BELHAJ Amine
BUI Michel
TP6 & TP7

RAPPORT SAE13.2 2023 - Mini-projet assembleur

Sommaire

Introduction

Ce projet de SAE 13.2 porte sur la programmation du fameux jeu du Snake en assembleur MIPS32. En effet, ce projet reprend des notions de S12 que nous avons vues en cours afin de les mettre en pratique grâce au sujet. Afin de mener à bien le projet, notre binôme est donc composé d'Amine BELHAJ du groupe de TP6 et de Michel BUI du groupe de TP7, car nous avons déjà fait des projets de programmation ensemble. Ensuite, pour préciser, cet SAE 13.2 nous aura occupé au

total toutes les séances de SAE réservé à celui-ci ainsi que plusieurs heures de travail à domicile notamment grâce à la plateforme de communication : Discord. La répartition du travail est plutôt homogène, c'est-à-dire que toutes les fonctions à coder, nous les avons faites ensemble en même temps. Le jeu du Snake a donc pour objectif d'avoir un serpent avec la taille la plus grande possible en mangeant les pommes sans mourir. Toutefois derrière cet objectif, nous devons bien évidemment passer par les fonctions déjà programmées en assembleur par notre professeur ainsi que les fonctions que nous devons codés nous mêmes en MIPS32 dans le logiciel mars. Ce rapport comportera donc quatres principales parties qui serviront à décrire les principales fonctions à coder permettant de faire fonctionner le jeu du Snake dans son intégralité, c'est-à-dire : la direction, la mise à jour de l'environnement du Snake, les conditions de fin de jeu et l'affichage du score à la fin de la partie.

I. Fonction de mise à jour de la direction

1. Algorithme

Dans un premier temps pour la fonction majDirection, nous avons pensé à notre tout premier algorithme qui reste néanmoins peu optimal:

Si la direction du snake actuelle est égale à 2 (gauche) et que l'argument passé n'est pas égale à 0 (droite: sens opposé), on change la direction par l'argument passé.
Si la direction du snake actuelle est égale à 3 (haut) et que l'argument passé n'est pas égale à 1(bas: sens opposé), on change la direction par l'argument passé.

Mais en prenant du recul, nous avons cherché à optimiser l'algorithme, car il y avait quand même beaucoup de conditions pour un cas de direction à gérer. Donc, nous avons pensé à faire avec le calcul du modulo (direction actuelle + argument % 2) et de vérifier que si le reste est zéro, on ne change pas la direction, car l'argument passé sera donc le sens opposé de la direction actuelle du snake. Par exemple, si la direction actuelle est égale à 2 et que l'argument passé est égale à 0, alors le reste sera de 0. Ainsi notre algorithme optimisée ressemble à celle-ci:

Si la direction actuelle du snake modulo l'argument passé de la nouvelle direction est égal à 0, on ne change pas la direction.
Sinon, si c'est différent de 0, on change la direction par l'argument passé.

2. En assembleur MIPS32

Nous avons décidé, avant de coder en assembleur, de le coder en utilisant un langage de haut niveau telle que le C# pour mieux visualiser le code avant le passage en assembleur.

Ainsi, en codant en assembleur, nous avons complété la fonction majDirection de cette façon:

```
################################ majDirection ################################
# Paramètres: $a0 La nouvelle position demandée par l'utilisateur. La valeur
#                 étant le retour de la fonction getInputVal.
# Retour: Aucun
# Effet de bord: La direction du serpent à été mise à jour.
# Post-condition: La valeur du serpent reste intacte si une commande illégale
#                 est demandée, i.e. le serpent ne peut pas faire un demi-tour
#                 (se retourner en un seul tour. Par exemple passer de la
#                 direction droite à gauche directement est impossible (un
#                 serpent n'est pas une chouette)
##############################################################################

majDirection:

# En haut, ... en bas, ... à gauche, ... à droite, ..


lw $t0 snakeDir #on charge snakeDir dans le registre $t0

add $t1 $t0 $a0 # On additionne la direction actuelle avec l'argument passé à cette fonction
li $t2 2
div $t1 $t2 #S
mfhi $t3 # on prend le reste de la division


if: beq $t3, $0, else # Si le reste de la division est égal à zéro (flèche contraire), on saute cette condition.
sw $a0, snakeDir

else:

jr $ra
```

II. Fonction de mise à jour des structures de données

1. Algorithme

Pour la fonction *updateGameStatut* nous avions trois objectif: faire bouger le snake, tester si le serpent a mangé le bonbon (si oui, augmenter sa taille) , ajouter un nouvel obstacle et un nouveau bonbon.

Pour le déplacement du snake, nous avons tout d'abord géré le cas où *tailleSnake* = 1, c'est-à-dire qu'il n'y a que la tête. Cependant on a rencontré un problème les positions x et y. Pour préciser, nous avons rencontré en cours de route un problème: les axes x et y étaient inversés. C'est à dire que l'axe des abscisses est y et que celui des ordonnées est x, ce qui nous a mis en situation de difficulté, car notre snake ne bougeait pas comme il le fallait et que nous devons trouver en plus de cela d'où provient ce problème. Donc, nous avons décidé de changer le x et le y pour faire correspondre à quelque chose de compréhensible. Donc, les nouvelles valeurs correspondant à la direction après modification sont : (2 pour la gauche, 0 pour la droite, 3 pour le haut, 1 pour le bas).

Tout d'abord, voici l'algorithme pour le déplacement de la tête:
Si la direction du snake = 0 on ajoute 1 à snakePosY, si la direction du snake = 2 on décrémente 1 à snakePosY, si la direction du snake = 1 on décrémente 1 à snakePosX, et si la direction du snake = 3 on ajoute 1 à snakePosX.

Ensuite on devait gérer le cas où la taille est supérieure à 1. Nous avions au départ pensé à cette algorithme:
i = tailleSnake-1
tant que i > 0
snakePosX[i] = snakePosX[i-1]
snakePosY[i] = snakePosY[i-1]
i = i - 1
une fois la boucle terminée, on saute vers la partie où on gère le mouvement de la tête.

Cependant on a remarqué que lorsqu'on augmentait la taille du snake, il se mettait à faire des mouvements assez particuliers et que la tête ne réagissait plus correctement. Après plusieurs jours à essayer de résoudre le problème, on a compris que *tailleSnake* n'est pas égal à *snakePosX.Length* (le nombre d'élément qu'on a dans le tableau *snakePosX*), déjà car c'est un tableau et que sa taille ne change pas et surtout car le rôle de tailleSnake est principalement pour savoir combien de pixel de *snakePosX* et *snakePosY* on veut afficher. C'est à dire qu'on peut très bien avoir 10 coordonnés x et y dans *snakePosX* et *snakePosY*, mais si *tailleSnake* = 4, on affichera que 4 pixels.
C'est alors qu'on a pensé à changé notre boucle pour que *i* soit égal à *tailleSnake* et non pas *tailleSnake* - 1. En faisant cela, on aura toujours un pixel à la fin du snake qui ne sera pas affiché et on l'a alors appelé le pixel fantôme. En ajoutant notre pixel fantôme on a réglé notre problème, car si la taille augmente → on affiche le pixel fantôme et on met à jour les coordonnées du nouveau pixel fantôme.

Une fois que le snake a bougé il fallait vérifier que les coordonnées de la tête du snake étaient égales aux coordonnées du bonbon:
Si snakePosX[0] est différent de candyPosX → on va à la fin de la fonction
Si snakePosY[0] est différent de candyPosY → on va à la fin de la fonction
Si à ce moment on est pas à la fin de la fonction c'est lorsque la tête du snake est sur les coordonnées du bonbon, et dans ce cas on va changer les coordonnées du bonbon, on augmente le score et on augmente la taille du snake.

Suite à cela, on va s'occuper de l'ajout d'un nouvel obstacle. On commence par mettre à jour *numObstacle* pour qu'il corresponde au *scoreJeu*, puis on ajoute à la fin de *obstaclesPosX* et de *obstaclesPosY* des nouvelles coordonnées.

2. En assembleur MIPS32

```
############################## updateGameStatus ##############################
# Paramètres: Aucun
# Retour: Aucun
# Effet de bord: L'état du jeu est mis à jour d'un pas de temps. Il faut donc :
#                  - Faire bouger le serpent
#                  - Tester si le serpent à manger le bonbon
#                    - Si oui déplacer le bonbon et ajouter un nouvel obstacle
##############################################################################

updateGameStatus:
# jal hiddenCheatFunctionDoingEverythingTheProjectDemandsWithoutHavingToWorkOnIt

addi $sp $sp -4
sw $ra 0($sp)

Queue:
lw $t0, tailleSnake # on charge la taille Snake dans le registre $t0
la $t1, snakePosX # on charge l'adresse du snakePosX dans $t1
la $t2, snakePosY #on charge l'adresse du snakePosY dans $t2
li $t5 4
subi $t6 $t0 1 # taillesnake - 1

li $t7, 0 #index

loop: beq $t7, $t0, Tete #Tant que l'index dans le registre $t7 n'est pas égal à 0, on continue la boucle

#De la queue jusqu'à le bout du corps du snake avant sa tête
mul $t8 $t6 $t5
addu $t9, $t8, $t1

lw $t4 0($t9) # snakePosX
sw $t4 4($t9) # pixel fantôme

addu $t3, $t8, $t2
lw $t4, 0($t3) # snakePosY
sw $t4 4($t3) # pixel fantôme

addi $t7 $t7 1 # incrémentation de l'index de la boucle
subi $t6 $t6 1
j loop
```

```
Tete:

lw $a0 snakeDir

# NB : x et y sont inverés , (voir explication dans la partie 1 du rapport)

droite:bne $a0 0 gauche # premier cas si $a0 = 0 (droite), on ne saute pas
lw $t1 snakePosY($0)
addi $t1 $t1 1
sw $t1 snakePosY($0) # On sauvegarde la poistionY + 1 dans la tête du snake
j candyUpdate

gauche: bne $a0 2 haut # deuxième cas si $a0 = 2 (gauche), on ne saute pas
lw $t1 snakePosY($0)
subi $t1 $t1 1
sw $t1 snakePosY($0) # On sauvegarde la positionY - 1 dans la tête du snake
j candyUpdate

haut:bne $a0 1 bas # troisième cas si $a0 = 1 (haut), on ne saute pas
lw $t1 snakePosX($0)
addi $t1 $t1 1
sw $t1 snakePosX($0) # On sauvegarde la positionX + 1 dans la tête du snake
j candyUpdate

bas: # dernier cas si $a0 = 3 (bas)
lw $t1 snakePosX($0)
subi $t1 $t1 1
sw $t1 snakePosX($0) # On sauvegarde la positionX - 1 dans la tête du snake
j candyUpdate
```

```
candyUpdate:
addi $sp $sp -4
sw $ra 0($sp)

# On charge toutes les choses nécessaire à la position du candy dans des registres temporaires

lw $t1, snakePosX($0)
lw $t2, snakePosY($0)
lw $t3 candy
lw $t4 candy + 4
lw $t5 tailleSnake
li $t6 4

candyCheck:
bne $t1 $t3 finupdate # On vérifie si la position de la tête du snake est celle du bonbon
bne $t2 $t4 finupdate

jal newRandomObjectPosition # Fonction qui renvoie des positions X et Y aléatoires
sw $v0 candy # On sauvegarde la position X retournée par la fonction random dans Pos X de candy
sw $v1 candy + 4 # On sauvegarde la position Y retournée par la fonction random dans Pos Y de candy

lw $t0 scoreJeu
addi $t0 $t0 1 # On incrémente le score du jeu
sw $t0 scoreJeu # On le sauvegarde dans scoreJeu


addi $t5 $t5 1 # On incrémente la taille du snake et on la sauvegarde dans tailleSnake
sw $t5 tailleSnake

```

```
Obstacles:

jal newRandomObjectPosition # Fonction qui renvoie des positions X et Y aléatoires

lw $t0, scoreJeu
sw $t0, numObstacles # Il y a autant d'obstacles que de points dans le score du jeu, donc on sauvegarde le score dans numObstacles
la $t1, obstaclesPosX
la $t2, obstaclesPosY
li $t6 4

mul $t3 $t0 $t6

addu $t4, $t3, $t1 # permet d'atteindre la position x de l'obstacle voulu

addu $t5, $t3, $t2 # permet d'atteindre la position y de l'obstacle voulu

sw $v0, 0($t4) # On sauvegarde la position X renvoyée par la fonction random dans l'adresse de la position X de l'obstacle voulu
sw $v1, 0($t5) # On sauvegarde la position Y renvoyée par la fonction random dans l'adresse de la position Y de l'obstacle voulu

jal printObstacles # permet d'afficher les obstacles


finupdate:

lw $ra 0($sp)
addi $sp $sp 4

jr $ra

```

III. Fonction de détection et des conditions de fin de jeu.

1. Algorithme

Pour les conditions de fin de jeu on en retrouve 3:
Si le snake touche une bordure
Si le snake se mange lui même
Si le snake touche un obstacle

Si une de ces conditions est réalisée, on saute vers *failure* où on change la valeur de v0 pour la mettre à 1 avant de sauter dans *FinJeu* qui renvoie la valeur de v0 dans *mainloop*. Ce qui va provoquer le game over.

Pour la première condition, on a simplement fait des conditions pour savoir si les coordonnées de la tête du snake étaient en dehors de la grille:
Si snakePosX = 16 → on saute vers failure
Si snakePosY = 16 → on saute vers failure
Si snakePosX = -1 → on saute vers failure
Si snakePosY = -1 → on saute vers failure

Dans le cas où on n'est pas à failure (c'est à dire que le snake n'a pas touché une bordure) on va vérifier si le snake se mange lui-même. Cependant, dans le cas où la taille du snake est 1 on peut directement sauter à *FinJeu* sans passer par *failure* car si la taille est 1, cela veut dire qu'il n'y a que la tête et donc qu'il ne peut pas se manger lui même, et qu'il n'a pas encore mangé de bonbon donc pas encore d'obstacle.

Si la taille est supérieur à 1, on va faire une boucle où on va vérifier si les coordonnées de la tête du snake sont égaux aux coordonnées des autres pixels de la queue du snake:
i = 1
tant que i < tailleSnake
si snakePosX[i] != snakePosX[0] → on passe au pixel suivant en incrémentant i
si snakePosX[i] = snakePosY[0] → on saute dans failure

Si à la fin de la boucle on a toujours pas sauté dans failure, cela veut dire que la tête du snake n'a pas touché un autre pixel de la queue et dans ce cas, on vérifie si la tête touche un obstacle.

Pour cela, on refait une boucle similaire à celle qu'on vient de réaliser:
i = 0
tant que i < numObstacle
si snakePosX[0] != obstaclePosX[i] → on passe au pixel suivant en incrémentant i
si snakePosX[0] = obstaclePosY[i] → on saute dans failure

Si à la fin de la boucle on a toujours pas sauté dans failure, cela veut dire que la tête du snake n'a pas touché d'obstacle, on va donc dans *FinJeu*.

2. En assembleur MIPS32

```
############################## conditionFinJeu ##############################
# Paramètres: Aucun
# Retour: $v0 La valeur 0 si le jeu doit continuer ou toute autre valeur sinon.
#############################################################################

conditionFinJeu:

li $v0 0
lw $t0 snakePosX($0)
lw $t1 snakePosY($0)
lw $t2 tailleSnake

border: # Si la tête du snake touche une bordure, on met fin au jeu.
beq $t0 16 failure
beq $t1 16 failure
beq $t0 -1 failure
beq $t1 -1 failure


beq $t2 1 FinJeu # S'il n'y a que la tête du snake, on a pas besoin de vérfier les deux autres conditions de fin du jeu.

autofeed: # On vérifie si la tête du snake touche une partie de son corps.
li $t4 4
li $t5 1

whileautofeed: beq $t5 $t2 crashObstacle # On sort de cette boucle pour passer à la dernière condition de fin de jeu seulement si la tête du Snake ne touche pas son corps.
mul $t9 $t5 $t4
lw $t6 snakePosX($t9)
lw $t7 snakePosY($t9)
bne $t0 $t6 incrementautofeed # si la position X de la tête du snake n'est pas égale à celle d'un bout de son corps, on continue la boucle
beq $t1 $t7 failure # si la position Y est celle d'un bout de son corps, on met fin au jeu et on sort de la boucle.

incrementautofeed: # Incrémentation du i de la boucle de vérification
addi $t5 $t5 1
j whileautofeed
```

```
crashObstacle:
lw $t2, numObstacles
li $t5, 0

whilecrashObstacle: beq $t2 $t5 FinJeu # Si la tête du snake ne touche aucun obstacle, on continue le jeu

mul $t9 $t5 $t4 # On accède à chaque obstacle
lw $t6 obstaclesPosX($t9)
lw $t7 obstaclesPosY($t9)

bne $t0 $t6 incrementcrashObstacle # On vérifie si la position X de la tête du snake est celle de l'obstacle pointée
beq $t1 $t7 failure # On vérifie si la position Y de la tête du snake est celle de l'obstacle pointée

incrementcrashObstacle: # Incrémentation pour faire continuer la boucle while
addi $t5 $t5 1
j whilecrashObstacle


failure: # On met fin au jeu (défaite)
li $v0 1

FinJeu:
jr $ra
```

IV. Fonction affichage de la fin de la partie avec le score

1. Algorithme

Pour l'affichage du score nous avions pensé à afficher un message particulié en fonction du score de la personne.

Si le score = 0 on affiche veryBadGame → on saute à la fin de l'affichage
Si le score < 5 on affiche badGame → on saute à l'affichage du score
Si le score < 10 on affiche okGame → on saute à l'affichage du score
Sinon on affiche goodGame → on saute à l'affichage du score

2. En assembleur MIPS32

On a donc écrit les différents messages dans la partie .data avec avec l'écriture .asciiz, exemple:

veryBadGame: .asciiz "??????????? tu trolls gros"

```
############################## affichageFinJeu ##############################
# Paramètres: Aucun
# Retour: Aucun
# Effet de bord: Affiche le score du joueur dans le terminal suivi d'un petit
#                mot gentil (Exemple : «Quelle pitoyable prestation!»).
# Bonus: Afficher le score en surimpression du jeu.
#############################################################################

affichageFinJeu:

li $v0, 4
lw $a0, scoreJeu
blt $a0, 1, veryBadMessage # Si le score est inférieur à 1
blt $a0, 5, badMessage # si le score est égale ou supérieur à 1 et est inférieur à 5
blt $a0, 10, okMessage # si le score est inférieur à 10 mais au dessus ou égal à 5

goodMessage: # Sinon si le score est supérieur ou égal à 10
la $a0, goodGame
syscall
j printScore

veryBadMessage:
la $a0, veryBadGame
syscall
j fin

badMessage:
la $a0, badGame
syscall
j printScore

okMessage:
la $a0, okGame
syscall
j printScore

GoodMessage:
la $a0, goodGame
syscall

printScore:
li $v0, 1
lw $a0, scoreJeu
syscall
li $v0, 10
syscall

fin:
jr $ra
```

V. Quelques ajouts

1. Fréquence du nombre d'obstacle

On a tout d'abord pensé à changer la fréquence d'apparition des obstacles en mettant un obstacle à toutes les deux pommes mangées. Pour ce faire, il a juste à ajouter *div $t0, $t0, 2* dans l'étiquette 'Obstacles' entre *lw $t0, scoreJeu* et *sw $t0, numObstacles* :

lw $t0, scoreJeu
div $t0, $t0, 2
sw $t0, numObstacles

En faisant cela, *numObstacle* augmentera à chaque fois que le joueur aura mangé deux pommes. Cependant, étant donné que la difficulté était déjà assez simple nous avons décidé de mettre de côté cette idée et d'en trouver une nouvelle pour augmenter la difficulté du jeu.

2. Augmentation de la vitesse du snake

Afin de rendre le jeu plus difficile nous avons eu l'idée d'augmenter la vitesse du snake à chaque fois qu'il mange un bonbon. Etant donnée que le nombre de bonbon

mangé est les score, on a mis en place une relation pour baisser la vitesse du snake qui est de base à 500:
vitesse du snake = 500
*nouvelle vitesse du snake = vitesse du snake - (score * 20)*
En faisant comme cela, la vitesse du snake va augmenter progressivement de manière linéaire.

3. Rainbow Snake

Pour faire le rainbow snake, on a tout d'abord ajouter un nouveau tableau rainbow avec des couleurs qu'on a mises dedans (le même fonctionnement que ce qui avait déjà de base avec le tableau *colors*).
Une fois le tableau créé, on va dans l'étiquette *printSnake* et plus précisément dans la boucle *PSLoop*, où on va générer une valeur aléatoire entre 0 et 9 (car on a 9 couleurs dans notre tableau rainbow) avec syscall en mettant 42 dans v0 et 9 dans a1. Quand on fait syscall avec 42, on génère un nombre aléatoire entre 0 et le nombre qu'on a mis en argument, ici 9.
Une fois qu'on a notre nombre aléatoire, on le multiplie par 4 pour avoir l'adresse de la couleur aléatoire dans le tableau rainbow, et on finit par afficher cette couleur pour le pixel qu'on traite, puis on passe au pixel suivant jusqu'à sortir de la boucle.

Conclusion:

Pour conclure, ce projet snake en assembleur nous a permis de mettre en pratique les connaissances acquises lors des cours de s12 et d'améliorer nos compétences en algorithmique notamment lorsqu'on cherchait à optimiser nos fonctions. Le travail a été réparti équitablement et chacun a pu toucher à tout, que ce soit l'algorithmique ou la programmation en assembleur.
De plus, on a apprécié travailler sur ce projet et de voir que malgré les difficultés on arrivait toujours à trouver une solution et à avancer dans le projet.