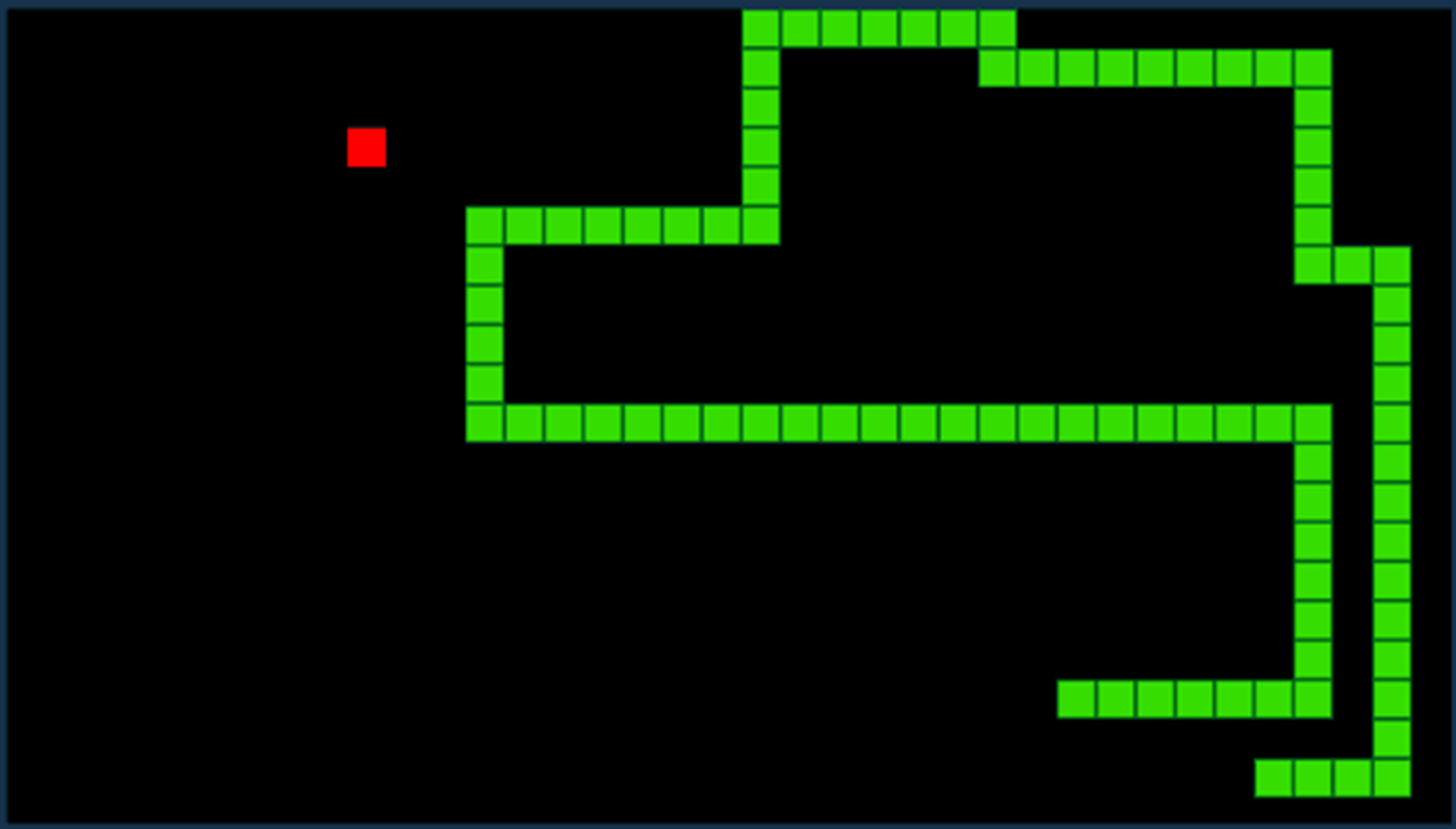
PROJET 1.1 - APL 2019

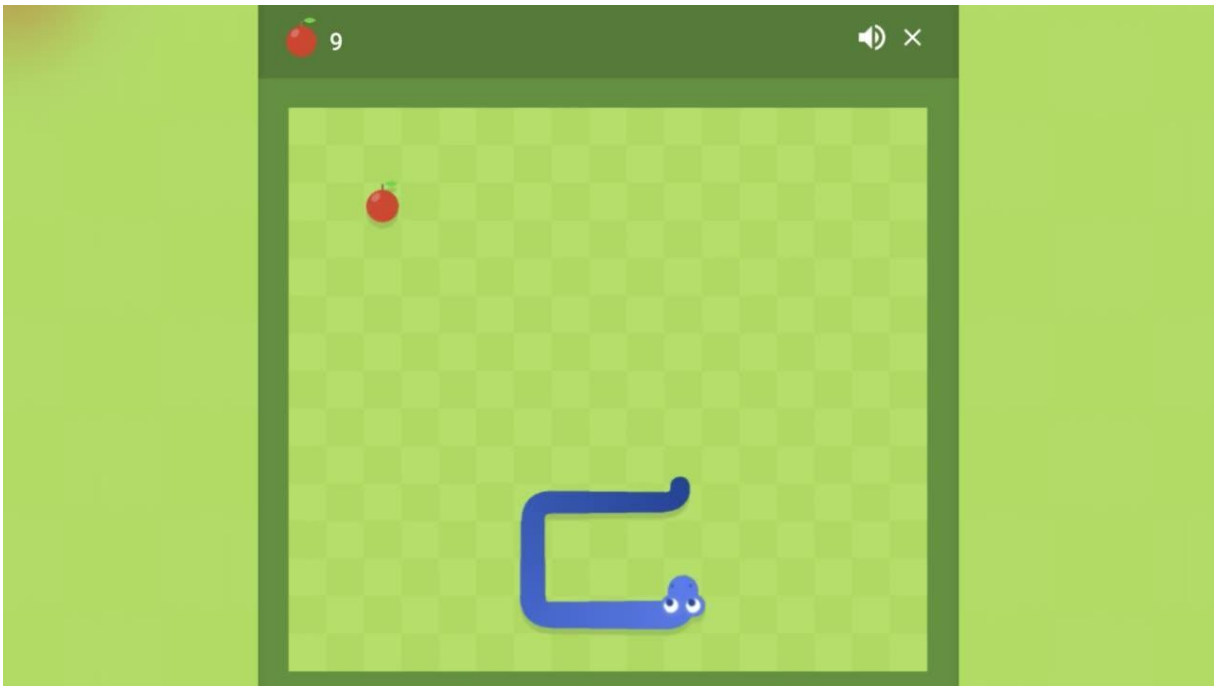
SNAKE



20 DÉCEMBRE 2019

VASILE CIOCOIU
DECORBEZ NICOLAS

Table des matières



Exemple d'un Snake développé par Google

Introduction

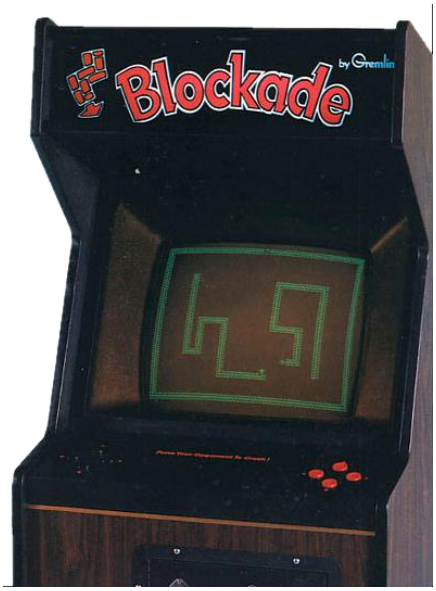
Ce projet a pour but de réaliser un Snake. Ce jeu, inspiré du jeu vidéo d'arcade Blockade (développé et édité par Gremlin Industries en 1976), ne possède pas de version « standard ». Le concept est simple : le joueur dirige une ligne (le serpent) qui grandit et constitue un obstacle. Il doit manger des pommes pour grandir et augmenter son score, et il perd s'il touche les bords ou se mord la queue.

Le jeu connu une certaine popularité quand Nokia, en 1998, a intégré le jeu dans ses produits (téléphones mobiles notamment). C'est donc devenu à la fois un classique, mais aussi un précurseur des jeux sur mobiles : en effet, sa jouabilité est la portée de tous et il est très simple d'y jouer à n'importe quel moment de la journée.

Peu à peu, il s'est démocratisé et de nombreux clones sont apparus ; on ne note pas moins de 8 clones développés par Nokia pour leurs téléphones. Ensuite, il fut développé pour les navigateurs, calculatrices mêmes, jusqu'à devenir un exercice de programmation, comme ici.

Exemple de Snake sur un Nokia 3310

Nous avons donc voulu rappeler cet esprit très « retro » avec notre Snake. Pas d'images, un design très basique, un peu à la manière de Blockade en apportant une petite touche de modernité.



Jeu d'arcade Blockade (1976)

Codage du Snake

Découverte de la bibliothèque graphique et de la gestion des évènements

C'est en C que nous devions programmer ce jeu.

Nous n'avions jamais codé avec une bibliothèque graphique en C. Il nous a fallu découvrir toutes les options disponibles petit à petit afin de nous forger une prière idée de comment nous allions coder notre projet. Nous avons donc créé une multitude de petits programmes afin de nous familiariser avec cette bibliothèque, du simple InitialiserGraphique() jusqu'au gestion des événements du clavier. Nous n'avons pas voulu gérer les événements de la souris dans un soucis de style : nous voulions garder l'esprit arcade, où on peut jouer directement au jeu sans régler une multitude de paramètres.

Il nous a fallu malgré tout un léger temps d'adaptation avant de bien comprendre la sémantique de chaque fonction à utiliser et de faire le tri parmi toutes celles proposées dans la documentation. Finalement, nous sommes partis sur un concept très basique : un espace de jeu de 600px par 400px, ce qui nous fait des cases de 10px de côté pour un espace de jeu en 60x40 cases.

Les choix des couleurs furent arbitraires : un rouge bordeaux pour le fond, le blanc se distinguait bien pour afficher le texte, le serpent et les bords de l'espace de jeu. A l'inverse du blanc, du noir pour afficher les pommes, et enfin du jaune pour notre obstacle.

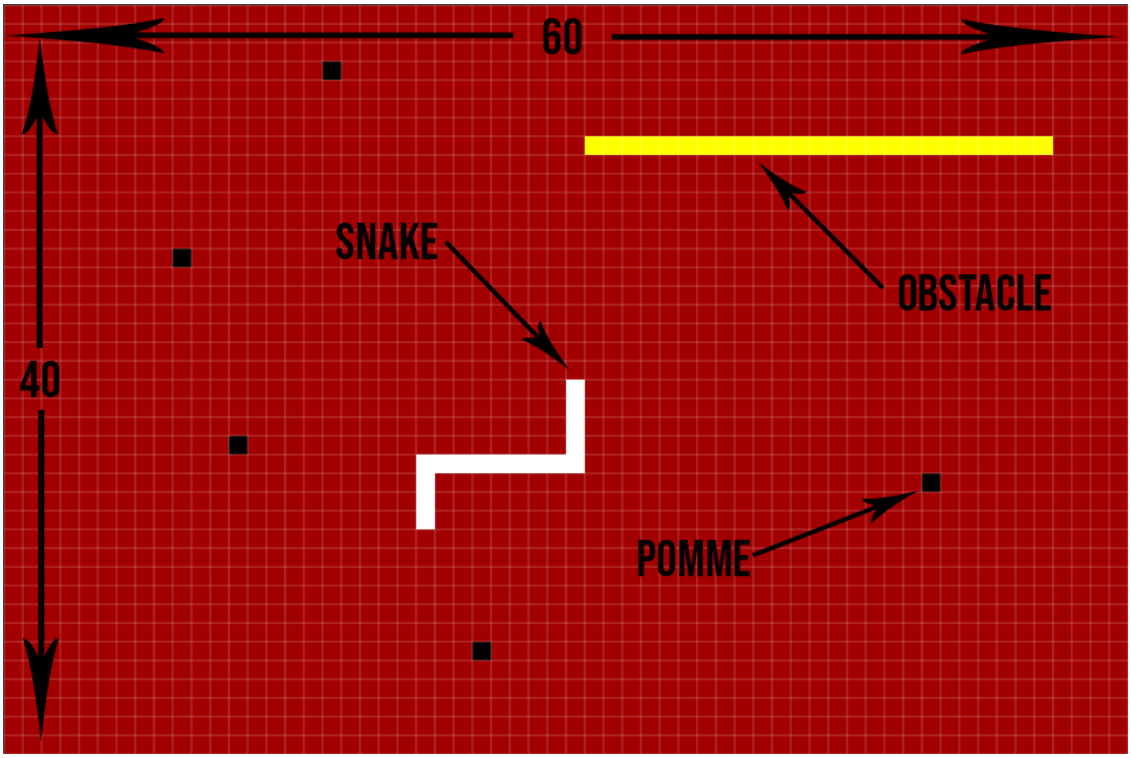
Premier code du Snake

Dans un premier temps, nous avons décidé de coder le Snake dans un seul grand programme, afin d'être sûr qu'il soit bien fonctionnel en excluant des erreurs dues à des fonctions mal déclarées, par exemple. Nous nous sommes rendu compte de plusieurs bugs, notamment graphique (segments de serpent isolés, etc.) et nous avons donc dû corriger ces erreurs au fur et à mesure que le code avançait.

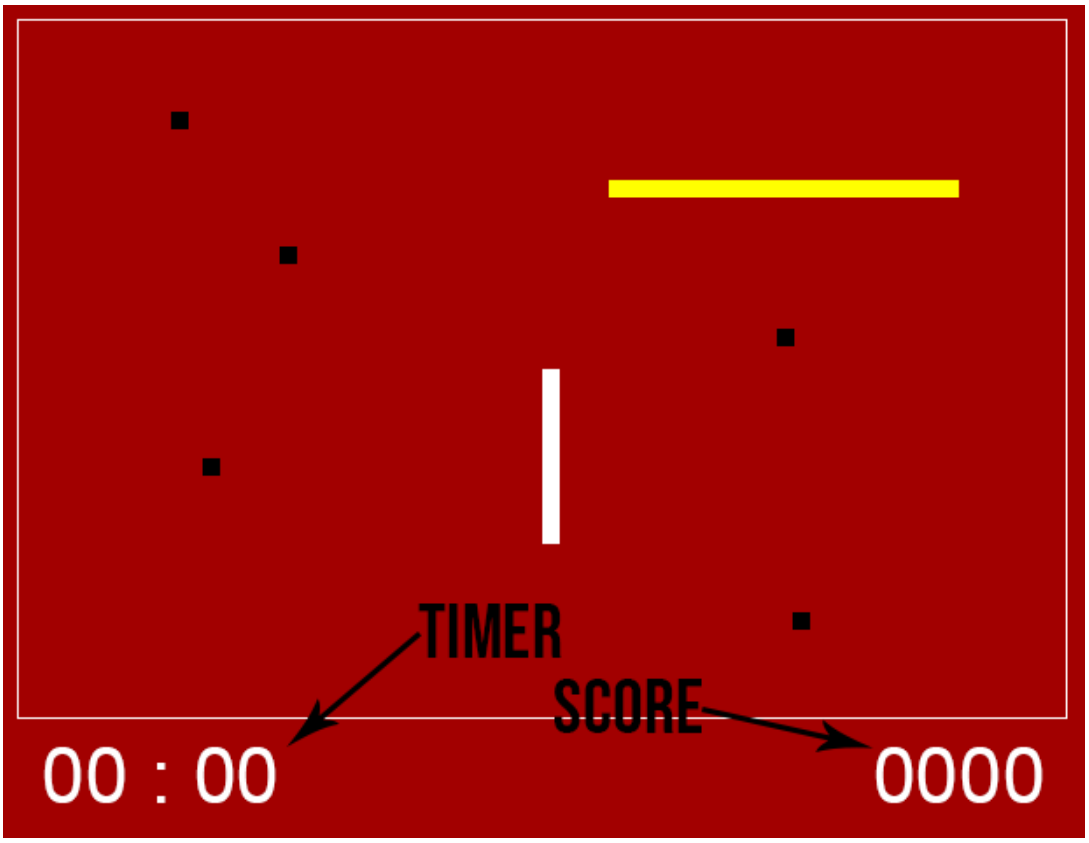
Malgré ce grand code (de plus de 500 lignes, quand même), nous faisions en sorte de trier les fonctions par type afin de nous y retrouver pour plus tard, beaucoup de commentaires nous ont aidé au moment du découpage en différentes fonctions.

Enfin, nous avons pensé à l'optimisation du programme en réduisant au maximum les variables, et en utilisant des allocations dynamiques pour la taille du serpent par exemple, plutôt que de déclarer un tableau de 2400 valeurs. Nous avons aussi rencontré des problèmes avec la fonction realloc(), qui nous créait des erreurs de segmentations diverses et variées ; nous avons donc dû contourner ce problème avec des free() et de nouveaux calloc().

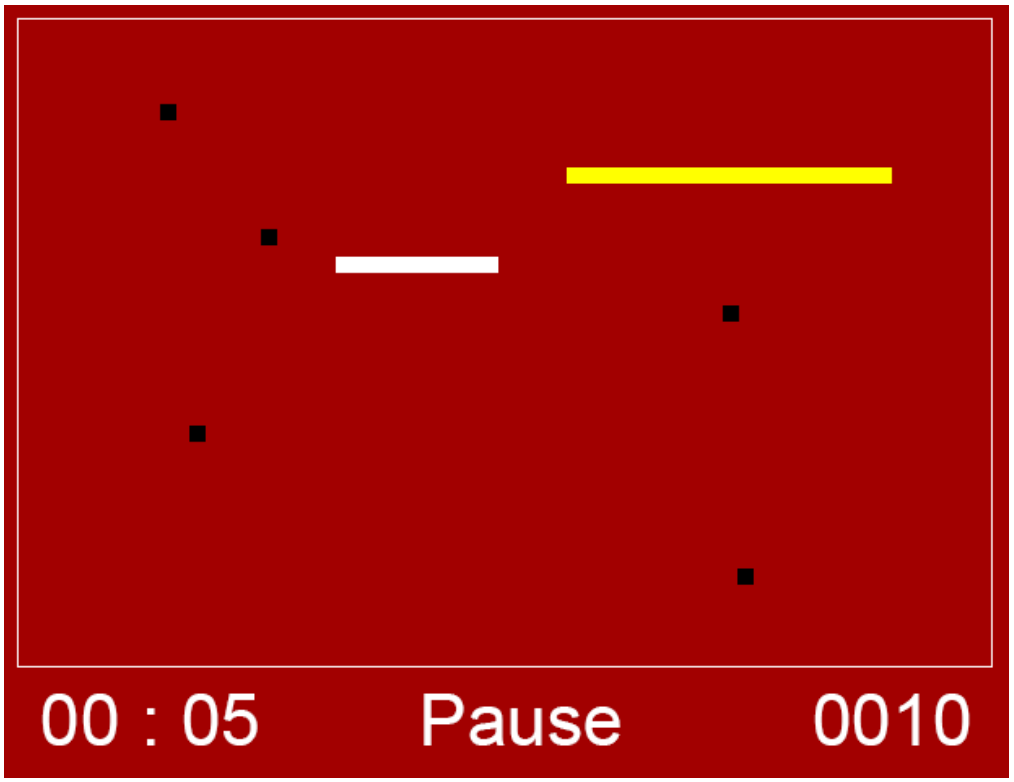
En option supplémentaire, nous avons choisi de créer un obstacle qui apparaît aléatoirement sur la zone de jeu.



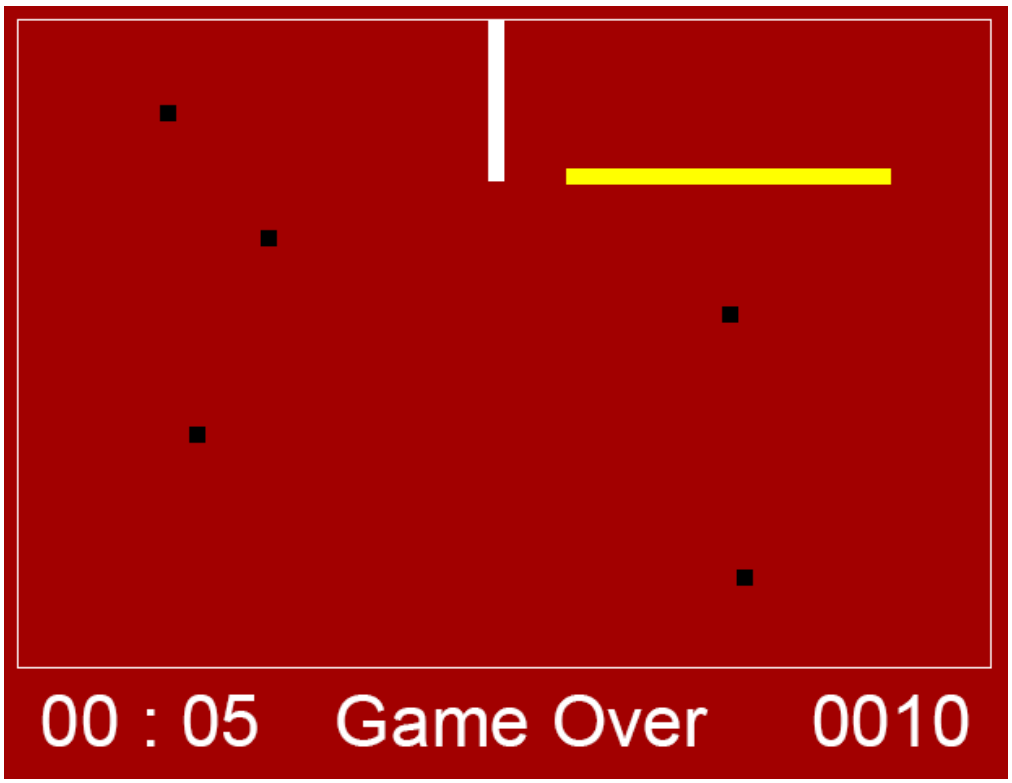
Capture d’écran de la zone de jeu



Affichage final de la fenêtre de jeu



Fonction pause



Fonction Game Over

Découpage du programme en fonctions

Mise en place des fonctions et tri

Nous avons donc, une fois le programme fonctionnel et fini, voulu le découper en plusieurs fonctions afin de le rendre plus lisible et d'éviter la répétition de certains bouts de code (par exemple, l'affichage du « Game Over » ou l'affichage d'une nouvelle pomme)

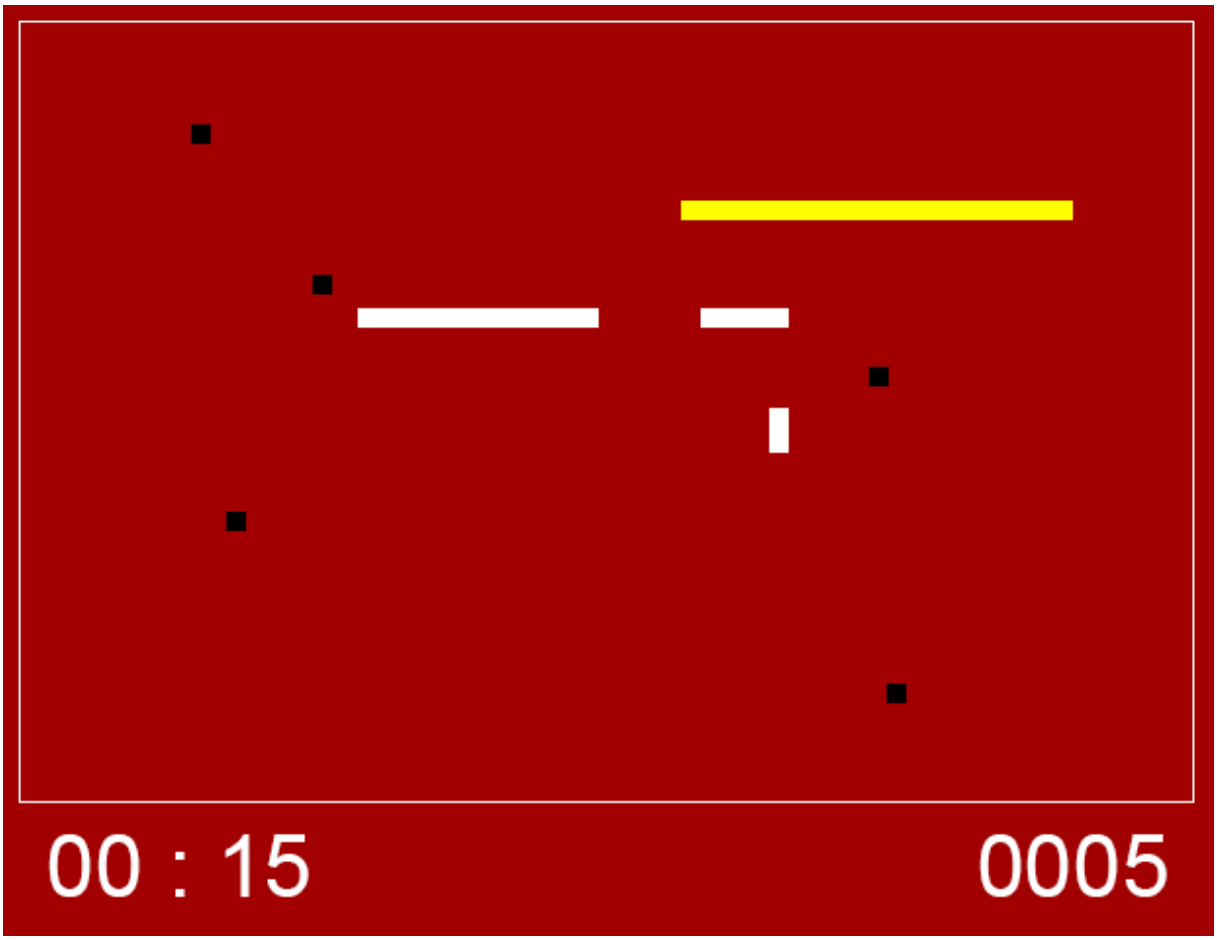
On a éprouvé pas mal de problème à ce moment-là du projet ; en effet, autant pour coder le Snake nous n'avons pas eu beaucoup de soucis, autant le découpage du programme nous a pris plus de temps que prévu. De nombreuses fonctions rentraient en conflit, et notre connaissance limitée au niveau des fonctions nous n'ont permis de réduire le main.c qu'en dessous des 200 lignes. Néanmoins, nous avons pu créer une quinzaine de fonctions afin de simplifier notre code, regroupées dans 3 fichiers différents : initialisation.c (qui contient l'initialisation de la fenêtre, des pommes, du serpent et de l'obstacle), jeu.c (qui continent plein de petites fonctions, comme l'affichage du « Game Over » ou du timer) et serpent.c (qui contient plusieurs fonctions en rapport avec le serpent, comme par exemple les collisions).

Nous avons également essayé d'ajouter des commentaires pour chaque « paragraphe » de nos fonctions et de notre main, toujours dans un souci de lisibilité.

Réécriture du main.c

Là aussi, nous avons rencontré quelques problèmes : nous n'avons que moyennement réussi à tout découper sans obtenir beaucoup de bugs et d'erreurs ; c'est d'ailleurs pour ça que nous avons un main assez long, nous avons préféré un Snake fonctionnel et optimisé avec un main long, plutôt qu'un Snake plein de bugs avec un main court.

Malgré ça, nous avons tous les deux beaucoup appris par ce biais : le découpage en fonctions restait assez flou jusqu'à ce projet, qui nous a permis de vraiment approfondir cet aspect du codage.



Bug récurent durant le découpage des fonctions

Explication du code

Grille et tableaux de pointeurs

Pour le codage de notre jeu, nous avons choisi de créer un tableau multidimensionnel, nommé grille[60][40] initialisé à 0 pour toutes les valeurs. Ensuite, il a fallu choisir des valeurs pour représenter chaque type de case possible : 1 pour les pommes, 3 pour l'obstacle, et 5 pour le serpent. Ainsi, il nous suffisait de tester si les coordonnées de la tête (int x, int y) rentrées dans la grille était égal à un espace vide (0), une pomme (1), etc. Pour la gestion des collisions, c'est ce qui nous a paru le plus simple.

Les coordonnées du corps du serpent sont stockées dans deux tableaux de pointeurs (int* xQueue, int* yQueue). La taille du serpent est gérée par la variable int taille, qui s'incrémente de 2 à chaque pomme mangée. Ainsi, les tableaux sont alloués dynamiquement avec la fonction « xQueue = (int*) calloc(taille, sizeof(int)); » afin de ne pas utiliser trop de mémoire inutilement.

Les tableaux de pointeurs vont de [taille-1] à [0]. Une fois arrivé à 0, grille[xQueue[0]][yQueue[0]]=0 afin de rétablir un espace vide après la queue du Snake. Aussi, on utilise ces coordonnées afin d'afficher un carré de la couleur du fond, pour effacer derrière le serpent.

Les pommes sont affichées avec la fonction nouvellePomme(int xp, int xp, int grille[60][40]). Les coordonnées sont générées aléatoirement grâce à la fonction « srand(time(NULL)) » puis rand(). Les coordonnées sont rentrées dans grille, codées 1.

Quand une pomme est mangée, les coordonnées de la tête sont rentrées dans grille pour réinitialiser la valeur à 0, le score augmente de 5, et la taille s'incrémente de 2.

La fonction serpentCollision() teste les coordonnées de la tête en les rentrant dans grille. Elle continent 6 conditions qui renvoient chacune la fonction gameOver(), qui elle affiche le « Game over » en cas de collision avec l'obstacle, la queue ou les bords de la zone de jeu.

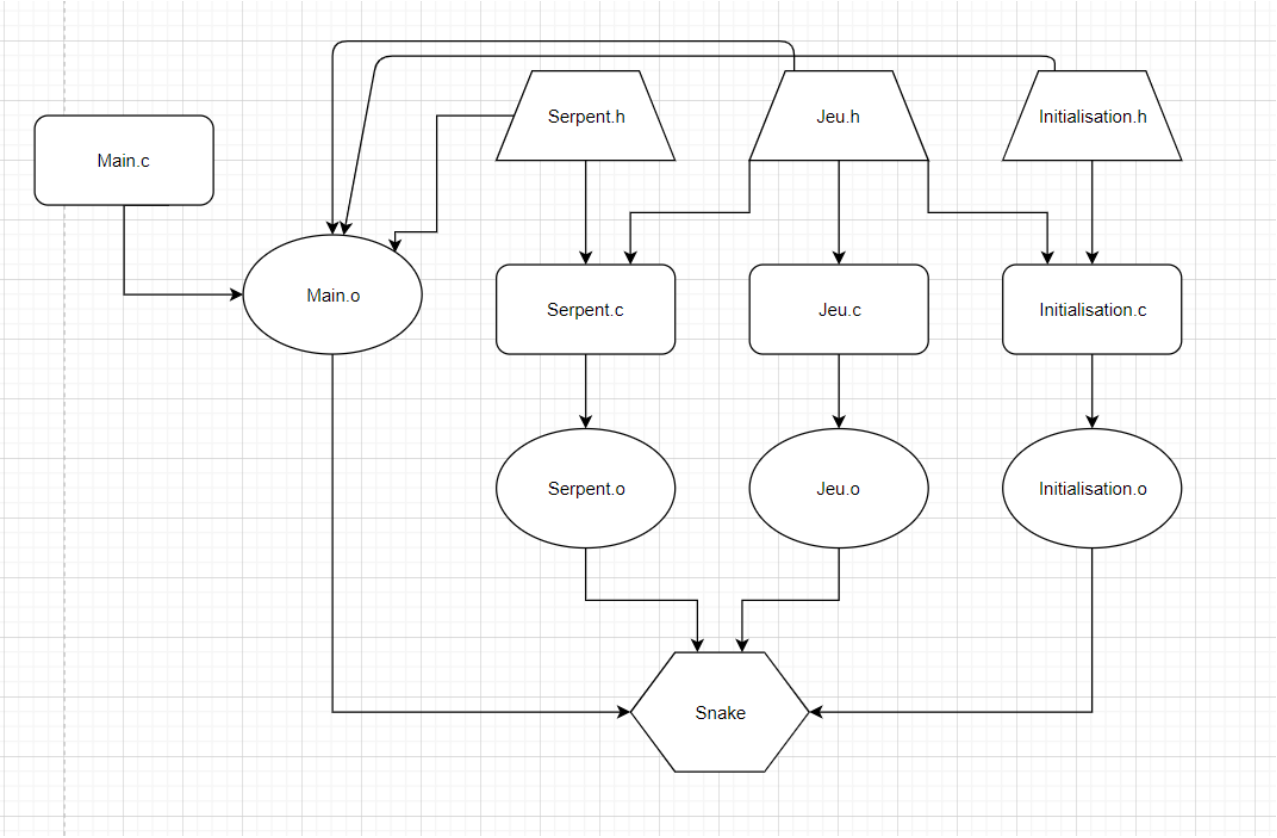
Autres fonctions

Le reste des fonctions sont majoritairement des fonctions d'affichage, comme jeuTimer() ou jeuScore, qui transforment des entier en chaine de caractère avec sprintf() afin de les afficher grâce à EcrireTexte()

Structure du programme

Le programme est structuré simplement. Il n'y a que le fichier serpent.c qui nécessite jeu.c pour compiler correctement (à cause de la fonction gameOver()). Sinon, tous les fichiers sont indépendants (comme on peut le voir dans le Makefile)

Voici un diagramme qui représente la structure du programme :



Conclusions personnelles

Nicolas :

J'ai été un peu surpris par la charge de travail qu'à demander ce projet. En effet, avec l'apprentissage des fonctions de la bibliothèque graphique et la gestion des évènements, je me suis rendu compte que coder ce jeu allait nous prendre plus de temps que prévu. Malgré son allure un peu « simpliste », il en reste néanmoins un peu complexe, surtout suite aux différents bugs auxquels nous avons dû faire face.

Mais ce fut amusant de coder un jeu, ça nous change des petits programmes de TP, et j'ai beaucoup appris au niveau des fonctions, notamment, car cela restait une notion assez complexe et abstraite pour moi jusque-là. J'ai beaucoup appris à travers ce Snake, tout en éprouvant un certain plaisir à le voir évoluer petit à petit, de l'idée de base, le design, le code, puis corriger les bugs jusqu'à son achèvement final.

Vasile :

Ma contribution à ce projet se compose principalement de petites fonctions de vérification. Elles ont comme objectif d'assurer le bon fonctionnement du jeu. De plus, j'ai contribué à l'élimination de différents bugs survenus lors de la création de ce jeu. En tout cas, ce projet a été pour moi l'occasion de rattraper mon retard sur Makefile et, surtout grâce à Nicolas, sur les fonctions et comment celles-ci devraient être conçues. Vu que l'année dernière j'ai fait un travail en binôme similaire pour l'épreuve de bac de l'ISN, cette fois n'était qu'un rappel de l'importance de la planification et d'une bonne communication.

Commune :

Nous avons tous les deux appris beaucoup de choses, tout en renforçant nos acquis et corrigeant certaines lacunes. De plus, ce fut amusant de créer ce jeu, à deux, et ainsi de s'organiser en groupe afin de se répartir les tâches, mettre en commun les bouts de code et avancer plus vite et efficacement que si nous étions seuls. Nous avons le sentiment d'avoir apporté à l'autre là où nous somme plus à l'aise avec le langage C.