Objectifs

- Maîtriser la vitesse d'exécution de l'application
- Contrôler la vitesse de rafraîchissement de l'affichage
- Calculer la dynamique des déplacements des joueurs

Il est impératif de valider chaque étape avant de passer à la suivante.

En cas de doute ou de difficulté, consultez votre enseignant.

Gestion du temps

La vitesse d'exécution d'un programme peut varier en fonction de la machine sur laquelle il s'exécute, mais aussi des ressources disponibles sur cette machine. Pour éviter que la vitesse de notre application ne varie nous allons cadencer son exécution. Pour cela, nous allons imposer un taux de rafraîchissement de 100 Hz (100 images par seconde). Comme pour les graphismes, la bibliothèque SFML nous offre des outils permettant de maîtriser la vitesse de l'application. Le type Clock permet de mesurer le temps écoulé :

```
sf::Clock Timer;
```

Pour remettre l'horloge à zero, utiliser la fonction Reset :

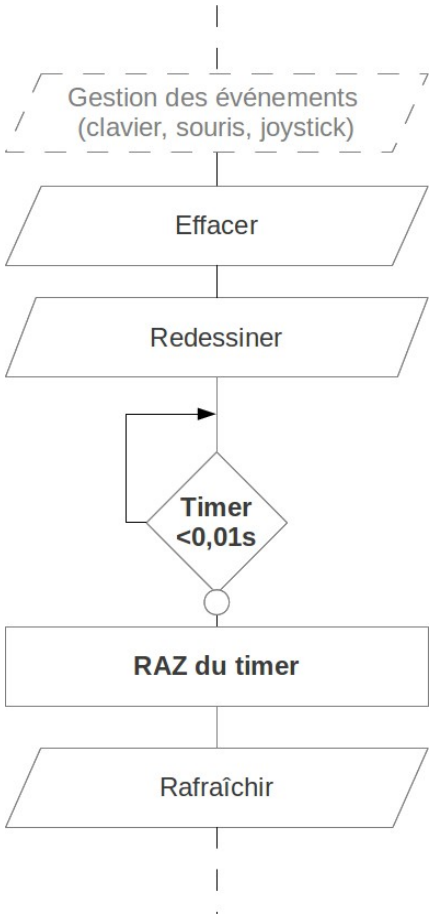
```
Timer.Reset();
```

Et enfin, la fonction GetElapsedTime retourne le temps écoulé en secondes depuis la dernière remise à zéro de l'horloge (lors de la création, l'horloge est initialisée à zéro) :

```
float t=Timer.GetElapsedTime();
```

La maîtrise du taux de rafraîchissement consiste à attendre 10 ms entre chaque rafraîchissement. L'horloge est créée avant la boucle principale. Si au moment de rafraîchir l'affichage, moins de 10 ms se sont écoulées, le programme va attendre. Dès que les 10 ms sont écoulées, l'horloge doit immédiatement être remise à zéro pour le prochain cycle.

Voici une vue générale du mécanisme :



Implémenter ce contrôle de la vitesse d'exécution et tester avec les lignes ci-dessous qui permettent de fermer automatiquement l'application au bout de 10 secondes (1000 cycles) :

```
[...]
sf::Clock Timer;
int i=0;
while (App.IsOpened())
{
      if (i++>1000) App.Close();
[...]
```

Exécuter le programme et mesurer la durée de vie de la fenêtre. Si celle-ci se ferme au bout de 10 secondes, c'est que le taux de rafraîchissement est bien à 100Hz, sinon révisez votre code. Lorsque vous avez validé le résultat, n'oubliez pas de supprimer les lignes de test.

Résolution de la fenêtre

Dans la suite du projet, nous allons avoir besoin des dimensions de la fenêtre pour déplacer les

éléments ou effectuer des calculs. Il est bien sûr possible d'utiliser directement les valeurs numériques dans le code, mais cette solution pose le problème d'un éventuel changement de dimensions qui imposerait de changer manuellement les valeurs dans tout le programme, ce qui est assez fastidieux. Nous allons, pour anticiper ce problème, définir deux constantes symboliques dans notre bibliothèque tools.h :

```
#define          WINDOW_WIDTH            1024
#define          WINDOW_HEIGHT           768
```

Dans le programme principal, modifier la création de la fenêtre App de façon à ce que la déclaration utilise dorénavant les constantes symboliques.

Propriétés des joueurs

Avant de gérer les déplacements de notre sprite, nous allons définir les propriétés de chaque joueur conformément à la liste suivante:

- la position selon l'axe des abscisses (x),
- la position selon l'axe des ordonnées (y),
- l'orientation par rapport à l'axe des abscisses (alpha),
- la vitesse de déplacement (speed).

Créez une bibliothèque player dans votre projet (inspirez vous de la création de la bibliothèque tools du TP précédent). Dans le fichier player.h, déclarez une structure PlayerProperties contenant les champs ci-dessus selon le modèle suivant :

```
typedef struct
{
    double x,y;
    double alpha;
    double speed;
}
PlayerProperties;
```

Au début du programme principal, complétez la création du sprite avec les propriétés du joueur.

```
// Création du joueur 1
Sprite Player1("images/player1.png");
PlayerProperties PropPlayer1={ 100. , WINDOW_HEIGHT/2. , 0.78 , 0.};
```

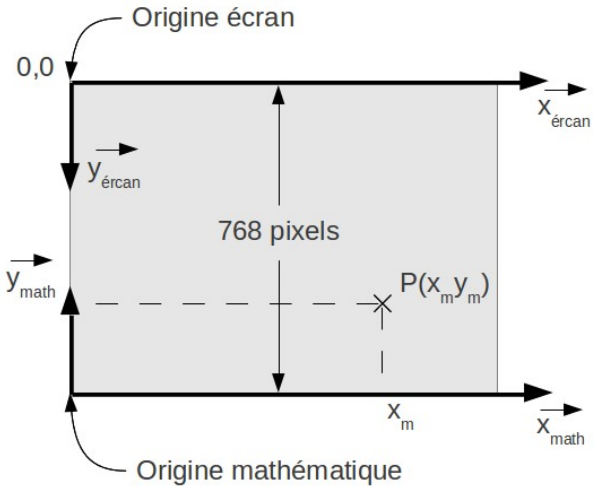
Modifier l'affichage du joueur 1 afin que celui-ci se fasse conformément à ses propriétés (fonctions SetPosition et SetRotation). Les champs d'une structure sont accessibles en stipulant le nom de la structure suivi du nom du champ séparé par un point. Par exemple, pour accéder à l'angle du joueur 1 il faut utiliser la syntaxe suivante:

```
PropPlayer1.alpha
```

Tester votre programme en modifiant la taille de la fenêtre, la position et l'angle initial de votre joueur.

Affichage cartésien

Vous aurez probablement remarqué que l'axe des ordonnées est orienté du haut vers le bas. Cette convention est historique. Elle est liée à l'utilisation des écrans à tube cathodique qui affichaient les pixels en commençant par le coin supérieur gauche. Malgré l'évolution technologique, cette convention a été conservée. Elle n'est pas toujours pratique, en particulier lorsque l'on fait des opérations mathématiques dans un repère cartésien. Considérons le schéma suivant :



Soit le point P de coordonnées (x_m,y_m) exprimées dans le repère mathématique. Déterminer les coordonnées de ce même point dans le repère écran et en déduire la transformation qui permet de passer d'un repère à l'autre. Dans la suite des TP, tous les calculs seront faits dans le repère mathématique et le changement de repère aura lieu au moment où les sprites seront positionnés. Dans la structure `PlayerProperties` les coordonnées x et y seront exprimées dans le repère mathématique. Modifier le positionnement du joueur 1 (`Player1.SetPosition...`) de façon à prendre en compte le changement de repère. Vérifier que, lorsque le joueur 1 est initialisé en (100,100), ce dernier s'affiche bien en bas à gauche de la fenêtre. Pensez à utiliser les constantes symboliques précédemment déclarées!

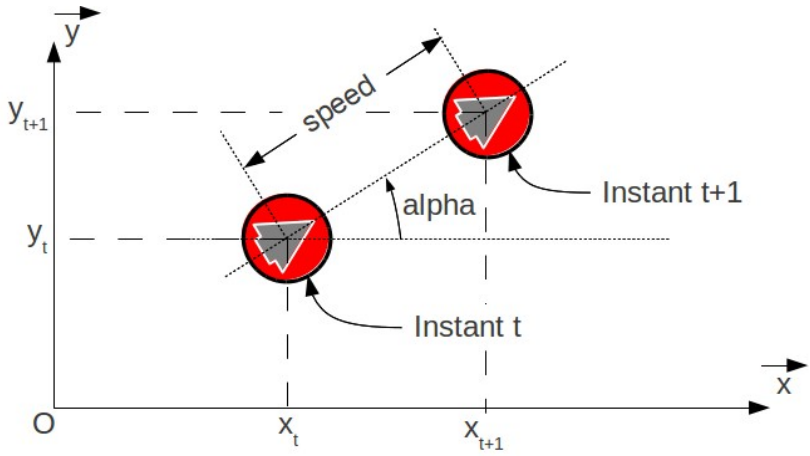
Déplacement des joueurs

La dynamique de notre joueur – inspiré des êtres humains – va avoir certaines propriétés. Par exemple, le joueur ne pourra pas courir sur le côté ni en arrière. Pour calculer les déplacements, nous allons écrire un modèle mathématique qui consiste à mettre à jour la nouvelle position (x,y) du joueur en fonction de son ancienne position (x,y), de sa vitesse instantanée (speed) et de son orientation (alpha). Créez une fonction `ComputeDynamic` dans la librairie `player`. Le prototype de la fonction vous est donné ci-dessous:

```
PlayerProperties ComputeDynamic(PlayerProperties prop);
```

Notez que tous les éléments nécessaires au calcul se trouvent dans la structure `PlayerProperties`. Le paramètre d'entrée correspond aux propriétés du joueur avant traitement et la fonction retourne en

sortie les propriétés après traitement. Le schéma ci-dessous montre l'évolution de notre joueur entre deux instants (t et t+1) séparés de 10 ms, notre période d'échantillonnage. La distance parcourue, exprimée en nombre de pixels par période, est donnée par la variable speed [pixels / 10ms]. On rappelle que l'angle alpha est donné en radians.



Calculer x_{t+1} et y_{t+1} en fonction de speed, alpha, x_t et y_t. Vérifiez votre résultat avec l'application numérique suivante (vous devez trouver x_{t+1}=4,250 et y_{t+1}= 4,165) :

- speed=2,5
- alpha=π/3
- x_t=3
- y_t=2

Lorsque le calcul est validé, complétez l'implémentation de la fonction ComputeDynamic. Insérez un appel à la fonction avant l'affichage du sprite :

```
// Actualise le joueur 1
PropPlayer1=ComputeDynamic(PropPlayer1);
Player1.SetPosition(PropPlayer1.x,WINDOW_HEIGHT-PropPlayer1.y);
Player1.SetRotation(Radians_To_Degrees(PropPlayer1.alpha));
App.Draw(Player1);
```

Testez votre implémentation avec l'initialisation suivante :

```
PlayerProperties PropPlayer1={ 100. , 100. , 0.61 , 1.};
```

Si tout fonctionne correctement, votre joueur doit traverser le terrain en diagonale du coin inférieur gauche au coin supérieur droit en approximativement 10 secondes.

Au début de la fonction ComputeDynamic multipliez la vitesse par 0,99. Cela permet de simuler le ralentissement naturel du joueur. Vérifiez que la vitesse du joueur décroit.

Commentez, indentez et faites valider par l'enseignant.