenne "fausse" quand la règle ne doit pas passer et "vraie" quand elle le doit. Et une information dans un état "non calculé" par défaut, qui indique si la cellule est en train d'exploser.

La règle explosion demande au cache l'état de cette cellule. Si l'état n'est pas indéfini alors la règle fini instantanément en retournant cet état, sinon elle va lancer une recherche sur toutes les cellules.

La recherche regarde s'il y a une cellule explosive, et est ce que la probabilité moyenne associée dans le cache est vrai. Si l'explosif est censé exploser sur la *frame*, elle regarde toutes les cellules dans un rayon autour de l'explosif. Si la cellule est un matériau explosable alors la cellule devient "détruite" par l'explosif et son état est noté dans le cache.

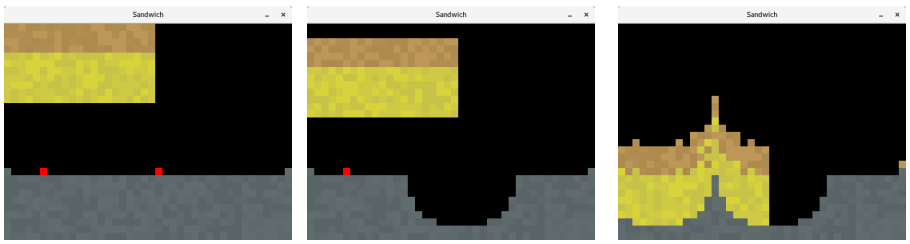


Figure 11: Les explosifs explosent à un moment aléatoires.

Notre comportement d'explosion permet d'illustrer l'utilité du cache pour enregistrer des états complexes obtenus suite à un événement aléatoire. Grâce au cache, il n'y a pas besoin de recalculer la

présence d'un explosif, s'il explose effectivement, et si la cellule est dans le rayon, pour chaque cellule pouvant être affectée par une explosion.

Le cache permet même aux autres règles de prendre en compte des états prioritaires en considération. Par exemple, une cellule de sable pourrait tomber instantanément si la cellule en dessous est notée détruite dans le cache.

5.2.2 Sable dense

Notre idée ici, était de continuer sur l'utilisation du cache pour modéliser un état intéressant. Faire du sable dense qui tombe petit à petit en dessous du sable *normal*.

Pour faire ce comportement, nous avions besoin de deux règles, une pour transformer le sable dense en normal et inversement pour donner l'illusion que les cellules échangent de place. À noter que nous aurions pu considérer les deux cellules comme deux particules et laisser notre système gérer la boucle infinie de déplacement pour échanger les deux particules de place. Mais nous n'avions pas encore mis ce système en place.

La question était la suivante : comment faire en sorte que les deux règles décident en même temps, qu'elles doivent être vraie en incluant de l'aléatoire ?

Encore une fois, nous avons réglé ce problème en utilisant le cache et la probabilité moyenne. Les deux règles regardent si le grain supérieur de la transformation a une probabilité moyenne vraie associée dans le cache. Et dans ce cas-là, les deux règles retournent vrai en même temps.
Le résultat est satisfaisant à regarder et plutôt réaliste. C'est un comportement qui montre encore une fois l'étendue des choses qui pourraient être faites avec le système de cache.

En y ajoutant un peu d'aléatoire, cela rendrait les systèmes un peu plus réalistes, Par exemple une cellule de feu qui aurait une probabilité de se propager ou de s'éteindre.

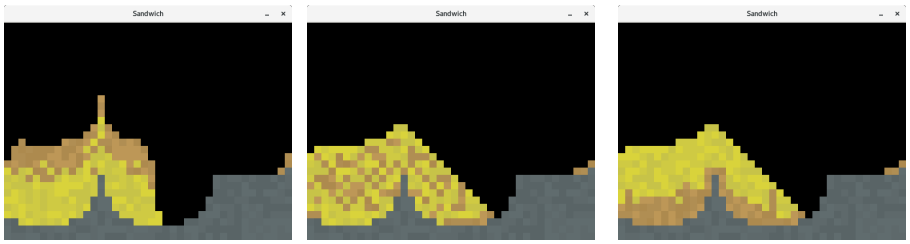


Figure 12: Le sable dense tombe petit à petit au fond du tas de sable.

6 Les "plus" de notre programme

6.1 Préambule

Nous avons fait en sorte que l'utilisation du programme soit agréable pour celui qui voudrait coder des règles où bien pour celui qui voudrait utiliser le programme en sortie de compilation.

6.2 Les arguments de compilation

À la compilation certaines choses peuvent être paramétrées. La hauteur et la largeur du monde évidemment. L'ensemble de régles peut aussi être choisi.

Dans notre système plusieurs ensembles de règles sont choisisables, par defaut c'est l'ensemble de règle "réaliste" qui est choisi. Nous avons un emsemble de règle pour le jeu de la vie et un emsemble de règle pour tester la compression et la perte de cellules. Un futur codeur pourrait rajouter des ensembles de règles s'il le voulait.

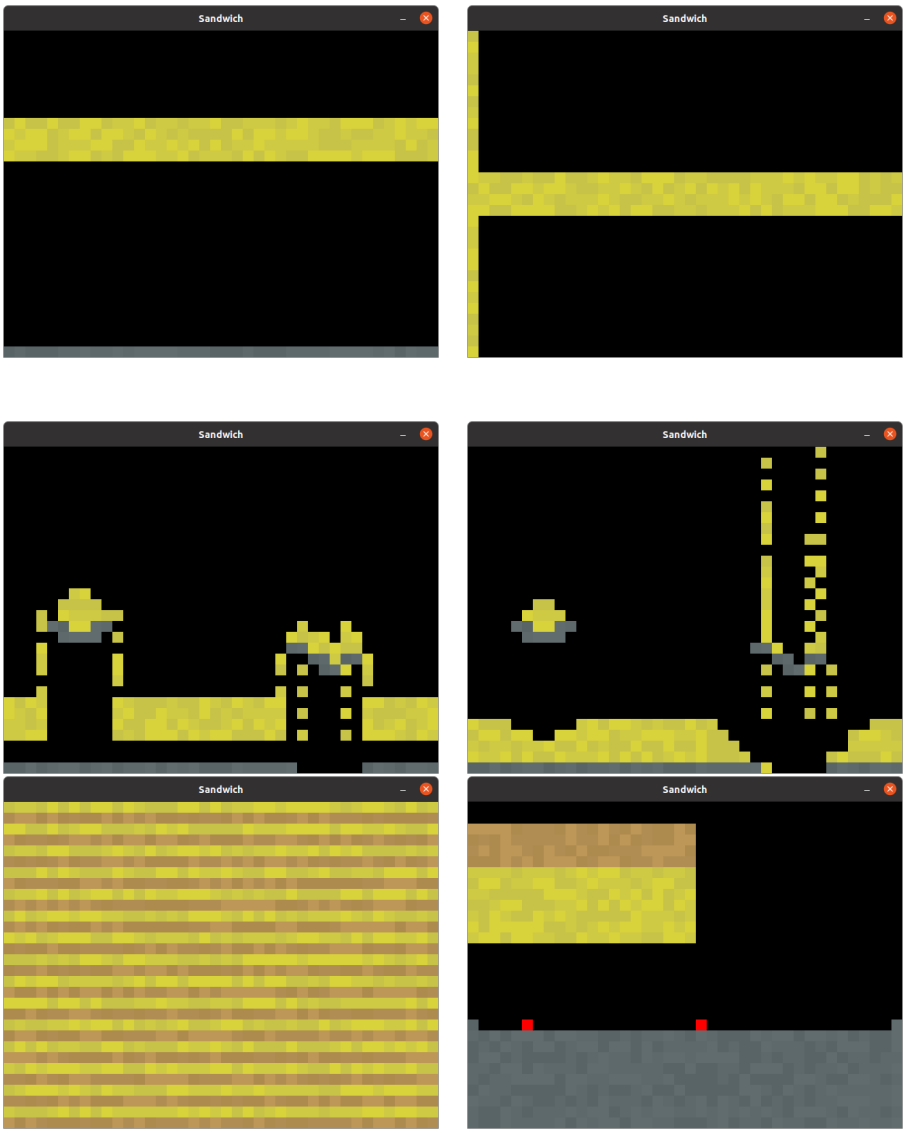
6.3 Les arguments de programme

Tout d'abord, l'utilisateur peut ajouter l'argument "-h" pour que le programme lui affiche directement l'aide.

Nous avons des arguments pour choisir le nombre d'images qui seront générées, la graine de l'aléatoire, le monde à charger dont nous expliquerons le fonctionnement juste en dessous et le choix du mode panel.

Le panel de monde est un mode de lancement spécial, qui ne tient pas compte du nombre d'images ou du fichier à charger donné en argument. Ce mode va charger et simuler tous les fichiers que le codeur aura spécifié dans le code.

Dans l'état actuel du programme, 5 mondes sont chargés et simulés les uns après les autres.


Sandwich
Sandwich
Sandwich
Sandwich
Sandwich
Sandwich

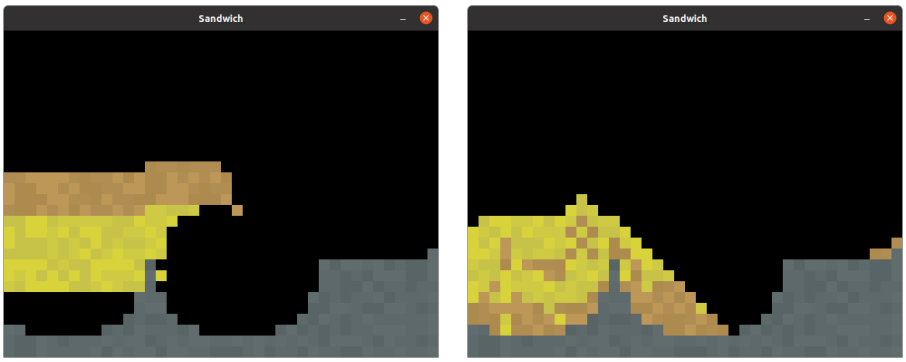


Figure 13: Florilège de simulations affiché par la commande –all

6.4 Chargement de fichier

6.4.1 Problématique

Au début de notre projet, il était assez difficile de changer la configuration d'un monde. À la seule exception du jeu de la vie qui générait une configuration initiale aléatoire, nous devions créer nous-mêmes chaque monde cellule après cellule. Ce travail long et fastidieux nous a donné la motivation de trouver des fonctions afin de nous aider.

6.4.2 Qu'est-ce qu'un monde et comment le générer rapidement

Le premier choix a été de générer les mondes à l'aide de types puis de changer les couleurs en suivant. Dans notre tableau *world*, chaque case contient donc un entier qui correspond au type de la cellule. Ce choix nous a simplifié la tâche et nous a permis d'approfondir tôt l'aspect graphique filtrages de notre projet. Nous avions dès lors des nuances de couleurs pour chaque matériaux. Pour réaliser ces changements, des fonctions de conversions entre couleurs et types ont créé.

La deuxième étape de notre simplification a été de mettre en place des fonctions permettant de générer rapidement des mondes ayant la configuration désirée. Un des outils mis en place a été un générateur de blocs d'un type donné. Nous avons dès lors pu expérimenter sur le sable en faisant tomber des blocs de ce type sur un sol de pierre (comme on peut le voir sur la figure 3).

6.4.3 Sauvegarder et charger des mondes à volonté

À l'aide de toutes ces fonctions, il était possible de créer des mondes simples assez facilement, mais il était beaucoup trop compliqué de créer des structures plus complexes comme celle de la figure 5. De plus, passer d'une configuration à une autre était fastidieux et frustrant lorsqu'on alternait entre deux configurations de manière régulière. La solution a été de créer un système pour sauvegarder des configurations et les charger dans des mondes à volonté.

Le format le plus adapté est le fichier texte car, il est possible de le manipuler facilement et de modifier manuellement les cellules désirées. Pour sauvegarder un fichier, il suffit d'écrire le type de chaque cellule (c'est-à-dire un entier) et de séparer ces entiers par des espaces, comme ils le seraient dans la sortie standard lors de l'exécution. Afin de charger un fichier, il faut donner le chemin vers le fichier texte en question. L'algorithme va ensuite lire chaque type et le reporter aux coordonnées correspondantes dans le monde.

7 Méthotodologie et résultats des tests

Tester les programmes est une étape importante du projet.

Le bon fonctionnement des tests assure la correction des programmes et une compréhension profonde des concepts que l'on manipule ainsi que des faiblesses du code. Afin de cibler les problèmes, les tests ont été répartis en trois "groupes", chacun testant les fonctions propres à chaque règle.
Chaque fonction de test fait appel à la bibliothèque `assert.h`, de cette manière, le programme renvoie directement la cause de l'erreur et le test au cours duquel elle s'est produite.

7.1 Les fonctions utilitaires

Le projet repose sur la structure de files pour choisir quelles règles sont à appliquer. Il était donc essentiel de vérifier que cette structure fonctionne bien. Des tests ont été réalisés afin de s'assurer que la file est bien initialisée, qu'il est possible d'ajouter et d'enlever des éléments. On s'assure aussi qu'il est impossible d'ajouter un élément à une file pleine ou de retirer un élément à une file vide.

Une fonction importante est celle qui permet de donner les prochaines coordonnées d'une particule en fonction de sa position et d'un vecteur de déplacement. Notre méthode pour tester cette fonction a été de comparer les résultats de la fonction aux résultats théoriques. La fonction a été testée sur des coordonnées classiques et sur des cas pathologiques (les bords du monde).

Un certain nombre de fonctions implémentées ne sont pas nécessaires à la validation de nos objectifs, mais nous facilitent grandement la tâche. Il était crucial de vérifier que ces fonctions soient correctes, car une erreur sur ce genre de fonctions serait difficile à repérer (il est plus facile de penser que le problème vient d'une fonction ayant une plus grande importance). Nous avons donc réalisé des tests sur toutes les fonctions en rapport le chargement et la sauvegarde de mondes pour s'assurer qu'elles fonctionnent et qu'il est impossible de les faire fonctionner avec de mauvais arguments.

7.2 Tests de compressibilité

La compressibilité est le problème le plus important pour notre simulation. En effet, la compressibilité (ou la conservation du nombre de particules) peut être compromise par une mauvaise gestion des conflits entre particules et par une mauvaise gestion des cycles. La vérification de la compressibilité est toutefois simple à mettre en œuvre puisqu'il suffit de compter le nombre de cellules d'un certain type au début de la simulation puis de vérifier que ce nombre reste le même à chaque étape. Nous avons décidé de créer un environnement avec une configuration et des règles spéciales pour vérifier la compressibilité. Des tests similaires ont ensuite été réalisés sur la simulation pour vérifier que le problème était bien résolu.

7.3 Tests globaux

Nous avons séparé ces tests en trois catégories : les tests sur la file, les tests sur la simulation et enfin les tests sur la compressibilité. De cette manière, il était possible de travailler sur un aspect du projet et de lancer les tests de façon régulière pour s'assurer d'avancer dans la bonne direction. Cette méthode nous assure de travailler sur une base de code saine sans pour autant tester toutes les fonctions à chaque fois. En effet, la structure de file et toutes les fonctions y étant associées ont très peu changé depuis le début du projet et les problèmes de compressibilité ne dépendaient pas des fonctions utilitaires.

8 Conclusion

Notre but lors de ce projet a été de reproduire certains des comportements du jeu pris comme exemple du sujet : Noita.

Nous avons réussi à faire un comportement de sable réaliste, des explosions, et une règle de sable aléatoire assez intéressante.

Nous aurions aimé coder plus de règles, pour obtenir une simulation plus complète, mais nous nous sommes plus concentré à obtenir un système complet.

Avec ces semaines de projets, nous avons créer un système assez robuste, qui permet de créer facilement des comportements complexes comme de l'eau ou du feu.

9 Annexe

D'après la figure 14, ci dessous, nous pouvons dire plusieurs choses.

En représentant notre code, nous pouvons le séparer en 5 catégories. Les fichiers intra-catégories ont beaucoup de dépendances entre eux et il n'y pas tellement de dépendances entre les catégories ce qui reflètent une bonne compartimentation du code.

Au-dessus de tout ça se trouve *celltype*, la définition des types de cellules et les méthodes utilitaires pour les manipuler.

Tout le code concernant les règles peut être conceptuellement regroupées ensemble. Les règles fonctionnent par elle-même, et ne dépendent pas du code gérant la simulation globalement.

Notre implémentation de liste se trouve dans une catégorie *tools*, dans le sens où si l'implementation était différente, le code en dépendant ne changerait pas fondamentalement.

La catégorie *world* représentant le code permettant de manipuler le monde.

Le code gérant la simulation est dans sa propre catégorie *simulation*, elle fonctionne quelque soit les règles par exemple.

Projet, le fichier principal est l'interface entre l'utilisateur et la simulation, c'est elle qui s'occupe des arguments par exemple.

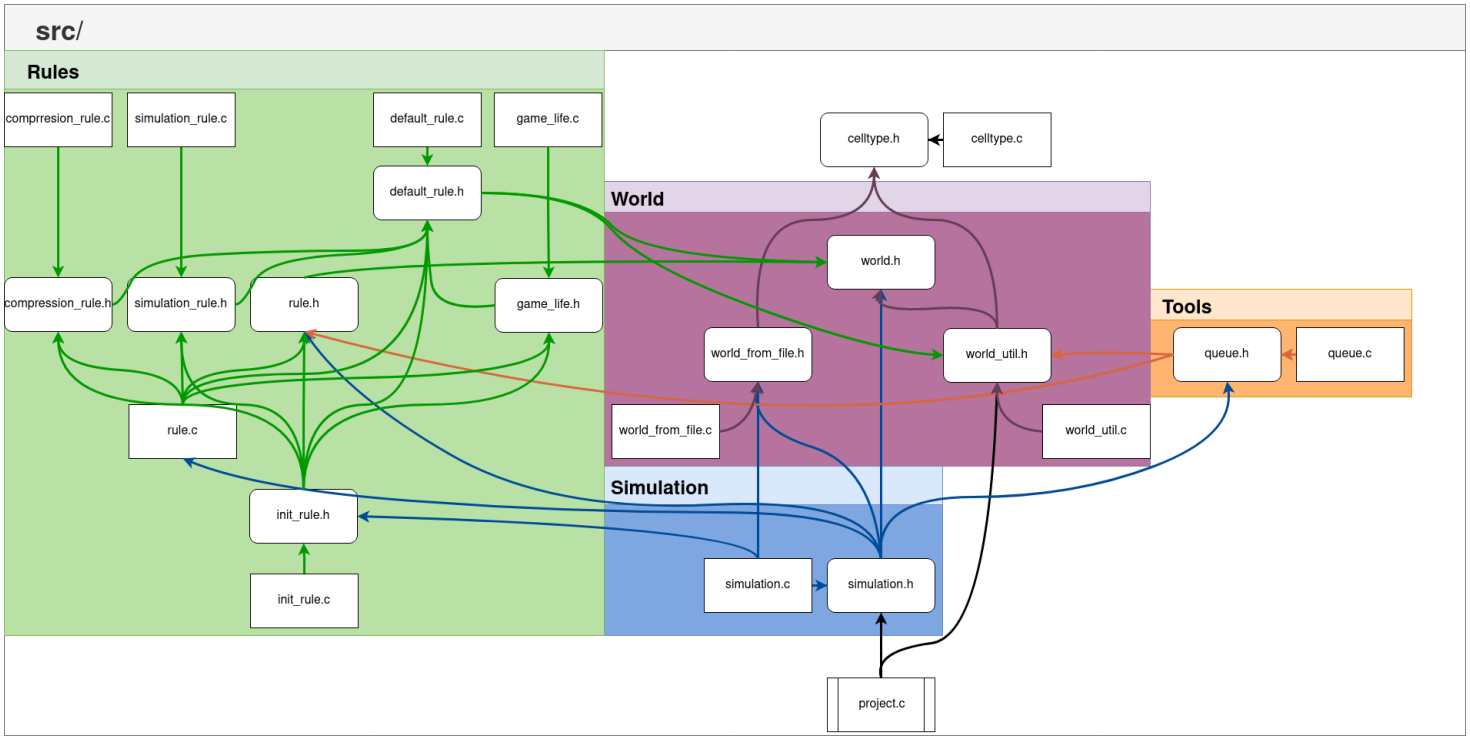


Figure 14: Graphe des dépendences des fichiers