Anas Katim
Antoine Camusat
Aleksi Mouvier

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet informatique : Insacar

STPI 2.C

Année 2019-2020

Table des matières

Introduction

Dans le cadre du projet informatique nous avons choisi de développer un jeu de voiture dans lequel un ou deux joueurs contrôlent une voiture sur un circuit en vue de dessus dans le but de faire le meilleur temps possible. De nombreuses fonctionnalités ont ainsi été imaginées comme, un déplacement de la caméra, un HUD, des collisions, etc.

L'objectif était de pouvoir exploiter ce que nous avions appris en algorithmique et sur le langage Pascal lors de notre première année, et de le combiner aux bases sur la SDL vues pendant ce semestre, afin de pouvoir obtenir un jeu que nous considérons satisfaisant autant en termes de graphisme qu'en termes de gameplay.

Nous avons rencontré et surmonté de nombreuses difficultés durant la conception de ce programme, sur lesquelles nous reviendront durant ce rapport.

Dans un premier temps, nous reprendrons le cahier des charges réalisé auparavant en faisant le bilan sur les fonctionnalités implémentées. Ensuite, nous traiterons de la conception globale avec l'évolution de l'analyse descendante et des structures de données utilisées. Puis, nous détaillerons le fonctionnement des différentes procédures et fonctions. Enfin, nous reviendrons sur notre expérience du travail de groupe lors de ce projet.

I/ Cahier des charges

Nous allons donc commencer par reprendre le cahier des charges que nous nous étions imposés ainsi que le calendrier des versions associées.

Fonctionnalités attendues :

- Menu principal
- Affichage tutoriel
- Consultation des scores (utilisation de fichiers).
- Configuration de sa partie (choix du circuit, de sa voiture).
- Mode contre-la-montre
- Mode 1 vs 1
- Contrôle voiture(s) grâce au clavier.
- Déplacement caméra à travers le circuit.
- Mouvements physiques de la voiture (déplacement, collisions, accélération, etc.)
- HUD (ensemble d'informations affichées à l'écran pendant la course (vitesse, tours, temps, etc.)).

Calendrier des versions prévues :

- Version n°1 / date prévue : mi-octobre
  - Contrôle voiture grâce au clavier.
  - Déplacement libre (sans collisions) de la voiture à travers un circuit
- Version n°2 / date prévue : mi-novembre
  - Menu principal
  - Affichage tutoriel
  - Consultation des scores (utilisation de fichiers).
  - Configuration de sa partie (choix du circuit et de sa voiture).
  - Mode contre-la-montre
  - Déplacement caméra à travers le circuit (mode contre-la-montre).
  - HUD
- Version n°3 / date prévue : début décembre
  - Plusieurs circuits
  - Collisions

- Mode 1 vs 1
- Déplacement caméra à travers le circuit (mode 1vs1)
- HUD du joueur 2.

L'ensemble des fonctionnalités prévues ont été ajoutées à notre jeu. Le cahier des charges a donc été respecté.

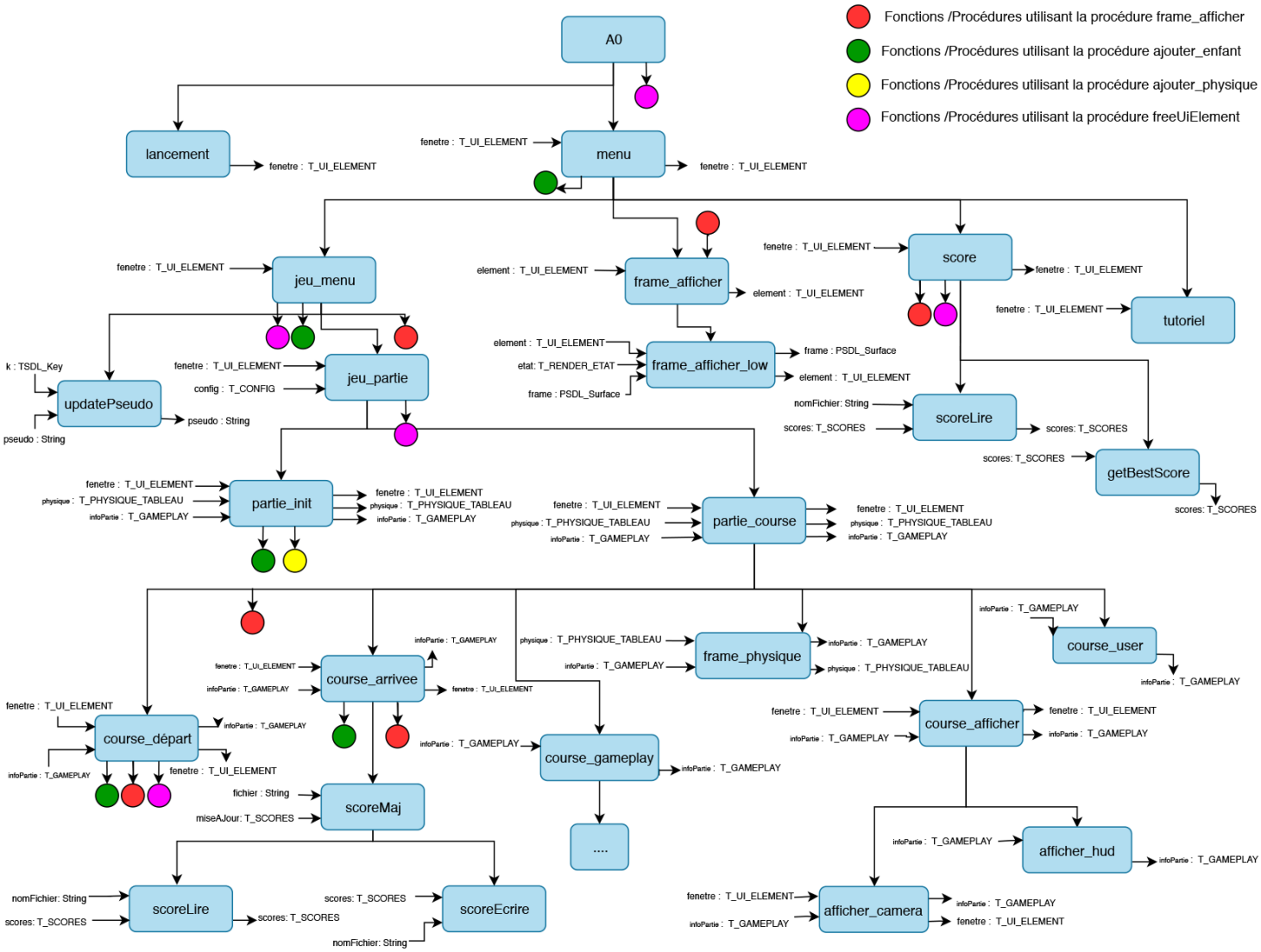
Concernant le calendrier, il n'a pas été suivi dans son intégralité. En effet, si la version n°1 a été réalisée à temps, nous avons pris un peu de retard sur les deux autres.

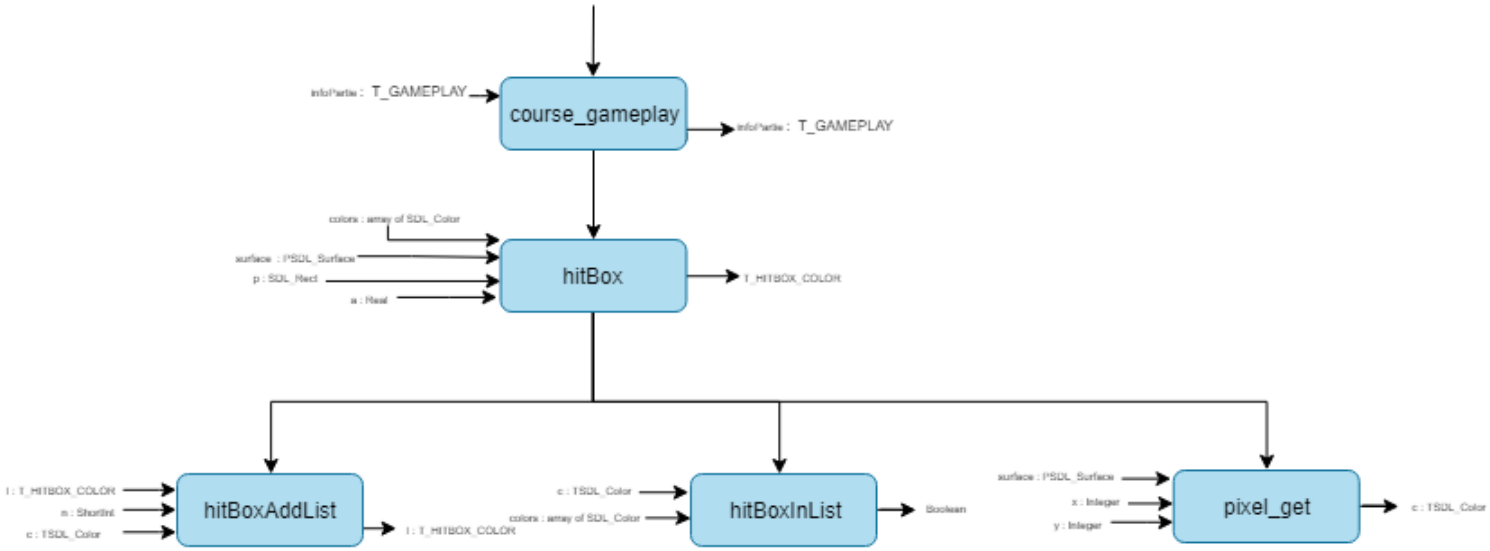
Ce léger retard s'explique par le fait que nous avons implémenté l'ensemble des procédures et fonctions directement dans l'optique de réaliser un mode à 2 joueurs. Ce choix a finalement été judicieux car nous avons pu facilement adapter notre programme pour ce mode. La transition entre la version 2 et 3 s'est donc bien déroulée.

II)Conception globale

1)Analyse descendante

Voici l'analyse descendante finale qui représente l'architecture de notre jeu.





En observant l'analyse descendante réalisée en Septembre, on remarque que celle-ci n'a pas subi de profonds changements. En effet, l'architecture globale de notre jeu n'a pas changé. Nous avons surtout effectué des modifications sur les structures de données (voir ci-après) et sur les entrées-sorties de nos procédures et fonctions. De plus, les seules fonctions qui ont été ajoutées sont des outils (unit tools) permettant de faciliter l'implémentation et d'améliorer la clarté du code (exemple : procédures ajouter_enfant, ajouter_physique, updatePseudo, etc.)

2)Lexique / Informations

- Un état est un couple de coordonnées (x,y) et dimensions (w,h)
- Une surface (TSDL_Surface) est un type de la librairie SDL contenant la texture et les informations d'un élément affichable sur l'écran. La librairie fonctionnant par pointeur, on sous-entend ici surface comme un pointeur vers une TSDL_Surface (PSDL_Surface).
- Lors des déplacements, les véhicules utilisent un repère polaire placé sur la voiture (pour l'incrémentation de la vitesse par exemple)
- Les surfaces (T_UI_ELEMENT) ont toutes comme origine leur coin en haut à gauche. Leur référentiel est celui de leur parent (donc lié à la fenêtre)
- Les éléments physique (T_PHYSIQUE_ELEMENT) ont comme référentiel le coin en haut à gauche de l'image du circuit de base (sans zoom).
- Le temps d'affichage (de la SDL) est séparé du temps physique, ce qui permet d'avoir une physique constante, non liée à la vitesse de la machine ou à la fréquence de rafraîchissement.

3) Structure de données

T_UI_ELEMENT :

Ce type contient tout ce qui est nécessaire à l'affichage d'une surface : un état, le pointeur vers la surface, son type (image, texte, couleur) sa valeur et police pour le texte, une couleur (rgb). Il contient aussi un enregistrement style afin par exemple de régler la transparence de la surface (T_RENDER_STYLE).

Enfin, un tableau (T_UI_TABLEAU) contient des 'enfants' c'est-à-dire des T_UI_ELEMENT dont les coordonnées ont pour origines les coordonnées de l'élément parent. C'est cette architecture en arbre qui permet un rendu facile dans frame_afficher. On stocke aussi un pointeur (P_UI_ELEMENT) vers l'élément parent pour tracer celui-ci (isInElement)

T_GAMEPLAY :

Ce type contient toute l'information nécessaire à une partie : le temps, le niveau de zoom, le statut de la partie (actif ou non), l'indice du premier joueur, des pointeurs vers les éléments de HUD variable dans le temps. Il sert aussi à stocker la surface non modifiée du circuit ainsi que la version actuellement affichée (pointeur vers la surface d'un T_UI_ELEMENT). Cela permet de ne pas altérer la surface lors du rotozoom. Ce type contient aussi un pointeur vers un T_CONFIG, contenant les choix de l'utilisateur pour sa partie.

Il contient enfin un tableau de T_JOUEUR, les joueurs de la partie en cours.

T_JOUEUR :

Contient les informations relatives à un joueur : temps (secteurs, tours), son nom, des pointeurs vers les éléments de HUD liés au joueur (vitesse) et enfin les informations relatives au skin utilisé surface de base, celle actuellement utilisée (zoom / rotation). On y ajoute des pointeurs vers le T_UI_ELEMENT et le T_PHYSIQUE_ELEMENT de la voiture.

T_CONFIG :

Contient les informations que l'utilisateur remplis lors de jeu_menu. (Circuits, joueurs)

T_PHYSIQUE_ELEMENT :

Contient des variables correspondant aux divers systèmes de coordonnées possibles (cartésien ou polaire) ainsi que la valeur numérique de leur dérivées (vitesse). Ce type est contenu dans un T_PHYSIQUE_TABLEAU

T_HITBOX_COLOR :

Ce type sert lors du calcul des hitbox : il contient une liste d'enregistrement, associant un point de la hitbox avec la couleur trouvée.

T_SCORE :

C'est le type stocké dans les fichiers de store, c'est un enregistrement contenant le nom du joueur et son temps. (T_SCORES est un tableau de T_SCORE)

III)Détail des procédures et fonctions

FUNCTION SECONDE_TO_TEMPS(T: LONGINT): STRING;

Transforme le temps (stocké comme un entier, en centième de seconde) en une chaîne de caractère au format mm: ss: ccc (m minutes, s secondes, ccc centièmes).

On obtient la valeur des minutes, secondes et centièmes via les opérateurs modulo et diviseur, et on assure le format de sortie via l'ajout éventuel de 0 pour les minutes et secondes.

PROCEDURE AJOUTER_PHYSIQUE(VAR PHYSIQUE: T_PHYSIQUE_TABLEAU);

Ajoute un nouvel élément physique au tableau.

La procédure sauvegarde l'ancien pointeur, demande au système l'allocation d'un bloc de mémoire d'une taille supérieure de 1 par rapport au précédent, recopie via une boucle l'adresse des anciens éléments physiques dans le nouveau bloc, demande et stocke l'adresse du nouvel élément physique dans la dernière case (la 'nouvelle' case) et initialise ces valeurs à 0. La procédure libère alors l'ancien bloc de pointeurs, et augmente la taille du tableau de 1.

PROCEDURE AJOUTER_ENFANT(VAR ELEMENT: T_UI_ELEMENT);

Ajoute un enfant à un élément d'UI.

La procédure sauvegarde l'ancien pointeur, demande au système l'allocation d'un bloc de mémoire d'une taille supérieure de 1 par rapport au précédent, recopie via une boucle l'adresse des anciens éléments physiques dans le nouveau bloc, demande et stocke l'adresse du nouvel enfant dans la dernière case (la 'nouvelle' case) et initialise ses valeurs. La procédure libère alors l'ancien bloc de pointeurs, et augmente le nombre d'enfants de 1.

FUNCTION ISINELEMENT(ELEMENT: T_UI_ELEMENT; X, Y: INTEGER): BOOLEAN;

Teste si un élément d'UI se situe au coordonnées indiquées (référentiel fenêtre)

En premier, la procédure soustrait récursivement l'ensemble des coordonnées x,y des parents de l'élément au coordonnées des positions demandées, afin de pouvoir par la suite tester si celles-ci se situent dans le rectangle x,x+w,y,y+h de l'élément cible.

FUNCTION HITBOX(SURFACE: PSDL_SURFACE; P: SDL_RECT; A: REAL; COLORS: ARRAY OF TSDL_COLOR): T_HITBOX_COLOR;

Compare les couleurs d'un rectangle de 12 points (cf types) , d'état p et d'angle a, d'une surface avec les couleurs fournies, renvoie un T _HITBOX_COLOR, liant chaque couleur trouvée avec son point.

La fonction initialise d'abord les données nécessaires, en calculant par exemple les fonctions trigonométriques afin d'optimiser les performances. Elle va ensuite tester chacun des 12 points : le point numéro 1 d'abords, puis 2 bande de 5 points de chaque côté (de la voiture) et enfin le point

arrière. La couleur de chaque point est récupérée via pixel_get, testée via hitBoxInList, et enfin ajouté au tableau de sortie via hitBoxAddList.

Cette fonction est celle des collisions, en effet elle permet de tester sur quel 'sol' est notre voiture : on teste la couleur des points sur l'image du circuit.

FUNCTION PIXEL_GET(SURFACE: PSDL_SURFACE; X,Y: INTEGER): TSDL_COLOR;

Renvoie la couleur d'un pixel sur une surface

La fonction commence par bloquer la surface si cela est requis, récupère le pointeur vers les pixels de la surface SDL : Dans SDL, les pixels d'une surface de w*h sont stockés sous forme d'un tableau a 1 dimension, de taille w*h. On lit ensuite le pixel correspondant à la coordonnée demandée. Le pixel est stocké comme un entier sur 4 octets, 3 pour chaque couleurs, et 1 pour la transparence (alpha). On convertit cette entier en TSDL_COLOR (r,g,b,a).

Nous avons rencontré des problèmes de boutisme avec cette fonction, en effet, les architectures x86(-64) ou ARM utilisé sur nos ordinateurs stockent les bits de poids faible en tête, ce qui a pour effet d'inverser la valeur de r et b. On utilise la compilation conditionnelle permise par pascal afin d'inverser ces 2 valeurs. On débloque la surface à la fin de la lecture.

FUNCTION HITBOXINLIST(C: TSDL_COLOR; COLORS: ARRAY OF TSDL_COLOR): BOOLEAN;

Teste si une couleur est présente dans une liste de couleurs.

Cette fonction boucle simplement dans une liste de couleurs et renvoie vrai si elle a trouvé une couleur spécifique passée en paramètre.

PROCEDURE HITBOXADDLIST(VAR L: T_HITBOX_COLOR; N: SHORTINT; C: TSDL_COLOR);

Ajoute un élément à un T_HITBOX_COLOR.

La procédure augmente la taille du tableau de 1 et initialise la case avec le numéro du point et la couleur passé en paramètre.

FUNCTION ISSAMECOLOR(A: TSDL_COLOR; B: TSDL_COLOR): BOOLEAN;

Compare 2 couleurs

Ici, on renvoie vrai uniquement si les 3 valeurs de couleurs sont identiques (gain de lisibilité dans le code)

FUNCTION ZOOMMIN(A,B: REAL): DOUBLE;

Renvoie le minimum borné entre deux valeurs.

Renvoie min(min(a,b),0.3,1)

PROCEDURE IMAGELOAD(CHEMIN: STRING; VAR SURFACE: PSDL_SURFACE; ALPHA: BOOLEAN);

Charge et transforme une image png en surface prête à l'affichage.

La procédure charge l'image via la librairie sdl_image, et transforme celle-ci via displayFormat (avec un masque alpha ou non). Cette transformation permet à l'image d'être adaptée au format d'affichage de l'écran, on augmenter ainsi les performances.

PROCEDURE UPDATEPSEUDO(K : TSDLKEY; VAR PSEUDO: STRING);

Met à jour un pseudo (chaîne de caractère) en fonction d'un appui clavier.

La procédure récupère d'abord l'état actuel du clavier, c'est-à-dire un tableau de booléens correspondant à chaque touche. Elle teste ensuite suivant les valeurs de la table ASCII la touche (k) pressée et ajoute cette lettre au pseudo (a-z, A-Z avec shift, 0-9).

Les claviers sous Windows renvoyant toujours des valeurs correspondant à un clavier qwerty, on utilise la compilation conditionnelle permise par Pascal afin d'ajouter la bonne lettre.

PROCEDURE SCOREMAJ(FICHIER: STRING; MISEAJOUR: T_SCORES);

Met à jour le fichier de scores en fonction des temps réalisés.

La fonction va d'abord lire le contenu du fichier (scoreLire), puis initialiser un tableau de booléens (dont la taille est le nombre joueurs/temps).

Par la suite, on boucle dans chaque score du fichier, on teste si l'utilisateur est le même (on bascule l'indice de booléen à vrai) et si son temps est inférieur, si cela est le cas, on modifie le tableau de scores.

Si à la fin de la boucle dans le fichier on n'a pas trouvé tous les utilisateurs (c'est-à-dire que l'on a un nouvel utilisateur) on ajoute leurs temps à la fin du tableau de score.

Enfin, on réécrit le fichier de score (scoreEcrire)

PROCEDURE SCORELIRE(NOMFICHIER: STRING; VAR SCORES: T_SCORES);

Lit le contenu d'un fichier score et renvoie un tableau T_SCORES.

La fonction assigne a une variable le fichier, positionne le curseur de lecture au début, et lit chaque score afin de remplir un tableau T_SCORES de même taille, avant de fermer le fichier.

PROCEDURE SCOREECRIRE(NOMFICHIER: STRING; SCORES: T_SCORES);

Écrit un variable T_SCORES dans un fichier

La fonction assigne à une variable le fichier, l'ouvre en écriture et écrit chaque score dans le fichier avant de le refermer.

FUNCTION MIN(LISTE : ARRAY OF LONGINT): LONGINT;

Renvoie le minimum d'un tableau d'entier

PROCEDURE FREEUIELEMENT(VAR ELEMENT: P_UI_ELEMENT);

Libère la mémoire récursivement d'un T_UI_ELEMENT

La procédure assigne la valeur 0 au tableau d'enfants, puis libère la surface de l'élément via SDL_FreeSurface, libère la police via TTF_Close, et enfin détruit l'élément en libérant l'élément.

PROCEDURE FREEINFOPARTIE(VAR INFOPARTIE: T_GAMEPLAY);

Libère la mémoire de T_GAMEPLAY

On libère la surface de la map (SDL_FreeSurface), le skin de chaque joueur, et enfin on détruit le tableau de joueurs.

PROCEDURE GETBESTSCORE(VAR SCORES: T_SCORES);

Algorithme de tri sur T_SCORES : implémentation d'un tri par insertion

PROCEDURE FRAME_AFFICHER_LOW(VAR ELEMENT: T_UI_ELEMENT; VAR FRAME: PSDL_SURFACE; ETAT: T_RENDER_ETAT);

Cette procédure crée la surface à afficher sur l'écran a partir d'un T_UI_ELEMENT (fenêtre) récursivement.

Cette fonction est une mise en application de la procédure SDL_BlitSurface : elle empile les surfaces afin d'en créer une seule (frame), qui sera affichée sur l'écran. On va d'abord appliquer les paramètres à nos surfaces (couleur pour un type 'couleur', texte pour un type 'texte'), et ensuite on applique les styles aux surfaces (transparence), puis on colle celle-ci sur la surface principale (frame)

La position sur laquelle on colle la surface est passée récursivement et incrémentée à chaque fois.

Si un élément est de type texte, on incrémente de plus cette position avec la largeur du texte, afin de pouvoir positionner des éléments au bout de celui-ci (utile pour les curseurs des textes par exemple)

PROCEDURE FRAME_AFFICHER(VAR ELEMENT: T_UI_ELEMENT);

Cette procédure initialise et prépare frame_afficher_low

On initialise les valeurs de la position de collage à 0 puis on lance frame_afficher_low.

PROCEDURE AFFICHER_HUD(VAR INFOPARTIE: T_GAMEPLAY);

Affiche le HUD

On modifie la valeur du temps global, du numéro de tour actuel, du premier joueur, de la vitesse et du temps pour le secteur de chaque joueur.

PROCEDURE AFFICHER_CAMERA(VAR INFOPARTIE: T_GAMEPLAY; VAR FENETRE: T_UI_ELEMENT);

Gère l'affichage de la caméra (mouvements et échelle)

La fonction calcule d'abord l'échelle (zoom) adapté a la situation (1 / 2 joueurs) :

Après avoir déterminé le centre de l'écran (référentiel map), on impose une constante de distance (%) entre la voiture et le bord de l'écran en hauteur et largeur. On calcule le zoom nécessaire à avoir cette distance, et on va créer une map de la bonne taille à l'aide de rotozoomSurface à partir de la map de base (infopartie.map.base). On prend soin de libérer l'ancienne map et de sauvegarder le zoom (on ne crée pas une map à chaque frame, seulement si le zoom change suffisamment (via arrondi). En réalité, ce n'est pas la camera qui bouge, mais l'ensemble du circuit.

On va ensuite placer les joueurs, pour cela, on libère leur ancienne surface, on en crée des nouvelles (rotozoom) et on les place à leur position (changement de référentiel).

PROCEDURE COURSE_AFFICHER(VAR INFOPARTIE: T_GAMEPLAY; VAR FENETRE: T_UI_ELEMENT);

Cette procédure génère une image à afficher.

On appelle successivement afficher_camera puis afficher_hud

PROCEDURE COURSE_GAMEPLAY(VAR INFOPARTIE: T_GAMEPLAY);

Gère les évènements de jeu

On définit les couleurs de map importantes (lignes de départ / Secteurs), puis on va tester si un joueur passe ces lignes via la procédure hitbox. En fonction de la ligne, on incrémente son nombre de tour, son secteur, et on détecte la fin de partie.

PROCEDURE FRAME_PHYSIQUE(VAR PHYSIQUE: T_PHYSIQUE_TABLEAU; VAR INFOPARTIE: T_GAMEPLAY);

Calcule la physique du jeu à chaque boucle

On calcule la physique du jeu à chaque frame. On teste pour chaque joueur les collisions avec l'herbe du circuit, et on applique suivant le résultat un coefficient de frottement différent (air ou terre) à sa vitesse.

Ensuite, on va calculer la position de tous les éléments physique, en ajoutant dt*vitesse à notre position

PROCEDURE COURSE_USER(VAR INFOPARTIE: T_GAMEPLAY);

Convertit les entrées utilisateur en valeurs de infoPartie.

On lit les événements SDL et on incrémente les vitesses des joueurs / leur angle en fonction de la touche pressée. La touche P permet de forcer l'arrêt de la partie.

PROCEDURE COURSE_ARRIVEE(VAR INFOPARTIE: T_GAMEPLAY; VAR FENETRE: T_UI_ELEMENT);

Affiche le tableau de fin de partie.

On cache le HUD, on remplit un tableau T_SCORES, on récupère les meilleurs scores du circuit et enfin on affiche la surface.

On boucle ensuite en attendant une validation par l'utilisateur, auquel cas on met à jour le fichier de score.

PROCEDURE COURSE_DEPART(VAR INFOPARTIE: T_GAMEPLAY; VAR FENETRE: T_UI_ELEMENT);

On génère une image du circuit/voitures seulement (afficher_camera), on cache le HUD, on effectue un décompte en affichant les feux tricolores, puis on libère les surfaces et enfin on ré-affiche le HUD.

On actualise ensuite les temps de début de partie et last (dernier temps) à la valeur actuelle de la SDL

PROCEDURE PARTIE_COURSE(VAR INFOPARTIE: T_GAMEPLAY; VAR PHYSIQUE: T_PHYSIQUE_TABLEAU; VAR FENETRE: T_UI_ELEMENT);

Cette procédure contient la boucle principale de la course :

en premier, on lance la procédure de départ (couse_départ), puis on boucle sur les procédures suivantes :

1)On met à jour le temps du jeu (dt l'intervalle depuis la dernière boucle), last le temps actuel

2)On récupère les entrées utilisateurs : course_user

3)On met à jour la physique : frame_physique

4)On met à jour les informations de la partie : course_gameplay

5)On génère une image à afficher : course_afficher

6)On affiche cette image : frame_afficher

Et enfin on calcule le temps à attendre afin de maintenir une cadence de rafraîchissement de l'écran C_REFRESHRATE

PROCEDURE PARTIE_INIT(VAR INFOPARTIE: T_GAMEPLAY; VAR PHYSIQUE: T_PHYSIQUE_TABLEAU; VAR FENETRE: T_UI_ELEMENT);

On initialise les données de la partie.

On initialise infoPartie, la couleur de la fenêtre, on charge la map dans infoPartie, les skins des joueurs ainsi que leurs données :temps secteurs, nbTour, position (en fonction du circuit) et le HUD.

PROCEDURE JEU_PARTIE(CONFIG: T_CONFIG; FENETRE: T_UI_ELEMENT);

Procédure correspondant a une partie.

On lie la config à infoPartie, on initialise via partie_init et on lance une partie avec partie_course

En fin de partie, on libère les surface et infoPartie avec freeUiElement et freeInfoPartie

PROCEDURE JEU_MENU(FENETRE: T_UI_ELEMENT);

Affiche le menu de configuration de la partie

On charge les textures des menus roulants au préalable afin d'éviter toute fuite mémoire, on initialise les couleurs, les positions et les enfants dans fenêtre. Enfin, on affiche celles-ci.

Lorsqu'un utilisateur clic sur un élément, on met a jour sa valeur, et on teste chaque élément à la fin de la boucle afin de mettre à jour l'affichage / la configuration en fonction des choix de l'utilisateur.

Lorsque l'utilisateur valide sa partie (entrée) on initialise la config et on lance jeu_partie.

PROCEDURE TUTORIEL(FENETRE: T_UI_ELEMENT);

Affiche le tutoriel

On lit dans un fichier le tutoriel, et on affiche les images correspondantes au numéros de ligne dans le texte.

PROCEDURE SCORE(FENETRE: T_UI_ELEMENT);

Affiche les records de score par circuit.

Lit les données a l'aide de scoreLire, et getBestScore, puis les affiche dans des panneaux (1 par circuit)

PROCEDURE MENU(VAR FENETRE: T_UI_ELEMENT);

Menu principal

Affiche 4 boutons correspondants au fonctions principales du jeu.

FUNCTION LANCEMENT(): T_UI_ELEMENT;

Initialise SDL

Initialise et test le lancement de la librairie SDL (lance la fenetre)

PROGRAMME PRINCIPAL A0

Initialise la fenetre (lancement) puis lance le menu

en fin de jeu, libère toutes les surfaces, libère la librairie TTF et la SDL.

IV/ Travail en groupe et utilisation d'un logiciel de gestion de versions

Au cours de ce projet, nous nous sommes assez bien réparti le travail selon les capacités de chacun. De plus, nous nous sommes souvent réunis en salle informatique pour travailler et avancer ensemble. Cet aspect du projet s'est bien déroulé.

Le point le plus difficile a été de coordonner les différentes versions de chacun. En effet, travailler à plusieurs sur le même code source peut rapidement mener à des problèmes comme des modifications qui écrasent celle d'une autre personne par exemple.

Pour éviter cela, nous avons décidé d'utiliser un logiciel de gestion de version. Ces logiciels permettent de travailler à plusieurs sans risquer de perdre des informations si deux personnes travaillent en même temps sur un même fichier. De plus, les logiciels de gestion de version permettent de suivre l'évolution d'un code source à chaque modification. Cela peut permettre de revenir en arrière dans les versions en cas de problème par exemple.

Nous avons donc utilisé « Git » un des logiciels de gestion de versions les plus connus. Ce dernier a été difficile à prendre en main au début mais nous avons vite observé son intérêt dans le projet.

Finalement, cet outil a grandement facilité la collaboration dans notre projet et sans lui il aurait été très difficile de travailler efficacement en groupe.

Conclusion

Pour conclure, ce projet informatique nous a permis d'approfondir et mettre en pratique nos connaissances en Pascal et plus généralement en algorithmique.

De plus, ce projet aura été l'occasion de découvrir et prendre en main la bibliothèque SDL permettant de réaliser des jeux 2D assez satisfaisants en termes de graphiques.

Enfin, nous avons pu d'expérimenter le travail de groupe sur un projet informatique et apprendre des méthodes pour travailler efficacement avec notamment l'utilisation d'un logiciel de gestion de version.