Travaux dirigés d'algorithmique n°6

Cours d'Informatique

— IMAC Première Année —

Mini Projet : **Zelda**

- partie 1 -

Cette série de TP vise à la création d'un jeu de type Zelda
(version Super Nintendo, **The Legend of Zelda : A Link to the Past**).

Le développement d'un jeu vidéo est une tache difficile mais passionnante qui permet d'apprendre énormément de chose en programmation, en algorithmique et en génie logiciel.

Ce qui est particulièrement difficile, c'est de savoir par où commencer. Il y a tellement d'élément constituant un jeu que la masse de travail semble insurmontable.

Il faut donc procéder par itérations et effectuer des tests régulièrement.

Avant de commencer à coder, il faut comprendre le fonctionnement global d'un jeu (d'un point de vue programmation). Un jeu est une application interactive qui boucle tant que le joueur n'a pas demandé explicitement la fin du jeu. Cette boucle est appelée boucle d'affichage (ou boucle de jeu). Dans certains langage de programmation, elle est directement prise en charge et le programmeur n'a pas à s'en occuper (Action Script 3 au sein de Flash, Javascript dans les navigateurs, ...).

Cette boucle à la structure suivante :

```
Tant que le joueur n'a pas quitté
{
  Tant qu'il y a des évènements utilisateur à traiter (clavier, souris, ...)
  {
    Traiter un évènement
  }
  Mise à jour du jeu

  Affichage

  Attente pour limiter le nombre d'images par seconde
}
```

Les parties que vous aurez à coder seront "traiter un évènement" et "Mise à jour du jeu" (bien que la bibliothèque graphique fournie vous permettra d'aller plus loin).

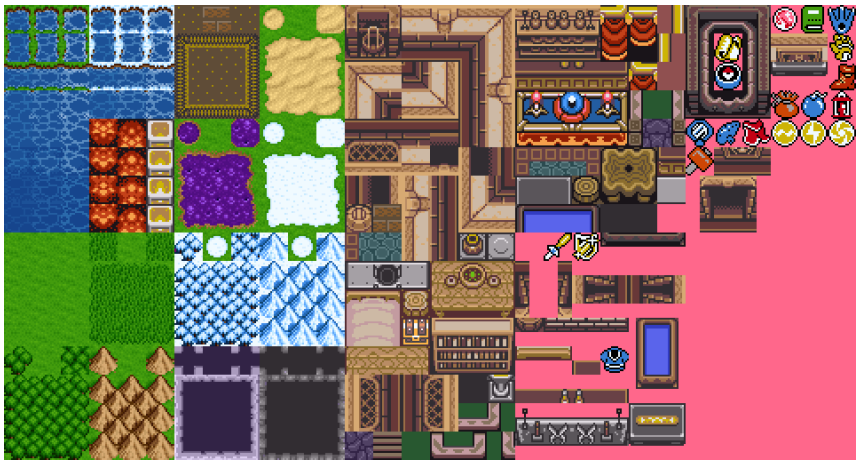
Afin de gérer l'affichage, une bibliothèque graphique vous est proposée (sTileZ). Elle permet le chargement de cartes de tiles (tuiles) et de sprites sur plusieurs couches. Cette bibliothèque ne gère que ce type de dessin. La version proposée se branche sur une autre bibliothèque graphique très célèbre et multi-plateforme : la SDL. Vous aurez un minimum de code à écrire en SDL (ouverture de la fenêtre et manipulation de la structure d'évènement). Tout cela sera expliqué dans les exercices à venir.

Il sera très important de bien séparer le moteur de jeu du moteur graphique. Néanmoins, puisque c'est le premier jeu pour beaucoup d'entre vous, nous commencerons par coder en mélangeant un peu les deux concepts. Cela nous permettra de constater les limites et les difficultés qui apparaissent lorsqu'il faut ajouter de nouvelles fonctionnalités. Nous ferons alors les modifications nécessaires pour ajouter de la modularité au code.

▸ **Exercice 1.** *Le monde*

Comme le jeu que nous voulons faire est en 2D isométrique, nous utiliserons un moteur de tuiles et nous représenterons donc une carte par un tableau à deux dimensions. Le type de ce tableau est quelque peu particulier, en effet, nous avons besoin de savoir si la case du tableau permet le déplacement, l'interaction, le blocage, . . .
Pour cela nous utiliserons une structure particulière que l'on appelle ***enum****.*

*Énumérez toutes les possibilités que peut prendre une case. Puis déclarez cet énumération (****TypeCase****) ainsi qu'une structure (****MapCase****) représentant une case du tableau et comprenant tout ce qui la représente (ici un* ***TypeCase****). Et enfin une structure (****Map****) contenant la hauteur et la largeur d'une case en pixel ainsi que le tableau à deux dimensions positionnées en statiques pour commencer.*



exemple de tileset

▸ **Exercice 2.** *Les personnages*

Il reste à déterminer ce que seront nos personnages jouables ou non, alliés ou non.

*Puisqu'ils sont tous représentés de la même façon (dû moins en grande partie), nous allons faire une structure (****Perso****) englobant toutes les caractéristiques communes.*

Qu'avons nous besoin ?
*Premièrement, un personnage ne prend pas forcément la taille d'un tile (case du tableau), il peut être bien plus gros ou plus petit. De plus, il ne se déplace pas de case en case, mais de point à point (pixel). Pour nous simplifier l'écriture, déclarez une structure (****Coord2D****) qui représente une coordonnée (x,y) dans un tableau (de* ***MapCase*** *ou de pixel). Et écrivez une fonction* ***Coord2D * MapPosition(Map * map, Coord2D * position)*** *qui prend une carte et une coordonnée en pixel et qui retourne cette position dans la carte.*

*Comme dans la plus part des jeux, un objet, personnage, ... interagi de façons différentes avec l'environnement. Ces différentes interactions sont chacune liée à une partie de son corps (hit box, collide box, ...). Déclarez une structure (**Box2D**) définie par deux coordonnées (**Coord2D**) : le coin supérieur gauche et l'inférieur droit.*

Nous parlions de parties du corps correspondant à une interaction, voici ce que nous rechercherons sur un personnage :

lnik.
C'est à partir de cette image qu'on détermine les différentes zones.

La place réelle de link.
Sa position est désignée par le coin supérieur gauche.

Hit box.
La zone qui subit des dégâts.

Collide box.
La zone en contact avec le décor.

*Définissez une structure (**Perso**) qui comprend une position (**Coord2D**), une hit box (**Box2D**) et une collide box (**Box2D**). Toutes ces zones seront définies avec des coordonnées relatives à la position.*

▸ **Exercice 3.** *Détection de collision*

Maintenant que nos structures sont faites,
nous allons gérer les collisions d'une hit box avec le décor d'une carte.
Écrivez une fonction ***int MapCollideAbs(Map * map, Box2D * box) ;***
qui prend en paramètre une carte (map) et une zone (box) et qui retourne 1 s'il y a collision 0 sinon. Les coordonnées de la zone sont considérées "absolues".
Écrivez une fonction ***int MapCollide(Map * map, Box2D * box, Coord2D * ref) ;***
qui prend en plus une coordonnées de référence (les coordonnées de la zone sont relatives à cette référence).

▸ **Exercice 4.** *Affichage graphique*

Vous allez enfin afficher quelque chose à l'écran ! Pour cela vous passerez par la bibliothèque graphique sTileZ qui a été écrite pour ce TP et qui permet l'affichage de cartes de tiles et de sprites évoluant dans ces cartes. Cette bibliothèque ne gère que l'affichage. En elle même elle ne permet pas l'ouverture de fenêtre ou la gestion des évènements. Cela permet de la brancher sur différentes bibliothèque graphique plus bas niveau. Celle qui a été choisie se nomme la SDL (www.libsdl.org). Une extension de sTileZ nommée sTileZSDL fournit les fonctions nécessaire au branchement ainsi que quelques utilitaires. Pour compiler votre projet vous passerez par le Makefile fourni qui s'occupera de définir les bonnes options de compilation.

1 Ouverture d'une fenêtre

La première étape pour afficher quelque chose consiste à ouvrir une fenêtre ! Comme précisé dans l'introduction de l'exercice, sTileZ ne s'occupe pas de l'ouverture de la fenêtre. Il va donc falloir passer par des fonctions de la SDL. Tout d'abord il faut inclure ***<SDL/SDL.h>*** *pour pouvoir utiliser les fonctions de la SDL. La fonction* ***SDL_Init(flags)*** *permet d'initialiser la SDL. Vous l'utiliserez de la manière suivante avant d'appeler toute autre fonction de la SDL :* ***SDL_Init(SDL_INIT_VIDEO) ;***

De la même manière en fin de programme vous appellerez la fonction : ***SDL_Quit() ;***

Cette fonction s'occupe de libérer toutes les ressources allouées par la SDL.

Enfin pour ouvrir une fenêtre il suffit d'utiliser la fonction SDL_SetVideoMode :
SDL_Surface* screen = SDL_SetVideoMode(largeur, hauteur, bbp, flags) ;

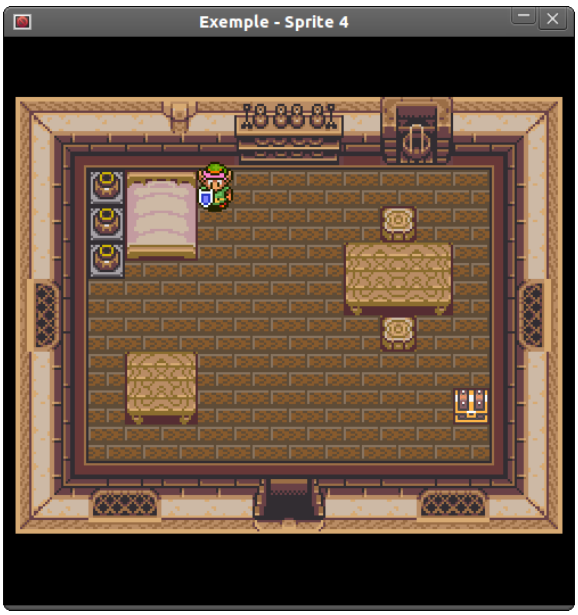
largeur *et* ***hauteur*** *sont les dimensions de la fenêtre.* ***bbp*** *est le nombre de bits par pixel. Vous utiliserez 32 pour cette valeur (32 bits par pixel = un int pour représenter un pixel). Enfin* ***flags*** *correspond à une série d'options. Vous utiliserez* ***SDL_DOUBLEBUF****. La fonction renvoie un pointeur sur une surface SDL servant à dessiner dessus. Celle ci sera utilisée par sTileZ par la suite pour dessiner les cartes et les sprites.*

Nous n'entrons volontairement pas dans les détails car ce TP n'a pas pour but de vous apprendre à utiliser la SDL, seulement les parties indispensables. Vous êtes néanmoins fortement encouragés à suivre un tutoriel pour apprendre à vous en servir correctement ! (le site du zéro constitue une très bonne introduction)

Écrivez un main qui initialise la SDL, ouvre une fenêtre et quitte la SDL.

Pourquoi la fenêtre disparait t-elle immédiatement ? Proposez une solution simple.

2 Affichage d'une carte



La bibliothèque sTileZ introduit plusieurs structures vous permettant d'afficher des cartes et des sprites. Conceptuellement, une carte est un tableau à 2 dimensions de tuiles. Une même tuile peut apparaitre à plusieurs endroits dans une carte. C'est pourquoi on stocke toutes les apparences possibles de tuiles dans ce qu'on appelle un tileset. C'est une sorte de palette de tuiles. La structure ***sZ_Tileset*** *représente un tileset au sein de sTileZ. La première étape est donc de charger un tileset depuis un fichier. Pour ce premier TP vous utiliserez le fichier* ***house.tileset*** *que vous placerez dans un répertoire* ***resources/tilesets*** *au coté de l'image* ***house.png****. La fonction :*

sZ_Error sZ_LoadTileset(sZ_Tileset* tileset, const char* filename) ;
vous permet de charger un tileset. ***tileset*** *doit être un pointeur vers une structure sZ_Tileset déclarée avant l'appel et* ***filename*** *le chemin vers le fichier* ***house.tileset*** *(relatif par rapport à l'endroit ou vous lancez l'exécutable en console).*

Voici un exemple plus concret :

```
sZ_Tileset tileset; /* Déclaration de la structure sZ_Tileset */
/* Chargement du tileset */
if(SZ_SUCCESS
    != sZ_LoadTileset(&tileset, "resources/tilesets/house.tileset"))
{
   /* Affichage d'un message d'erreur si echec */
   fprintf(stderr, "Erreur lors du chargement du tileset.");
   return EXIT_FAILURE;
}
```

Comme vous pouvez le voir, il faut vérifier que le chargement du tileset n'a pas échoué (sinon le programme risque de planter par la suite). L'énumération ***sZ_Error*** *énumère tous les codes d'erreurs que peuvent renvoyer les fonctions de sTileZ. Pour pouvoir exécuter le code précédent, il faut inclure* ***<sTileZ/sTileZ.h>****.*

L'étape suivante consiste à charger une carte. Les cartes sont représentées par la structure ***sZ_Map****. Vous utiliserez la carte* ***house.map*** *que vous placerez dans le répertoire* ***resources/maps****. Le chargement d'une carte est très similaire à celui d'un tileset.*
Il faut utiliser la fonction :

sZ_Error sZ_LoadMap(sZ_Map* map, const char* filename) ;

Il également indiquer sur quel tileset la carte est construite. C'est le tileset que vous avez chargé plus haut ! Pour cela il faut utiliser la fonction :

void sZ_SetMapTileset(sZ_Map* map, const sZ_Tileset* tileset) ;

map *est un pointeur sur la structure* ***sZ_Map*** *contenant la carte chargée et* ***tileset*** *est un pointeur sur la structure* ***sZ_Tileset*** *à utiliser.*

Chargez la carte et fixez son tileset.

Ensuite il va falloir créer une scène contenant cette carte. Une scène contient un pointeur vers une carte et des sprites. Vous allez en utiliser une pour afficher votre carte. Déclarez une variable ***scene*** *de type* ***sZ_Scene*** *et initialisez la avec la fonction :*

sZ_Error sZ_InitScene(sZ_Scene* scene, sZ_Map* map) ;

*La dernière phase d'initialisation avant de pouvoir afficher la carte consiste à créer une cible de rendu (****sZ_RenderTarget****) dans laquelle sTileZ pourra dessiner. Cette cible sera constituée d'un pointeur vers une* ***SDL_Surface****. Vous utiliserez le pointeur renvoyé par* ***SDL_SetVideoMode****. Créer une variable de type* ***sZ_RenderTarget*** *et utiliser la fonction suivante pour l'initialiser :*
void sZSDL_SetRenderTarget(sZ_RenderTarget* target, SDL_Surface* surface) ;

Cette fonction est déclarée dans ***<sTileZ/SDL/sZSDL.h>****, il faut donc l'inclure pour pouvoir l'utiliser.*

Nous pouvons enfin afficher ! Dans une boucle infinie appelez la fonction :
void sZ_DrawScene(sZ_Scene* scene, sZ_RenderTarget* target, uint x, uint y) ;
en lui passant les bon paramètre (votre scène et votre cible de rendu). Les paramètres "x" et "y" correspondent à la position de la caméra dans la carte. Mettez 0, 0 pour ce premier test. Appelez ensuite la ***SDL_Flip(screen) ;*** *qui s'occupera de mettre à jour la fenêtre et d'afficher votre carte.*

3 Affichage d'un sprite

Vous allez étoffez votre programme précédent pour afficher Link dans sa maison ! Il sera pour l'instant immobile (nous verrons comment le contrôler dans l'exercice 5).

De la même manière qu'une carte est construite sur un tileset, un sprite est construit sur un spriteset. Les spriteset sont décris dans les fichiers ****.spriteset*** *et sont accompagnés d'une image donnant toutes les animations possibles du sprite. Vous allez utiliser le fichier* ***Link.spriteset*** *et son image* ***link.spriteset****, que vous placerez dans le répertoire* ***resources/spritesets****. Pour afficher un sprite dans la scène, il faut d'abord charger son spriteset. La démarche est à nouveau la même, vous déclarez une variable* ***spriteset*** *de type sZ_Spriteset et vous chargez le fichier en utilisant la fonction :*

sZ_Error sZ_LoadSpriteset(sZ_Spriteset* spriteset, const char* filename) ;
Ensuite vous ajoutez le sprite à la scène en utilisant la fonction :

sZ_Sprite* sZ_AddSprite(sZ_Scene* scene, const sZ_Spriteset* spriteset, uint x, uint y, uint layer) ;
(x, y) *est la position du sprite (en pixels) dans la carte ((300, 200) est une bonne valeur pour cette carte).* ***layer*** *est l'index de la couche sur laquelle placer le sprite (une carte est composé de plusieurs couches, pour faire des ponts par exemple). La fonction renvoie un pointeur vers le sprite ajouté. Si NULL est renvoyé, la fonction à échouée.*
Vous utiliserez donc la fonction comme ça :

sZ_Sprite* link = sZ_AddSprite(&scene, &spriteset, 300, 200, 0) ;

Tout ce code doit être placé avant votre boucle infinie d'affichage. Exécuter le programme et admirez Link attendant patiemment de pouvoir débuter son aventure !

4 Utiliser une camera et la boucle d'affichage de sZSDL

Comme vous avez pu le constater, le fait d'utiliser une boucle infinie empêche de fermer proprement le jeu, c'est à dire en utilisant la croix ! Pour faire cela, il faudrait gérer les évènements, mais cela viendra à l'exercice 5. L'extension SDL de sTileZ vous permet d'encapsuler la boucle d'affichage et de lui dire de quitter la boucle lorsque l'utilisateur clique sur la croix.

Le prototype de la fonction encapsulant la boucle est le suivant :

sZSDL_Error sZSDL_RunDisplayLoop(sZ_Scene* scene, const sZ_Camera* camera, sZ_RenderTarget* target) ;

Il suffit de l'appeler et il dessinera la scène en boucle 60 fois par secondes. Elle prend en paramètre un pointeur vers une camera. Une camera permet de changer le point de vue d'affichage dans la carte. Par la suite on s'arrangera pour qu'elle suive Link. Pour l'instant vous allez initialiser une variable de type sZ_Camera de manière à la placer en (0, 0).

Pour se faire il faut utiliser la fonction :

void sZ_InitCamera(sZ_Camera* camera, int x, int y, uint width, uint height, uint worldWidth, uint worldHeight) ;
(x, y) est la position (en pixel) de la camera dans la carte. (width, height) sont les dimensions de la vue. Vous prendrez ***width = screen->w*** *et* ***height = screen->h****, qui correspondent aux dimensions de la fenêtre. (worldWidth, worldHeight) sont les dimensions "du monde" dans lequel la caméra évolue. Dans notre cas, ce monde est la carte. Les dimensions doivent être spécifié en pixels. Vous pouvez les récupérer en utilisant les fonctions :*

uint sZ_GetMapWidthInPixels(const sZ_Map* map) ;

uint sZ_GetMapHeightInPixels(const sZ_Map* map) ;

Créez une caméra, initialisez la et remplacez votre boucle infinie par un appel à ***sZSDL_RunDisplayLoop****.*

Dernière étape pour que la fenêtre se ferme bien, rajouter un appel à la fonction : ***void sZSDL_SetExitOnClose() ;*** *avant de lancer la boucle.*

Lorsque l'utilisateur cliquera sur la croix, la fonction ***sZSDL_RunDisplayLoop*** *se terminera. Vous serez alors de retour dans le main. Il faut alors libérer la mémoire allouée pour les différentes ressources chargées (carte, tileset, ...). Utilisez les fonctions de la forme :* ***void sZ_DeleteXXX(sZ_XXX* xxx) ;*** *(par exemple* ***sZ_DeleteScene(sZ_Scene* scene)****) pour libérez ces ressources.*

Vous pouvez vous amuser à lires les .h de la bibliothèque graphique pour voir tout ce qui vous est proposé !

▸ **Exercice 5.** *Déplacements*

Avoir un Link immobile c'est pas super amusant, on aimerait pouvoir le déplacer ! Pour cela il va falloir utiliser les écouteurs et manipuler un peu la SDL. Un écouteur est une fonction qui sera appelée par la boucle d'affichage à un moment particulier, par exemple lorsque le joueur appuie sur une touche de son clavier ou à chaque tour de boucle. Les écouteurs écoutant les évènements sont appelé des EventListener, ceux écoutant les tours de boucle sont des FrameListener.

1 Écouteurs d'évènements

Un EventListener est une fonction ayant un prototype de cette forme :

void ecouteur(const SDL_Event* event, void* data) ;

ecouteur *doit être le nom que vous donnez à votre écouteur. Si par exemple vous faites un écouteur pour afficher les positions des clics de souris, vous pourriez l'appeler* ***OnMouseClick****. Lorsque votre écouteur sera appelé,* ***event*** *pointera sur l'évènement SDL ayant déclenché l'écouteur et* ***data*** *pointera sur des données que vous fournissez lors de l'enregistrement de l'écouteur. Puisque le type de ces données est inconnue à sTileZ (c'est vous qui choisissez ce que vous envoyez !), un pointeur générique* ***void**** *est utilisé et devra être "casté" dans la fonction.*

Cela parait peut être un peu abstrait, nous allons donc passer à la pratique !

Tout d'abord il faut initialiser le gestionnaire d'écouteur avant le lancement de la boucle d'affichage :

InitListenerManager() ;

et après la boucle :

DeleteListenerManager() ;

Vous allez utilisez l'écouteur suivant :

```
void OnMouseClick(const SDL_Event* event, void* data) {
  printf("clic en : "%d %d\n", event->button.x, event->button.y);
}
```

Cet écouteur affiche la position d'un clic quand il survient. La structure ***SDL_Event*** *possède de nombreux champs, un pour chaque type d'évènement possible. Plutôt que de tous les décrire ici, voici une page internet décrivant la structure : http ://sdl.beuc.net/sdl.wiki/SDL_Event*

Vous devez donc définir la fonction OnMouseClick avant le main.

Ensuite retour dans le main, juste après l'appel à InitListenerManager que vous avez ajouté précédemment. Vous allez ajouter l'écouteur en utilisant la fonction :

sZSDL_Error sZSDL_AddEventListener(Uint8 type, sZSDL_EventListener listener, void* data) ;

type *correspond au type d'évènement à écouter. Les différent types sont énuméré par la SDL et sont disponible sur la page mentionné plus haut. Ici nous allons utiliser* ***SDL_MOUSEBUTTONDOWN*** *qui correspond au "pressage" d'un bouton de la souris.* ***listener*** *est la fonction écouteur à utiliser.* ***data*** *est un pointeur vers les données que vous voulez passer à votre écouteur. Ici l'écouteur n'a besoin d'aucune donnée. Vous passerez donc NULL :*

sZSDL_AddEventListener(SDL_MOUSEBUTTONDOWN, OnMouseClick, NULL) ;

Lancez votre programme et constatez l'affichage lors du clic de souris !

2 Faire bouger Link

Écouter des clics c'est marrant, mais dans Zelda ça sert à rien ! Ce qu'on voudrait c'est déplacer Link lorsque le joueur presse les touches directionnelles. On pourrait décider d'écouter l'évènement ***SDL_KEYDOWN*** *qui correspond à l'appui sur une touche du clavier, mais cet évènement n'est lancé qu'une fois lors de l'appui. Or Link se déplace tant que la touche reste enfoncée. La solution est d'utiliser un FrameListener, qui sera appelé à chaque tour de la boucle d'affichage. Dans ce listener on regardera si la touche qui nous intéresse est enfoncée, si oui on fera avancer Link.*

Un FrameListener est une fonction ayant un prototype de la forme :

void ecouteur(uint framecount, uint milliseconds, void* data) ;

framecount *est l'index du tour de boucle courant (au premier tour de la boucle d'affichage il vaudra 0, au deuxième il vaudra 1, ...).* ***milliseconds*** *est le nombre de milliseconds qui se sont écoulées depuis le dernier appel à votre écouteur (cela permet d'adapter les vitesses de déplacement en fonction du temps et non de la cadence d'affichage).* ***data*** *joue le même rôle que pour les EventListener.*

Pour faire bouger Link, vous allez utiliser ***data*** *pour passer le* ***sZ_Sprite*** *correspondant à votre écouteur. Voila l'écouteur à moitié écrit :*

```
void MoveLink(uint framecount, uint milliseconds, void* data) {
  /* On sait que data pointe sur le sprite de link, on le caste */
  sZ_Sprite* link = (sZ_Sprite*) data;

  if (SDL_GetKeyState(NULL)[SDLK_UP])
  {
    sZ_MoveSprite(link, 0, -5);
  }
}
```

Pour ajouter cet écouteur, il faut utiliser la fonction :

sZSDL_Error sZSDL_AddEnterFrameListener(sZSDL_FrameListener listener, void* data) ;

Vous l'utiliserez de la manière suivante :

sZSDL_AddEnterFrameListener(MoveLink, link) ;

(en supposant que link est le sZ_Sprite renvoyé par sZ_AddSprite, voir exercice 4).*

Lancez le programme et constatez qu'en appuyant sur la flèche haut, Link se déplace vers le haut. Améliorez la fonction d'écoute pour déplacer Link dans toutes les directions (en utilisant les constante SDLK_DOWN, SDLK_LEFT et SDLK_RIGHT pour les directions). Le prototype de ***sZ_MoveSprite*** *est le suivant :*

void sZ_MoveSprite(sZ_Sprite* sprite, int dX, int dY) ;

3 Animer Link

Link bouge, mais il ne marche pas ! On va donc l'animer en utilisant des fonctions de sTileZ. Un sprite est constitué d'animations. En utilisant la fonction :

void sZ_NextSpriteFrame(sZ_Sprite* sprite) ;

vous pouvez animer le sprite dans votre écouteur lorsqu'il se déplace. Pour l'instant il ne change pas de direction !

4 Animer (mieux) Link

Link est animé un peu rapidement (il change d'image 60 fois par secondes !). Il faut limiter un peu ça en faisant en sorte qu'il ne change d'animation que tous les 5 tours de boucles (par exemple). Modifiez l'écouteur pour attendre ce résultat (utilisez "framecount" et l'operation modulo (

5 Changer Link de direction

Pour que Link change de direction, il faut ajouter un EventListener sur ***SDL_KEYDOWN****. En fonction de la touche pressée, il faut modifier l'animation de Link en utilisant la fonction :*

void sZ_SetSpriteAnimation(sZ_Sprite* sprite, uint animation, uint frame) ;

animation *est l'index de l'animation à utiliser. Observez le fichier* ***link.png****, il y a 4 animations, constituées chacune de 4 frames. Les animations sont indexées de 0 à 3, de haut en bas.*

Écrivez une fonction ***ChangeLinkDirection*** *écoutant l'évènement* ***SDL_KEYDOWN*** *(même principe que celle qui écoutait les clic de souris) et qui modifie l'animation de Link en fonction de la touche pressée. Lisez la page http ://sdl.beuc.net/sdl.wiki/SDL_Event pour savoir à quel champs de la structure* ***SDL_Event*** *il faut acceder.*

6 Collisions

Attention on passe à un peu plus difficile : intégrez votre gestion de collision pour que les déplacements de Link ne soient effectués que si Link n'entre pas en collision avec un décor.

7 Si vous avez le temps

Il reste encore des défauts, par exemple lorsque Link arrête de bouger, il se stope en plein milieu d'animation. Essayez de corriger ça (indice : il faut utiliser un nouvel écouteur...)

Faites en sorte que la camera suive Link (Link doit être au centre de la caméra). Pour cela il faudra utiliser les fonctions de déplacement de camera (voir sTileZ/Camera.h).