Sujet finale Prologin : **Les Pulsars**

Equipe Soft Prologin

Table des matières

1 Introduction

Nous sommes en l'année 5142, sur la planète *Xilihp*, la guerre fait rage, les différentes nations se battent 36 heures sur 36.

1.1 La planète

Cette planète est rectangulaire, enfin le seul continent habitable, est rectangulaire.

Leurs technologies évoluaient à une vitesse fulgurante, jusqu'au jour où ils découvrent la technologie Anakronox. Cette technologie leur permet de créer des tanks surpuissants.

1.2 Les Anakronox

Les Anakronox sont des tanks surpuissants, émettant des radiations sur une certaine zone, suivant une distance et un angle définissant un arc de cercle.

Ces radiations sont si puissantes que seuls les Anakronox en sont protégés, aucune espèce vivante ne peut rester à la surface de la planète.

Très vite la guerre se transforma en guerre à distance, où les commandants sont enfouis dans des bunkers souterrains.

Cependant un problème se pose puisque le contrôle des Anakronox se fait par fréquences radio, et celles-ci peuvent être modifiées.

Pour cela il fut inventé des robots spécialisés dans la configuration des Anakronox.

1.3 Les R4D2

Les R4D2 sont des robots mobiles permettant de modifier la configuration de communication des machines modernes (Anakronox, R4D2, ...).

Ces robots sont plutôt résistants, mais lorsque ils sont irradiés par des Anakronox ennemis, leur vitesse de déplacement diminue de plus en plus à cause du champ de force qui les repousse. Si le champ est trop intense, dans un premier temps ils vont rester bloqués, et si le champ de force ennemi est ultra-puissant, les R4D2 explosent.

Comme les R4D2 sont contrôlés par fréquence radio, les commandants se sont aperçus qu'ils pouvaient prendre le contrôle des R4D2 ennemis grâce à leurs propres R4D2.

1.4 Pour la liberté !

Vous êtes le commandant de l'armée de votre nation, et vous devez permettre à votre peuple de recommencer une vie à la surface.

Pour cela, vous devez contrôler une zone, la plus grande possible, pour que votre peuple puisse sortir en toute sécurité. Le vainqueur sera donc la nation qui contrôlera le plus d'Anakronox.

2 Présentation de l'interface de communication

Pour communiquer avec vos unités, vous disposez d'une interface de communication.

2.1 Les satellites radars

Les satellites radars vous permettent de rechercher des unités amies ou ennemies sur la planète.

Ces satellites ont été envoyés avant la découverte des Anakronox, les perturbations crées par les radiations bloquent la vision des radars. Les chercheurs ont fini par trouver une solution : régler les radars sur la fréquence des radiations propres à votre nation.

Problème, si un ennemi irradie une zone, votre visibilité sera réduite.

2.2 Les Anakronox

Vous disposez de commandes de contrôle pour diriger vos Anakronox.

Vous pouvez :
- Déplacer
- Pulser : irradier une zone
- Linker : transmettre l'énergie d'un Anakronox à un autre

Bien sûr à un instant t, un Anakronox ne peut effectuer qu'une de ces actions à la fois.

2.3 Les R4D2

Vous disposez de commandes de contrôles pour diriger vos R4D2.

Vous pouvez :
- Déplacer
- Prendre le contrôle d'un Anakronox
- Prendre le contrôle d'un R4D2

Bien sûr à un instant $\mathbf{t}$, un R4D2 ne peut effectuer qu'une seule de ces actions à la fois.

3 Coding or not coding, that is the question ?

La salle de contrôle est dans une zone irradiée, donc vous ne pourrez accédez à cette salle que trente-six heures.

Vous devrez faire un programme assurant le contrôle automatique de vos unités, et vous faire decontaminer par la suite en vous faisant copieusement asperger d'un antidote connu sous le nom de monoxyde de dihydrogène.

Le système d'exploitation sur cet ordinateur vous fournit une API (Application Program Interface) qui permet cette gestion.

3.1 API Système

L'API de l'ordinateur contient une API système, qui assure la gestion système de votre programme.

La gestion des identifiants par le système est faite de la façon suivante :

- < 0 : toutes les nations sauf la nation -ID (ex : ID=-5 pour toutes les nations sauf la nation 5)
- 0 : nation sauvage (neutre)
- > 0 : nation ayant pour identifiant ID

La nation sauvage désigne des Anakronox abandonnés sur le champ de bataille lors de la dernière guerre. Ces unités ne font rien, et peuvent être capturées sans résistance.

Le système appellera votre programme régulièrement. Vous lui précisez les actions à faire exécuter, par l'intermédiaire de 4 fonctions que vous devrez écrire :

- `player_init` reçoit un entier correspondant à votre identifiant de nation, et est appelé à l'initialisation de votre programme :

  - C/C++ : void player_init(int team_id)
  - CPP : void Player : :Init(int team_id)
  - PAS : procedure player_init(team_id : integer)
  - CAML : let player_init n =
  - JAVA : void Prolo.Init(int team_id)

- `player_new_turn` reçoit un entier correspondant au numéro du tour en cour, et est appelé au début de chaque tour(sauf le tour 0 → init) :

  - C/C++ : void player_new_turn(int turn_number)
  - CPP : void Player : :NewTurn(int turn_number)
  - PAS : procedure player_new_turn(turn_number : integer)

- CAML : let player_new_turn n =
- JAVA : void Prolo.NewTurn(int turn_number)

- `player_akx_turn` reçoit un entier correspondant au numéro de l'Anakronox en cours, et est appelé au début de chaque tour pour chacun de vos Anakronox (sauf le tour 0 → init) :

  - C/C++ : void player_akx_turn(int akx_id)
  - CPP : void Player : :AkxTurn(int akx_id)
  - PAS : procedure player_akx_turn(akx_id : integer)
  - CAML : let player_akx_turn n =
  - JAVA : void Prolo.AkxTurn(int akx_id)

- `player_r4d2_turn` reçoit un entier correspondant au numéro du r4d2 en cours, et est appelé au debut de chaque tour pour chacun de vos r4d2 (sauf le tour 0 → init) :

  - C/C++ : void player_r4d2_turn(int akx_id)
  - CPP : void Player : :R4d2Turn(int akx_id)
  - PAS : procedure player_r4d2_turn(akx_id : integer)
  - CAML : let player_r4d2_turn n =
  - JAVA : void Prolo.R4d2Turn(int akx_id)

Le système décompose le temps sous forme de tours, et vous permet de connaître le numéro du tour en cours. De plus il vous permet de connaître le numéro du tour ou sera décrêté le cessez-le-feu :

- `turn_number` renvoie le nombre de tour total, du début jusqu'au cessez-le-feu :

  - C/C++ : int turn_number()
  - CPP : int General : :TurnNumber()
  - PAS : function turn_number : integer
  - CAML : turn_number : int → int
  - JAVA : int Prolo.jTurnNumber()

- `turn_counter` renvoie le numéro du tour en cours :

  - C/C++ : int turn_counter()
  - CPP : int General : :TurnCounter()
  - PAS : function turn_counter : integer
  - CAML : turn_counter : int → int
  - JAVA : int Prolo.jTurnCounter()

L'intervalle entre deux tours est constant, donc votre programme doit finir l'exécution

du tour avant la fin de ce temps. Si vous dépassez le délai autorisé, votre programme sera interrompu puis réinitialisé au tour suivant.

Vous disposez d'une fonction vous donnant le temps restant (en millisecondes) avant la fin du tour.

- `time_get_left` renvoie le temps en millisecondes restant avant la fin du tour :

  - C/C++ : int time_get_left()
  - CPP : int General : :TimeGetLeft()
  - PAS : function time_get_left : integer
  - CAML : time_get_left : int → int
  - JAVA : int Prolo.jTimeGetLeft()

Pour connaître une estimation de la puissance de votre nation, le système vous fournit une fonction :

- `score_get` renvoie le score de votre nation. Le score est calculé en fonction du nombre d'Anakronox et de R4D2 que vous possédez.

  - C/C++ : int score_get()
  - CPP : int General : :ScoreGet()
  - PAS : function score_get : integer
  - CAML : score_get : int → int
  - JAVA : int Prolo.jScoreGet()

Vos appels à l'API peuvent provoquer des erreurs, pour cela il existe une fonction vous permettant de connaître la dernière erreur qui s'est produite :

- `error_get` renvoie la dernière erreur de votre programme :

  - C/C++ : int error_get()
  - CPP : int General : :ErrorGet()
  - PAS : function error_get : integer
  - CAML : error_get : int → int
  - JAVA : int Prolo.jErrorGet()

Une fonction de l'API ayant détecté une erreur renvoie -1 (ou -1.0 en flottant), l'erreur renvoyée par cette fonction correspond à une de ces valeurs :

0 : pas d'erreur

1 : l'unité voulue n'est pas à vous

2 : l'unité voulue n'est pas visible

3 : l'identifiant ne correspond pas

4 : la position n'est pas valide

5 : erreur dans le type d'unité

6 : cible invalide

7 : unité détruite

3.2 API R4D2

Le système affecte des identifiants aux unités (R4D2 et Anakronox) qu'il utilise dans l'API pour définir les cibles des appels de fonction.

Le système vous fournit une API permettant la gestion des R4D2, vous permettant d'obtenir des informations ou de faire exécuter des actions aux R4D2 :

- `r4d2_get_team` renvoie l'identifiant de la nation du r4d2 passé en paramètre :
  - C/C++ : int r4d2_get_team(int r4d2_id)
  - CPP : int R4D2 : :GetTeam(int r4d2_id)
  - PAS : function r4d2_get_team(r4d2_id : integer) : integer
  - CAML : r4d2_get_team : int → int
  - JAVA : int Prolo.jR4d2GetTeam(int r4d2_id)

- `r4d2_get_pos_x` renvoie la position sur l'axe horizontal du r4d2 demandé :
  - C/C++ : float r4d2_get_pos_x(int r4d2_id)
  - CPP : float R4D2 : :GetPosX(int r4d2_id)
  - PAS : function r4d2_get_pos_x(r4d2_id : integer) : single
  - CAML : r4d2_get_pos_x : int → double
  - JAVA : int Prolo.jR4d2GetPosX(int r4d2_id)

- `r4d2_get_pos_y` renvoie la position sur l'axe verticale du r4d2 demandé :
  - C/C++ : float r4d2_get_pos_y(int r4d2_id)
  - CPP : float R4D2 : :GetPosY(int r4d2_id)
  - PAS : function r4d2_get_pos_y(r4d2_id : integer) : single
  - CAML : r4d2_get_pos_y : int → double
  - JAVA : int Prolo.jR4d2GetPosY(int r4d2_id)

- `r4d2_get_status` renvoie l'etat du r4d2 demande :
  - C/C++ : int r4d2_get_status(int r4d2_id)
  - CPP : int R4D2 : :GetStatus(int r4d2_id)
  - PAS : function r4d2_get_status(r4d2_id : integer) : integer

- CAML : r4d2_get_status : int → int
- JAVA : int Prolo.jR4d2GetStatus(int r4d2_id)

Les états du r4d2 sont donnés de la façon suivante :

0 : déplacement ou inactif

1 : capture d'un r4d2

2 : capture d'un Anakronox

- `r4d2_get_speed` renvoie la vitesse de base des r4d2 :

  - C/C++ : float r4d2_get_speed()
  - CPP : float R4D2 : :GetSpeed()
  - PAS : function r4d2_get_speed : single
  - CAML : r4d2_get_speed : int → double
  - JAVA : float Prolo.jR4d2GetSpeed()

- `r4d2_get_destroy_speed` renvoie la force opposée au mouvement des R4D2 en dessous de laquelle l'unité R4D2 explose. Cette force est calculée de la manière que la vitesse (voir 4.1, 15)

  - C/C++ : float r4d2_get_destroy_speed()
  - CPP : float R4D2 : :GetDestroySpeed()
  - PAS : function r4d2_get_destroy_speed : single
  - CAML : r4d2_get_destroy_speed : int → double
  - JAVA : float Prolo.jR4d2GetDestroySpeed()

  En général, le résultat retourné est une valeur négative.

- `r4d2_turn_take_r4d2` renvoie le nombre de tours nécessaires pour capturer un r4d2 sauvage :

  - C/C++ : int r4d2_turn_take_r4d2()
  - CPP : int R4D2 : :TurnTakeR4d2()
  - PAS : function r4d2_turn_take_r4d2 : integer
  - CAML : r4d2_turn_take_r4d2 : int → int
  - JAVA : int Prolo.jR4d2TurnTakeR4d2()

- `r4d2_turn_untake_r4d2` renvoie le nombre de tours supplémentaires si le r4d2 à capturer appartient à l'ennemi :

  - C/C++ : int r4d2_turn_untake_r4d2()
  - CPP : int R4D2 : :TurnUntakeR4d2()

- PAS : function r4d2_turn_untake_r4d2 : integer
- CAML : r4d2_turn_untake_r4d2 : int → int
- JAVA : int Prolo.jR4d2TurnUntakeR4d2()

- `r4d2_turn_take_akx` renvoie le nombre de tours nécessaires pour capturer un Anakronox sauvage :

  - C/C++ : int r4d2_turn_take_akx()
  - CPP : int R4D2 : :TurnTakeAkx()
  - PAS : function r4d2_turn_take_akx : integer
  - CAML : r4d2_turn_take_akx : int → int
  - JAVA : int Prolo.jR4d2TurnTakeAkx()

- `r4d2_turn_untake_akx` renvoie le nombre de tours supplémentaires si l'Anakronox à capturer appartient à l'ennemi :

  - C/C++ : int r4d2_turn_untake_akx()
  - CPP : int R4D2 : :TurnUntakeAkx()
  - PAS : function r4d2_turn_untake_akx : integer
  - CAML : r4d2_turn_untake_akx : int → int
  - JAVA : int Prolo.jR4d2TurnUntakeAkx()

- `r4d2_move` donne un ordre de déplacement, vers une destination, au r4d2 voulu :

  - C/C++ : int r4d2_move(int r4d2_id, float destx, float desty)
  - CPP : int R4D2 : :Move(int r4d2_id, float destx, float desty)
  - PAS : function r4d2_move(r4d2_id : integer ; destx, desty : single) : integer
  - CAML : r4d2_move : int → double → double → int
  - JAVA : int Prolo.jR4d2Move(int r4d2_id, float destx, float desty)

  Une fois que l'on a donné à un R4D2 l'ordre de se déplacer, il continue d'aller vers sa destination aux tours suivants sauf ordre contraire.

- `r4d2_take_r4d2` donne un ordre de capture d'un r4d2 :

  - C/C++ : int r4d2_take_r4d2(int r4d2_id, int target_id)
  - CPP : int R4D2 : :TakeR4d2(int r4d2_id, int target_id)
  - PAS : function r4d2_take_r4d2(r4d2_id, target_id : integer) : integer
  - CAML : r4d2_take_r4d2 : int → int → int
  - JAVA : int Prolo.jR4d2TakeR4d2(int r4d2_id, int target_id)

  La capture est possible seulement si la distance entre les R4D2 est inférieure à 1.

- `r4d2_take_akx` donne un ordre de capture d'un Anakronox :

  - C/C++ : int r4d2_take_akx(int r4d2_id, int target_id)
  - CPP : int R4D2 : :TakeAkx(int akx_id, int target_id)
  - PAS : function r4d2_take_akx(akx_id, target_id : integer) : integer
  - CAML : r4d2_take_akx : int → int → int
  - JAVA : int Prolo.jR4d2TakeAkx(int r4d2_id, int target_id)

  La capture est possible seulement si la distance entre le R4D2 et l'Anakronox est inférieure à 1.

3.3 API Anakronox

Le système vous fournit une API permettant la gestion des Anakronox, vous permettant d'obtenir des informations ou de faire exécuter des actions aux Anakronox :

- `akx_get_team` renvoie l'identifiant de nation de l'Anakronox passé en paramètre :

  - C/C++ : int akx_get_team(int akx_id)
  - CPP : int Akx : :GetTeam(int akx_id)
  - PAS : function akx_get_team(akx_id : integer) : integer
  - CAML : akx_get_team : int → int
  - JAVA : int Prolo.jAkxGetTeam(int akx_id)

- `akx_get_pos_x` renvoie la position sur l'axe horizontal de l'Anakronox demandé :

  - C/C++ : float akx_get_pos_x(int akx_id)
  - CPP : float Akx : :GetPosX(int akx_id)
  - PAS : function akx_get_pos_x(akx_id : integer) : single
  - CAML : akx_get_pos_x : int → double
  - JAVA : int Prolo.jAkxGetPosX(int akx_id)

- `akx_get_pos_y` renvoie la position sur l'axe vertical de l'Anakronox demandé :

  - C/C++ : float akx_get_pos_y(int akx_id)
  - CPP : float Akx : :GetPosY(int akx_id)
  - PAS : function akx_get_pos_y(akx_id : integer) : single
  - CAML : akx_get_pos_y : int → double
  - JAVA : int Prolo.jAkxGetPosY(int akx_id)

- `akx_get_status` renvoie l'état de l'Anakronox demandé :

  - C/C++ : int akx_get_status(int akx_id)
  - CPP : int Akx : :GetStatus(int akx_id)

- PAS : function akx_get_status(akx_id : integer) : integer
- CAML : akx_get_status : int → int
- JAVA : int Prolo.jAkxGetStatus(int akx_id)

Les états de l'Anakronox sont donnés de la façon suivante :

0 : déplacement ou inactif

1 : pulse sur une zone

2 : link vers un autre Anakronox

- `akx_pulse_angle` renvoie l'angle d'ouverture de l'Anakronox demandé :

  - C/C++ : float akx_pulse_angle(int akx_id)
  - CPP : float Akx : :PulseAngle(int akx_id)
  - PAS : function akx_pulse_angle(akx_id : integer) : single
  - CAML : akx_pulse_angle : int → double
  - JAVA : int Prolo.jAkxPulseAngle(int akx_id)

- `akx_pulse_destx` renvoie la destination X de tir de l'Anakronox demandé :

  - C/C++ : float akx_pulse_destx(int akx_id)
  - CPP : float Akx : :PulseDestX(int akx_id)
  - PAS : function akx_pulse_destx(akx_id : integer) : single
  - CAML : akx_pulse_destx : int → double
  - JAVA : int Prolo.jAkxPulseDestX(int akx_id)

- `akx_pulse_desty` renvoie la destination Y de tir de l'Anakronox demandé :

  - C/C++ : float akx_pulse_desty(int akx_id)
  - CPP : float Akx : :PulseDestY(int akx_id)
  - PAS : function akx_pulse_desty(akx_id : integer) : single
  - CAML : akx_pulse_desty : int → double
  - JAVA : int Prolo.jAkxPulseDestY(int akx_id)

- `akx_get_speed` renvoie la vitesse de base des Anakronox :

  - C/C++ : float akx_get_speed()
  - CPP : float Akx : :GetSpeed()
  - PAS : function akx_get_speed : single
  - CAML : akx_get_speed : int → double
  - JAVA : float Prolo.jAkxGetSpeed()

- `akx_get_power` renvoie la puissance de base des Anakronox :

- C/C++ : float akx_get_power()
- CPP : float Akx : :GetPower()
- PAS : function akx_get_power : single
- CAML : akx_get_power : int → double
- JAVA : float Prolo.jAkxGetPower()

C'est la quantité d'énergie minimum que les Anakronox contiennent au début d'un tour.

- `akx_pulse_coef` renvoie le coefficient d'affectation de vitesse de l'intensité de pulse :

  - C/C++ : int akx_pulse_coef()
  - CPP : int Akx : :PulseCoef()
  - PAS : function akx_pulse_coef : integer
  - CAML : akx_pulse_coef : int → int
  - JAVA : int Prolo.jAkxPulseCoef()

- `akx_see_power` renvoie la puissance nécessaire pour que vos satellites puissent voir une zone :

  - C/C++ : int akx_see_power()
  - CPP : int Akx : :SePower()
  - PAS : function akx_see_power : integer
  - CAML : akx_see_power : int → int
  - JAVA : int Prolo.jAkxSePower()

- `akx_move` donne un ordre de déplacement, vers une destination, à l'Anakronox voulu :

  - C/C++ : int akx_move(int akx_id, float destx, float desty)
  - CPP : int Akx : :Move(int akx_id, float destx, float desty)
  - PAS : function akx_move(akx_id : integer ; destx, desty : single) : integer
  - CAML : akx_move : int → double → double → int
  - JAVA : int Prolo.jAkxMove(int akx_id, float destx, float desty)

  Une fois que l'on a donné à un Anakronox l'ordre de se déplacer, il continue d'aller vers sa destination aux tours suivants sauf ordre contraire.

- `akx_pulse` donne un ordre de pulse à un Anakronox :

  - C/C++ : int akx_pulse(int akx_id, float destx, float desty, float angle)
  - CPP : int Akx : :TakeAkx(int akx_id, float destx, float desty, float angle)

– PAS : function akx_take_akx(akx_id : integer ;destx, desty, angle : single) : integer
– CAML : akx_take_akx : int → double→ double→ double → int
– JAVA : int Prolo.jAkxTakeAkx(int akx_id, float destx, float desty, float angle)

Les pulsions d'un Anakronox irradient un arc de cercle avec pour centre l'Anakronox, pour rayon la distance entre l'Anakronox et la destination et pour angle d'ouverture l'angle spécifié lors de l'appel.

- `akx_link` donne un ordre de transfert d'énergie d'un Anakronox vers en autre Anakronox :

  – C/C++ : int akx_link(int akx_id, int target_id)
  – CPP : int Akx : :Link(int akx_id, int target_id)
  – PAS : function akx_link(akx_id, target_id : integer) : integer
  – CAML : akx_link : int → int → int
  – JAVA : int Prolo.jAkxLink(int akx_id, int target_id)

  Après le transfert, l'Anakronox cible va emmagasiner l'énergie supplémentaire jusqu'à la prochaine commande `pulse`

3.4 API Satellite

Le système utilise une carte symbolique continue pour représenter la planète en interne, c.à.d. qu'une position n'est pas un couple d'entiers mais un couple de nombres à virgule (ex : $(1.2563566; 25.2135135)$) représentant la latitude(X) et la longitude(Y) de ce point.

Le système vous fournit une API permettant la gestion des satellites, vous permettant d'obtenir des informations sur la carte du monde (unités, états, ...) :

Les renseignements renvoyés par les fonctions de l'API satellite dépendent du champ de vision de vos Anakronox sauf : `map_get_size_x, map_get_size_y, map_count_akx, map_count_r4d2, map_count_my_akx et map_count_my_r4d2`.

- `map_get_size_x` renvoie la taille en X de la carte :

  – C/C++ : float map_get_size_x()
  – CPP : float Map : :GetSizeX()
  – PAS : function map_get_size_x : single
  – CAML : map_get_size_x : int → double
  – JAVA : float Prolo.jMapGetSizeX()

- `map_get_size_y` renvoie la taille en Y de la carte :

  - C/C++ : float map_get_size_y()
  - CPP : float Map : :GetSizeY()
  - PAS : function map_get_size_y : single
  - CAML : map_get_size_y : int → double
  - JAVA : float Prolo.jMapGetSizeY()

- `map_get_pulse` renvoie l'intensité des radiations pour un point et une équipe (ou groupe d'équipes) donnés :

  - C/C++ : float map_get_pulse(int team_id, float x, float y)
  - CPP : float Map : :GetPulse(int team_id, float x, float y)
  - PAS : function map_get_pulse(team_id : integer ;x, y : single) : single
  - CAML : map_get_pulse : int → double → double → double
  - JAVA : float Prolo.jMapGetPulse(int team_id, float x, float y)

- `map_get_pulse_id` renvoie l'intensité des radiations pour un point et un Anakronox donnés :

  - C/C++ : float map_get_pulse_id(int akx_id, float x, float y)
  - CPP : float Map : :GetPulseId(int akx_id, float x, float y)
  - PAS : function map_get_pulse_id(akx_id : integer ;x, y : single) : single
  - CAML : map_get_pulse_id : int → double → double → double
  - JAVA : float Prolo.jMapGetPulseId(int akx_id, float x, float y)

- `map_count_akx` renvoie le nombre total d'Anakronox sur la carte (y compris les Anakronox hors de votre champ de vision !) :

  - C/C++ : int map_count_akx()
  - CPP : int Map : :CountAkx()
  - PAS : function map_count_akx : integer
  - CAML : map_count_akx : int → int
  - JAVA : int Prolo.jMapCountAkx()

- `map_count_r4d2` renvoie le nombre total de R4D2 sur la carte (y compris les r4d2 hors de votre champ de vision !) :

  - C/C++ : int map_count_r4d2()
  - CPP : int Map : :CountR4d2()
  - PAS : function map_count_r4d2 : integer
  - CAML : map_count_r4d2 : int → int
  - JAVA : int Prolo.jMapCountR4d2()

- `map_count_my_akx` renvoie le nombre d'Anakronox de votre nation sur la carte :

  - C/C++ : int map_count_my_akx()
  - CPP : int Map : :CountMyAkx()
  - PAS : function map_count_my_akx : integer
  - CAML : map_count_my_akx : int → int
  - JAVA : int Prolo.jMapCountMyAkx()

- `map_count_my_r4d2` renvoie le nombre de R4D2 de votre nation :

  - C/C++ : int map_count_my_r4d2()
  - CPP : int Map : :CountMyR4d2()
  - PAS : function map_count_my_r4d2 : integer
  - CAML : map_count_my_r4d2 : int → int
  - JAVA : int Prolo.jMapCountMyR4d2()

- `map_get_nearest_akx_plot` renvoie l'Anakronox, de la nation spécifiée, le plus proche du point voulu :

  - C/C++ : int map_get_nearest_akx_plot(float x, float y, int team_id)
  - CPP : int Map : :GetNearestAkxPlot(float destx, float desty, int team_id)
  - PAS : function map_get_nearest_akx_plot(destx, desty : single ; team_id : integer) : integer
  - CAML : map_get_nearest_nearest_akx_plot : double → double → int → int
  - JAVA : int Prolo.jMapGetNearestAkxPlot(float destx, float desty, int team_id)

- `map_get_nearest_r4d2_plot` renvoie le R4D2, de la nation spécifiée, le plus proche du point voulu :

  - C/C++ : int map_get_nearest_r4d2_plot(float x, float y, int team_id)
  - CPP : int Map : :GetNearestR4d2Plot(float destx, float desty, int team_id)
  - PAS : function map_get_nearest_r4d2_plot(destx, desty : single ; team_id : integer) : integer
  - CAML : map_get_nearest_nearest_r4d2_plot : double → double → int → int
  - JAVA : int Prolo.jMapGetNearestR4d2Plot(float destx, float desty, int team_id)

- `map_get_nearest_akx` renvoie l'Anakronox, de la nation spécifiée, le plus proche de l'unité voulue :

  - C/C++ : int map_get_nearest_akx(int id, int team_id)
  - CPP : int Map : :GetNearestAkx(int id, int team_id)
  - PAS : function map_get_nearest_akx(id, team_id : integer) : integer

- CAML : map_get_nearest_nearest_akx : int → int → int
- JAVA : int Prolo.jMapGetNearestAkxPlot(int id, int team_id)

- `map_get_nearest_r4d2` renvoie le R4D2, de la nation spécifiée, le plus proche de l'unité voulue :

  - C/C++ : int map_get_nearest_r4d2(int id, int team_id)
  - CPP : int Map : :GetNearestR4d2(int id, int team_id)
  - PAS : function map_get_nearest_r4d2(id, team_id : integer) : integer
  - CAML : map_get_nearest_nearest_r4d2 : int → int → int
  - JAVA : int Prolo.jMapGetNearestR4d2(int id, int team_id)

4 Remarques

4.1 La gestion du monde

- Comme la carte est continue et les unités ponctuelles, il ne peut pas y avoir de collision.

- Comme la carte est continue, il peut y avoir beaucoup d'unités même dans une zone qui va de (0, 0) à (1, 1).

- L'évaluation du score est calculée de la manière suivante :

$$Score = Nombre\, d'Anakronox\, possédés + \frac{Nombre\, de\, R4D2\, possédés}{Nombre\, de\, R4D2\, total}$$

- L'intensité de l'irradiation des pulsions suit la formule :

$$Irradiation = \frac{Puissance\, Pulsar}{1 + Aire\, Zone\, Irradiée}$$

- Le calcul de la puissance sur une zone (ex : pour la vision des satellites) suit la formule :

$$Puissance = Puissance\, de\, vos\, Anakronox - Puissance\, des\, ennemis$$

- Le transfert d'énergie entre Anakronox suit la formule :

$$Energie\, recue = \frac{Puissance\, Anakronox\, source}{1 + Distance\, entre\, les\, Anakronox}$$

  Après le transfert, l'Anakronox cible va emmagasiner l'énergie supplémentaire jusqu'à la prochaine commande `pulse`

- Le calcul de la vitesse des R4D2 suit la formule :

$$Vitesse = Vitesse\, de\, base + Coefficient * Intensité$$

- Lorsque vos R4D2 sont irradiés par des Anakronox ennemis, dans un premier temps ils s'immobilisent, et si le champ de force ennemi est ultra-puissant, vos R4D2 vont exploser.

- L'identifiant d'une unité correspond à son index dans un unique tableau du serveur(les Anakronox et les R4D2 sont mémorisés dans le même tableau).

- Pour capturer une unité, vous devez exécuter l'action de capture correspondante pendant X tours, où X est le nombre de tours nécessaires pour capturer cette unité (ex : pour capture un Anakronox d'une nation ennemie)

  $$X = \text{r4d2\_turn\_take\_akx}() + \text{r4d2\_turn\_untake\_akx}()$$

  Au bout de X tours, l'unité sera en votre possession.

- Si plusieurs nations tentent de capturer la même unité, c'est la nation ayant le plus de R4D2 en mode capture sur cette unité qui gagne.

  En cas d'égalité la capture est annulée.

- Une unité ne peut exécuter qu'une action par tour. Si vous affectez plusieurs actions à une unité pendant le même tour, seule la dernière action sera prise en compte.

- L'ordre d'exécution des actions est le suivant :

  1. Destruction des R4D2
  2. Transfert d'énergie entre Anakronox
  3. Pulse des Anakronox
  4. Déplacement des unités
  5. Capture des unités

- Si aucun ordre n'est donné à une unité pendant le tour, celle-ci exécute par défaut l'action du tour précédent.

4.2 Les langages exotiques

- Pour les candidats qui codent en CAML : certaines fonctions reçoivent un entier dans la version CAML et pas dans les autres versions. Dans ce cas l'entier est ignoré par l'API, donc vous pouvez mettre une valeur quelconque (par exemple 51, et sans eau s'il vous plaît).

- Pour les candidats qui codent en JAVA : l'équipe serveur s'est vraiment compliquée la vie à cause d'eux. Il y aura des représailles.

4.3 Votre code

- Les fichiers où vous devez coder sont :

  - C : prolo.c
  - C++/CPP : prolo.cc
  - PAS : prolo.pas
  - CAML : prolo.ml
  - JAVA : Prolo.java

- Vous pouvez coder dans d'autres fichiers bien sûr (programmation modulaire !). Il faut alors ajouter dans le fichier **Makefile** le nom de vos fichiers sources sur la ligne **SRC = ...**, exemple :

  pour ajouter un fichier ia.c dans la version c, modifiez la ligne **SRC = prolo.c** en **SRC = prolo.c ia.c**

- Dans la version CAML, il est impossible de rajouter des fichiers secondaires (enfin, nous n'y sommes pas arrivés, ce qui à vrai dire n'est pas tout-à-fait pareil).

- Si vous ajoutez des fichiers dans le Makefile, ne vous trompez pas d'extension ! sinon la compilation ne marchera pas !

5 Meta-client

Le méta-client permet de gérer les matchs, champions et cartes de manière simple et conviviale.

5.1 Mode d'emploi

5.1.1 Généralités

Démarrage Pour lancer le méta-client : trois possibilités. Soit tu le lances depuis un terminal (en tapant : `prologin`), soit tu double-clic sur l'icône Prologin, soit tu appuies sur `Ctrl-Shift-P`.

Principes Le méta-client se compose de quatre listes : une liste de personnes, une liste de cartes, un liste de champions et une liste de matchs. Chacune de ces listes peut être actualisée en appuyant sur le bouton correspondant. Attention, une ré-actualisation nécessite de faire des requêtes sur notre serveur SQL, évitez de le faire trop souvent.

Il est possible de trier les listes selon chacune des colonnes, en cliquant sur le titre.

5.1.2 La liste de personnes

Cette liste contient l'ensemble des organisateurs et des candidats ainsi que leur statut (présent, absent, zlocké). Il s'agit donc d'un mini-ICQ... Pour choisir un pseudo, allez dans le menu système. Il est aussi possible d'envoyer un message à une ou plusieurs personnes.

5.1.3 La liste des cartes

Cette liste contient aussi bien les cartes officielles que celles des bitmaps. Vous pouvez en rajouter en cliquant sur *ajouter*. La taille doit être au format "largeur x hauteur" (ex : "100x100"). Le flag "publique" indique que la carte doit être visible par les autres candidats. Attention : il faut aussi que les permissions Unix soient en accord.

Les cartes sont décrites dans un fichier texte, le format est décrit dans le man (`map`(5)).

5.1.4 La liste des champions

Cette liste est similaire à la liste des cartes, et contient tous les champions publics des autres candidats. Pour plus d'informations sur les scores, voir la section 5.2, page 18.

5.1.5 Lancement de matchs

Pour lancer un nouveau match, il faut sélectionner un ou plusieurs champions dans la liste des champions et une carte dans la liste des cartes, et appuyer sur `Ctrl-L` (ou menu Match, Lancer). L'option *paramètres* correspond aux paramètres supplémentaires à passer au serveur (voir la page man pour plus d'informations : `man server`).

Il est aussi possible de se connecter sur un match public lancé par un autre candidat. Le client graphique restera bloqué le temps qu'un tour complet se soit écoulé (oui, on fait un `read` bloquant...) ; il n'est pas planté.

5.2 Le classement temps réel

Ce classement est mis à jour à chaque fois qu'un organisateur lance un match. Deux types de score sont affichés : le premier représente la meilleure performance brute (le score afiché à la fin du match), et l'autre le classement moyen calculé comme suit :

$$\frac{score\ total}{nombre\ de\ matchs}$$

A chaque match, le champion reçoit un certain nombre de points dépendant de son classement (p) et du nombre de joueurs (n) :

$$\frac{100 \times n}{2^p}$$

Ainsi, si quatre champions se sont affrontés, le premier reçoit 400 points, le deuxième 200, le troisième 100 et le dernier 50.

Pourquoi deux scores différents ? Pour ne pas favoriser trop les performances sur des grandes maps bien remplies. En effet réussir à obtenir 5 points sur une petite carte peut être plus difficile que d'en obtenir 8 sur une grande carte.

5.3 Remarques

- Ne pas appuyer toutes les 4.2 secondes sur `Ctrl-A` ("tout actualiser"), ayez pitié du serveur MySQL.

- Ne pas lancer deux matchs simultanément (ceci est impossible pour des raisons techniques liées à la gestion de la mémoire partagée sous NetBSD).

- Le flag "public" pour les champions et les cartes ne s'occupe pas des permissions Unix sur les fichiers. Pensez à mettre aussi les permissions appropriées.

- Essayez autant que possible de copier vos champions publics dans un autre répertoire. Ceci vous permet de continuer à travailler dessus, tout en gardant une version publique fonctionnelle.

- Les erreurs SQL sont probablement au fait que les noms de champions et de cartes doivent être uniques ; il ne faut donc pas s'inquiéter.

- Il est formellement interdit de flooder un candidat de messages ou de tenter de hacker notre serveur... sous peine de disqualification immédiate.

Bonne chance !

Il ne te reste que 36 heures `:)`