


STS 1 SN <b>IR</b> 2021-2022	PROGRAMMATION STRUCTUREE  <b>DS – 4H00</b>	
---------------------------------	--	---

## TRAVAIL DEMANDE - RECOMMANDATIONS

**UN LECTURE COMPLETE ATTENTIVE DE L'INTEGRALITE DU SUJET EST RECOMMANDEE AVANT D'ENTAMER TOUT DEVELOPPEMENT OU CODAGE.**

Une base applicative est fournie sous l'archive « BaseApplicative.zip » contenant un projet configuré Eclipse/SDL/GCC.

Après importation sous Eclipse de cette base, renommer aussitôt le projet, comme à l'accoutumée, avec vos <NomPrenom>.

Dans les entêtes des fichiers .h et .c, indiquer vos nom et prénom en lieu et place prévu à cet usage.

Le travail demandé consiste à l'élaboration des codes répondant aux fonctionnalités attendues aux emplacements marqués « ToDo ». **VOUS N'EFFACEREZ PAS CES MARQUES MAIS PLACEREZ LE CODE DEMANDE EN DESSOUS DE CHACUNE D'ELLES.**

Il est également recommandé de procéder par étapes constructives dans l'élaboration de l'application, on la construisant pas à pas, et en testant chaque étape avant d'empiler la suivante.

## A REMETTRE - RECOMMANDATIONS

Le projet complété et paramétré avec vos nom et prénom, sera archivé au format .zip par l'utilitaire d'exportation intégré à Eclipse. L'archive devra elle-même être nommée de vos nom et prénom, qui on le rappelle, SANS ACCENTS SANS ESPACES PREMIERE LETTRE DES MOTS EN MAJUSCULE.

Cette archive sera à déposer sous Moodle.

# CONCEPTION LOGICIELLE : « The Snake Game »

## 1 DESCRIPTION – CONCEPT

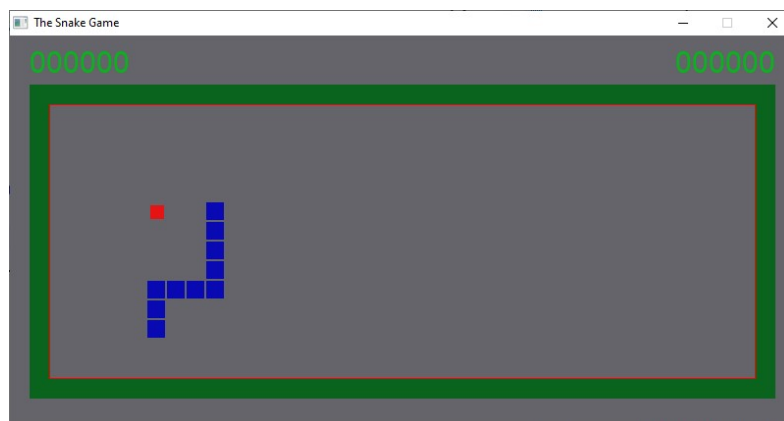
On reprend dans ce développement le jeu vidéo du « Serpent » dans lequel le joueur dirige un serpent qui grandit et constitue ainsi lui-même un obstacle.

Le joueur contrôle une longue ligne semblable à un serpent, qui doit slalomer entre les bords de l'écran et les obstacles qui parsèment le niveau. Pour gagner chacun des niveaux, le joueur doit faire manger à son serpent un certain nombre de pastilles similaires à de la nourriture, allongeant à chaque fois la taille du serpent. Alors que le serpent avance inexorablement, le joueur ne peut que lui indiquer une direction à suivre (en haut, en bas, à gauche, à droite) afin d'éviter que la tête du serpent ne touche les murs ou son propre corps, auquel cas il risque de mourir.

Certains clones proposent des niveaux de difficulté dans lesquels varient l'aspect du niveau (simple ou labyrinthique), le nombre de pastilles à manger, l'allongement du serpent ou encore sa vitesse.

Pour la suite du développement, on modifiera quelques règles :

- Lorsque le serpent atteint l'un des murs de la zone de jeu, celui aura la faculté de réapparaître dans la zone de jeu du côté opposé.
- Il n'y aura pas d'obstacles placés dans la zone de jeu.
- Les pastilles de nourriture apparaîtront une à une de manière aléatoire en position dans la zone de jeu et à intervalle de temps de longueur également aléatoire. Au maximum, une seule pastille de nourriture sera présente dans la zone de jeu. Tant qu'une pastille de nourriture est présente dans la zone de jeu, le processus de génération aléatoire est suspendu. Celui reprendra dès que le serpent aura absorbé la pastille de nourriture en passant sa tête sur celle-ci.



Un aperçu du rendu du « Snake Game ».

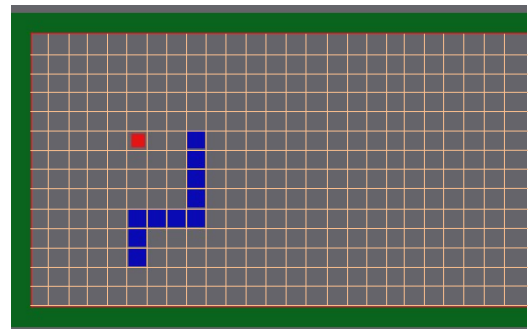
## 2 MODELISATION LOGICIELLE

### 2.1 Entité « Snake » :

#### 2.1.1 Structuration :

Une entité « snake » sera définie par les éléments structurels donnés ci-dessous.

Le serpent sera constitué de « **sections** » qui seront par la suite représentés à l'écran par des **carrés**. Les sections sont repérées par leur **coordonnées tabulaires** : la zone de jeu est considérée comme un tableau de cases à deux dimensions (voir ci-contre, la case de coordonnées 0 ; 0 se trouvant dans le coin haut à gauche). Chaque section a un sens de déplacement parmi les quatre possibles : **DIR\_RIGHT**, **DIR\_LEFT**, **DIR\_UP**, **DIR\_DOWN**, (**DIR\_NONE** existe pour les besoins du codage).



```
typedef enum e_direction{
    DIR_RIGHT = 0 ,
    DIR_LEFT,
    DIR_DOWN,
    DIR_UP,
    DIR_NONE
}t_direction;
```

Enfin chaque section possède un lien sur la section qui la suit et sur celle qui la précède.

```
typedef struct s_section{
    int m_iPosX; /* section x coordinate */
    int m_iPosY; /* section y coordinate */
    t_direction m_direction; /* section moving direction */
    struct s_section* m_pNext; /* pointer to next section */
    struct s_section* m_pPrev; /* pointer to previous section */
}t_section;
```

L'ensemble des sections, organisées en liste doublement chaînée, forme une entité serpent, elle même représentée par la structure suivante :

```
struct s_snake {
    t_section * m_pHead; /* pointer to the snake head section */
    t_section * m_pTail; /* pointer to the snake tail section */
    SDL_Color m_color; /* RGBA snake color */
    size_t m_szLength; /* snake length in number of sections */
};
```

#### 2.1.2 Actions & Comportements :

Une entité « snake » implémentera son comportement au travers des fonctions suivantes :

```
t_snake*SnakeNew(size_t szNbSections, int iHeadPosX, int iHeadPosY,
    t_direction direction, const SDL_Color*pColor);
```

Crée une nouvelle entité « snake » avec les aspects suivants :

```

size_t szNbSections : nombre initial de sections, tête du serpent comprise
int iHeadPosX : position x initiale de la tête du serpent
int iHeadPosY : position y initiale de la tête du serpent
t_direction direction : direction initiale du serpent : toutes les sections
                        adopteront la même direction à la création du serpent.
const SDL_Color* pColor : couleur RGBA du serpent (pointeur sur structure)
  
```

SnakeNew() devra créer une première section à l'aide la fonction « SectionNew() » qui représentera la tête du serpent, puis elle appellera la fonction « SnakeGrowUp() » autant de fois nécessaires pour compléter le nombre demandé de sections.

SnakeNew() retourne le pointeur sur la nouvelle entité créée.

```
t_snake*SnakeDel(t_snake*pSnake);
```

Restitue les ressources mémoire initialement requises par l'entité snake.

SnakeDel() retourne le pointeur NULL.

```
t_snake*SnakeDraw(const t_snake*pSnake, SDL_Renderer*pRenderer, const
SDL_Rect*pArea);
```

Réalise le tracé graphique de l'entité snake.

Chaque section du serpent devra être passée en revue pour être dessinée par un carré de dimensions SNAKE\_SECTION\_SIZE aux coordonnées graphiques déduites des coordonnées tabulaires m\_iPosX et m\_iPosY de la section concernée. Les champs x et y du paramètre pArea complètent le calcul des coordonnées graphiques. A noter que les sections sont espacées entre elles de la quantité SNAKE\_SECTION\_SPACING.

SnakeDraw() retourne le pointeur sur l'entité snake concernée par la fonction.

```
t_snake*SnakeMove(t_snake*pSnake, const SDL_Rect*pArea);
```

Réalise le déplacement de l'entité snake en suivant le principe décrit ci-après.

- Dans un premier temps chaque section est passée en revue et selon leur sens de déplacement, l'une de leurs coordonnées tabulaires m\_iPox **ou** m\_iPosY est incrémentée **ou** décrémentée.

A l'issue de cet incrément/décrément, la valeur est rectifiée en conséquence si l'une des limites de la zone de jeu est franchie. On rappelle que le serpent a la faculté de réapparaître sur le bord opposé de celui qui a été franchi.

*Pour information, le nombre de « cases » en ligne ou en colonne de la représentation tabulaire, est obtenu en divisant la largeur ou la hauteur de la zone de jeu par la somme :*

*(SNAKE\_SECTION\_SIZE+SNAKE\_SECTION\_SPACING).*

- Dans un second temps, pour simuler le mouvement d'un « véritable » serpent, chaque section devra adopter le sens de déplacement de la section qui la précède, bien entendu si toutefois cette section existe ! Ce traitement devra commencer par la queue du serpent en remontant jusqu'à sa tête.

SnakeMove() retourne le pointeur sur l'entité snake concernée par la fonction.

```
t_snake*SnakeGrowup(t_snake*pSnake);
```

Fait grandir le serpent en lui ajoutant une section à sa queue. La nouvelle section adopte le même sens de déplacement que celui de l'actuelle queue au moment de l'insertion.

SnakeGrowup() retourne le pointeur sur l'entité snake concernée par la fonction.

```
t_snake*SnakeChangeDirection(t_snake*pSnake, t_direction direction);
```

Fait changer la direction de la tête du serpent en lui appliquant la nouvelle direction passée en paramètre avec toutefois la restriction suivante :

- la direction de la tête du serpent ne sera pas changée si la nouvelle direction demandée est son opposée.

SnakeChangeDirection() retourne le pointeur sur l'entité snake concernée par la fonction.

```
int SnakeIsHeadAt(const t_snake*pSnake, int iAtX, int iAtY);
```

Renseigne sur le fait que la tête du serpent soit ou non sur la case de coordonnées tabulaires iAtX et iAtY. Retourne une valeur non nulle en cas de véracité.

```
int SnakeHasBittenHimself(const t_snake*pSnake);
```

Renseigne sur le fait que la tête du serpent soit ou non sur l'une de ses propres sections : en gros « le serpent se mord-il lui-même ? ». Retourne une valeur non nulle en cas de véracité.

Fonctions « internes » ou « privées » :

```
t_section*SectionNew(t_section*pPrev, t_section*pNext, int iPosX,  
                    int iPosY, t_direction direction);
```

Permet de créer une nouvelle section tout en l'initialisant avec les caractéristiques passées en paramètre et en la chaînant à la liste existante.

SectionNew() retourne le pointeur sur l'entité nouvellement créée.

```
t_section*SectionDelReturnNext(t_section*pSection);
```

Permet de détruire la section passée en paramètre tout en procédant au re-chaînage de la liste.

SectionDelReturnNext() retourne le pointeur sur l'entité qui suivait celle qui a été détruite.

`t_snake*SnakePushback(t_snake*pSnake);`

Permet de rajouter une nouvelle section à la queue du serpent passé en paramètre suivant le principe décrit ci-après :

- la direction de la nouvelle section sera celle de l'actuelle queue au moment de l'insertion.
- les coordonnées tabulaires de la nouvelle section seront déduites de celles de l'actuelle queue au moment de l'insertion en prenant en compte la direction de celle-ci.

\_SnakePushback() retourne le pointeur sur l'entité snake concernée par la fonction.

## 2.2 Entité « Scene » :

Cette entité contiendra la scène de jeu et en conséquence, toutes les entités graphiques y évoluant. Cette entité recevra d'autre part les actions du joueur en provenance de l'application et les répercutera sur les entités évoluant dans la scène.

### 2.2.1 Structuration :

Une entité « scene » sera définie par les éléments structurels donnés ci-dessous.

```

struct s_scene{
    SDL_Rect      m_frameArea;      /* for representing the scene frame delimiting area */
    SDL_Rect      m_gameArea;      /* for representing the effective game area */
    SDL_Renderer* m_pRenderer;      /* window scene graphical renderer */
    SDL_Color     m_colorBkgnd;     /* RGBA background color */
    SDL_Color     m_colorFrame;     /* RGBA scene frame delimiter color */
    TTF_Font      *m_pFont;        /* current font for textual items */
    /*-----*/
    t_snake       *m_pSnake;        /* the snake entity pointer */
    SDL_Point     m_ptFood;         /* the tabular coordinates of snake food */
    Uint32        m_foodTimer;      /* the timer food generator */
    Uint32        m_score;          /* the player game score */
    Uint32        m_elapsedTime;    /* the game elapsed time in seconds */
};
  
```

Le serpent sera constitué de « **sections** » qui seront par la suite représentés à l'écran par des carrés. Les sections sont repérées par leur coordonnées tabulaires : la zone de jeu est considérée comme un tableau à deux dimensions. Chaque section

### 2.2.2 Actions & Comportements :

Une entité «scene» implémentera son comportement au travers des fonctions suivantes :

`t_scene*SceneNew(int iWidth, int iHeight, SDL_Renderer*pRenderer,  
TTF_Font*pFont);`

Crée une scène de jeu avec les paramètres donnés. Initialise les champs de la structure. Crée une entité « snake ».

Retourne le pointeur sur la nouvelle entité « scene » créée.

```
t_scene*SceneDel(t_scene*pScene);
```

Détruit l'entité « scene » passé en paramètre en restituant les ressources mémoire requises lors de la création.

Retourne le pointeur NULL.

```
t_scene*SceneDraw(t_scene*pScene);
```

Procède aux actions liées à la représentation graphiques de toutes les entités concernées par celle-ci.

Retourne le pointeur l'entité « scene » concernée par la fonction.

```
t_scene*SceneAnimate(t_scene*pScene);
```

Procède aux actions liées à l'animation de toutes les entités concernées par celle-ci.

Retourne le pointeur l'entité « scene » concernée par la fonction.

```
t_scene*SceneKeydown(t_scene*pScene, SDL_KeyboardEvent*pEvent);
```

```
t_scene*SceneKeyup(t_scene*pScene, SDL_KeyboardEvent*pEvent);
```

Procèdent aux actions liées aux événements clavier en les répercutant sur les entités en charge par « scene ».

Retournent le pointeur l'entité « scene » concernée par ces fonctions.

## 2.3 Entité « App » :

### 2.3.1 Structure :

L'entité « app » sera définie par la structure donnée ci-dessous.

```
struct s_app{
    t_status      m_uStatus;          /* status flags of application      */
    SDL_Window *  m_pWindow;          /* the main window structure pointer */
    SDL_Renderer* m_pRenderer;        /* the main window graphical renderer */
    SDL_TimerID   m_timerID;          /* the main timer ID                */
    TTF_Font *    m_pFont;           /* the main font                     */
    /*-----*/
    t_scene*      m_pScene;           /* the scene entity pointer          */
};
```

### 2.3.2 Fonctionnalités :

**t\_app\*AppNew(void);**

Réserve dynamiquement l'espace mémoire nécessaire à la structure **t\_app**, initialise les champs de celle-ci et crée les diverses entités nécessaires à l'application.

**t\_app\*AppDel(t\_app\*pApp);**

Restitue les ressources allouées dynamiquement lors de la phase de « construction » de l'application.

**int AppRun(t\_app\*pApp);**

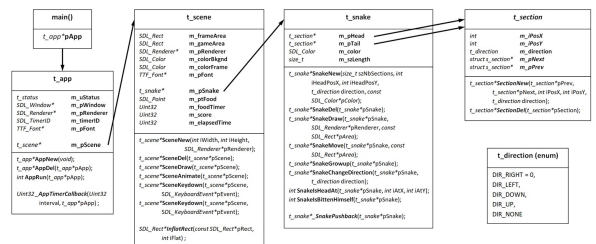
Implémente la partie active de l'application en procédant au traitement des événements s'y rattachant.

**Uint32 \_AppTimerCallback(Uint32 interval, t\_app\*pApp);**

Implémente la partie animation de l'application. Prend en charge le dessin et le mouvement des entités graphiques.

### 2.4 Schématisation du modèle retenu :

*SCHEMA DETAILLE EN ANNEXE 1 EN FIN DE DOCUMENT*



## 3 DEVELOPPEMENT

### 3.1 ÉTAPE 1 : AFFICHAGE D'UNE UNIQUE SECTION DANS LA SCÈNE

#### Fonctionnalités de base

- Compléter dans un premier temps les fonctions « internes » suivantes du module « snake » :
  - SectionNew()
  - SectionDelReturnNext()
- Compléter ensuite les fonctions suivantes du module « snake » :
  - SnakeNew()

*Pour les premiers tests mettant en œuvre un serpent constitué d'une seule section, la fonction SnakeNew() déclenchera la création d'une section par appel de la fonction SectionNew() et affectera la valeur de retour*



aux champs `m_pHead` et `m_pTail` de l'entité `Snake` ainsi créée. Le paramètre `szNbSections` ne sera donc pas pris en compte à ce stade-ci.

- `SnakeDel()`
- `SnakeDraw()`

➤ Pour tester cette étape, dans la fonction `SceneNew()`, créer une entité « Snake » en complétant la section :

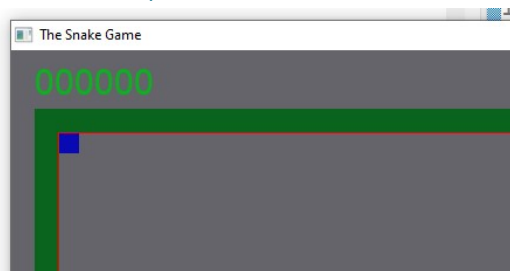
```
/**
 * @todo create a new snake entity
 */

//.m_pSnake      = .... ,
```

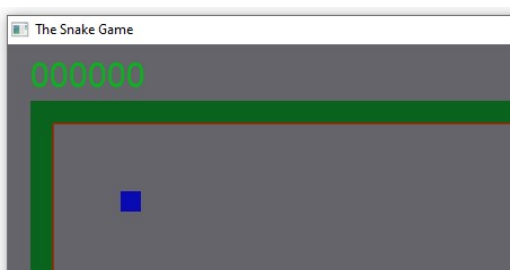
Compléter la fonction `SceneDel()` pour détruire l'entité « Snake ».

Compléter la fonction `SceneDraw()` pour invoquer la fonction de dessin de l'entité « Snake ».

Ci-dessous une capture du rendu obtenu, avec les coordonnées tabulaires (0 ; 0)



Ci-dessous une capture du rendu obtenu, avec les coordonnées tabulaires (3 ; 3)



### 3.2 ÉTAPE 2 : DÉPLACEMENT AUTOMATIQUE D'UNE UNIQUE SECTION DANS LA SCÈNE

Cette étape devra permettre le déplacement automatique de l'unique section dans la scène, avec la prise en compte des limites de celle-ci et l'implémentation du comportement en conséquence selon la description donnée en pages 4 et 5.

➤ Implémenter la première structure `while(pParse){ }` de la fonction `SnakeMove()` en complétant chaque case du `switch(pParse->m_direction)` de celle-ci.

- **Pour tester cette étape, dans la fonction SceneAnimate(), compléter la section suivante :**

```
/* Ask for moving the snake entity ***** */
/**
 * @todo call the snake entity moving function
 */
```

*A ce stade, si toutes les fonctions sont correctement implémentées, l'unique section doit se mouvoir dans la scène, réapparaissant au côté opposé lors des débordements des limites de celle-ci.*

*Pour tester les différentes directions, il suffit de changer celle paramétrée lors de la création de l'entité « snake » dans la fonction SceneNew().*

### 3.3 ÉTAPE 3 : CONTROLER A PARTIR DU CLAVIER LA DIRECTION DE DEPLACEMENT D'UNE UNIQUE SECTION DANS LA SCÈNE

Cette étape devra permettre de changer la direction du déplacement de la section présente dans la scène.

- **Implémenter la fonction SnakeChangeDirection() en complétant chaque case de la structure switch(direction) de celle-ci.**

*A ce stade, les sollicitations des touches de direction du clavier doivent influencer la direction du mouvement de la section présente dans la scène.*

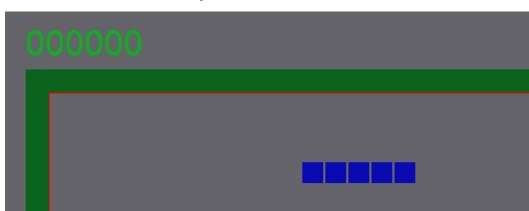
### 3.4 ÉTAPE 4 : UN SERPENT CONSTITUE DE PLUSIEURS SECTIONS

Cette étape devra permettre de construire un serpent constitué de plusieurs sections, et d'en contrôler le déplacement.

- **Implémenter la fonction \_SnakePushback() en complétant chaque case de la structure switch(pSnake->m\_pTail->m\_direction) selon la description faite en page 6.**
- **Compléter la suite de la fonction SnakeNew() pour prendre en compte le paramètre szNbSections afin de compléter la tête de l'entité « snake » avec le reste des sections pour totaliser la longueur attendue.**

*A ce stade, si toutes les fonctions sont correctement implémentées, une entité « snake » complète, selon le nombre initial de sections demandé à sa création, doit évoluer dans la scène.*

*Ci-dessous une capture du rendu obtenu, avec une entité « snake » composée de 5 sections :*



Pour parfaire le contrôle de la direction de déplacement de la totalité du serpent, procéder à la demande suivante :

- **Implémenter la seconde structure while(pParse){ } de la fonction SnakeMove() en tenant compte de la description faite en pages 4 et 5.**

*A ce stade, la totalité de l'entité « snake » doit évoluer correctement dans la scène.*

### 3.5 ÉTAPE 5 : UN SERPENT QUI SE NOURRIT ET QUI GRANDIT

Cette étape devra permettre à l'entité « snake » d'absorber les pastilles de nourriture présentes sur sa trajectoire et ainsi de se voir grandir à chaque absorption.

Principe de la pastille de nourriture :

- La pastille de nourriture est repérée dans la scène à l'aide de ses coordonnées tabulaires maintenues dans le champ m\_ptFood de l'entité « scene ».
  - Si la composante x de ce champ vaut -1, cela indique que la pastille n'est pas/ou n'est plus présente dans la scène : elle n'est donc pas affichée.
  - Pour les autres valeurs cohérentes des coordonnées, la pastille est affichée dans la scène.
  - La génération des pastilles de nourriture obéit à la description faite en page 2.
  - La pastille de nourriture est absorbée par l'entité « snake » lorsque sa tête l'atteint. Cette absorption est simulée par la mise à -1 de la composante x du champ m\_ptFood. Sur absorption d'une pastille, l'entité « snake » grandit d'une section, et le score est augmenté d'une quantité à convenir.
- **Implémenter la fonction SnakeGrowup() qui permet de faire grandir une entité « snake » d'une section.**
  - **Implémenter la fonction SnakelsHeadAt() qui permet de savoir si la tête d'une entité « snake » se trouve en position iAtX ; iAtY.**
  - **Pour tester cette étape, dans la fonction SceneNew(), compléter la section suivante :**

```
/**
 * @todo random a food timer value
 */

//.m_foodTimer = .... ,
```

**Dans la fonction SceneAnimate(), compléter les sections suivantes :**

```
/* Taking care of feeding the snake entity *****/
/**
 * @todo checking if snake entity is on food location, and taking actions if it is.
 */

/* Taking care of food generation process *****/
/**
 * @todo checking if there is no food in scene and taking actions it is.
 */
```

*A ce stade, si toutes les fonctions sont correctement implémentées, une entité « snake » est capable d'absorber les pastilles de nourriture apparaissant de manière aléatoire en position et en rythme. A chaque absorption, l'entité « snake » grandit d'une section et le score évolue.*

### 3.6 ÉTAPE 6 : UN SERPENT QUI NE DOIT PAS SE MORDRE LUI-MEME !

Cette étape devra permettre la détection d'un rebouclage de l'entité « snake » sur elle-même, autrement dit « le serpent se mord lui-même ! ». Dans ce cas le jeu est stoppé par un blocage de la simulation en mode PAUSE.

- **Implémenter la fonction SnakeHasBittenHimself() qui permet de savoir si une entité « snake » s'est mordue « elle-même » !**
- **Pour tester cette étape, dans la fonction SceneAnimate(), compléter la section suivante :**

```

/* Taking care of snake biting himself *****/
/**
 * @todo checking if there the snake is biting himself and return NULL if it is.
 */

```

*A ce stade, si l'entité « snake » se mord, la simulation est stoppée.*

## ANNEXE 1 : SCHEMA DU MODELE RETENU

