ANNÉE 2020-2021

Projet Algorithme

Lucas Moniot

Rayane Kadi

Eileen Lorenzo

BDV Corp

IUT Dijon Dijon-Auxerre

ANNÉE 2020-2021

SOMMAIRE :

Table des matières

Projet Algorithme S1

Descriptif général du jeu :

Anno 1701 est un jeu de gestion et de stratégie en temps réel sorti sur PC en 2006, édité par Sunflowers développé par Related Designs et distribué par Deep Silver. Il se déroule durant la période de l'âge d'or de la marine à voile, au moment de la colonisation des Amériques. Le but du jeu est de prendre son indépendance vis-à-vis de son royaume.

Une version minimale

Partie à faire	Fait par
Menu initial (commencé une partie/ quitter une partie)	Lucas
Un écran de création de partie	Rayane
Une interface de présentation de l'île	Lucas
Un système de ressources	Rayane
Un système de bâtiments	Rayane
Un système de population	Lucas
Un système de vente	Lucas

Améliorations

Un écran de titre (0.5 points) : FAIT
Votre jeu devra comporter un écran de titre raisonnable (avec un logo...).

Le temps qui passe (0.5 points) : FAIT
Le numéro du tour en cours devra être visible la plupart du temps.

Aux armes (0.5 points) : FAIT
Le joueur doit pouvoir recruter des soldats pour protéger son "empire". Le recrutement des soldats doit nécessiter des ressources et de l'or. Ponctuellement, le joueur devra être attaqué (par des pirates par exemple). Le joueur devra alors se défendre au travers d'un système de combat nécessitant des choix de sa part ayant une influence sur le déroulement du combat.

Hisser les voiles (0.5 points) : FAIT

Le joueur doit pouvoir bâtir sur son île un chantier naval lui permettant de construire des bateaux. Ces bateaux devront avoir un nom et le joueur doit pouvoir accéder à un écran lui permettant de voir la liste des bateaux en sa possession.

We Know The Way (1 point): FAIT

(Nécessite l'amélioration : Hisser les voiles) Le joueur doit pouvoir fabriquer des bateaux d'exploration qu'il pourra envoyer explorer le vaste monde. Après quelques tours d'exploration, un navire pourra découvrir une nouvelle ile. Votre monde doit contenir au moins 6 iles différentes ayant chacune un nom différent. L'écran de gestion des navires doit permettre de savoir au joueur où en sont ses navires d'exploration.

Même le plus profond des sacs a un fond (0.5 point) : FAIT

Les capacités de stockage des ressources sur une île devront être limitées. Cette limite doit pouvoir être augmentée en construisant des entrepôts.

Hisser le pavillon noir (2 points) : FAIT

(Nécessite les améliorations : We Know The Way et Aux armes) Les navires d'explorations du joueur pourront être attaqués. Ces combats navals devront disposer de leur propre système de combat. Le joueur devra pouvoir améliorer ses navires pour les rendre plus résistants ou plus efficaces aux combats.

Let me be a noob (0.5 points): FAIT

Lors de la création de la partie, le joueur doit pouvoir choisir un niveau de difficulté influant sur sa situation de départ (ressources initiales, argent...)

Lutte des classes (1 point) : FAIT

Votre population devra être divisée en plusieurs (au moins deux) catégories de personnes : Colons, Citoyens, Aristocrates, Marchands. Chaque catégorie de personnes doit disposer de ses propres besoins et de son propre type de maison.

Seuls, c'est un peu triste... (3 points) : FAIT

Rendre le jeu jouable à plusieurs. Chaque joueur doit disposer de sa propre île de départ, de ses propres ressources... Les joueurs devront pouvoir interagir entre eux (commerce, combat...).

Partie Multijoueur (Explication par Eileen)

Evaluation de la solution la plus efficace

Pour relier et faire interagir deux programmes entre eux il fallait mettre en place un serveur web. Pour cela, la solution la plus simple était d'utiliser un serveur apache sous licence GNU (une autre solution aurait été d'héberger le serveur directement grâce à Lazarus, mais cela aurait été beaucoup plus long à mettre en place). Ensuite, il fallait choisir un protocole, j'ai choisi un protocole http pour pouvoir créer des pages PHP et télécharger des documents hébergés. Un protocole FTP aurait laissé moins de liberté dans l'échange de données. Il fallait ensuite mettre en place deux moyen de communication : un moyen pour envoyer des données sur le serveur et un autre pour les récupérer.

Télécharger des données depuis le serveur (unitHTTP)

Le moyen le plus rapide et simple à mettre en place était de tout simplement créer un système de requêtes pour télécharger un document en ligne. Pour cela, j'ai utilisé la librairie synapse (GNU) qui permet d'enregistrer le contenu d'un document web dans un fichier texte local.

```
Procedure TelechargerSave(IP,Partie, Joueur : string);                  //Enregistre le contenu d'un fichier de savegarde en sur le serveur vers un fichier local
var
  HTTPEnvoi : THTTPSend;
  HTTPGet : boolean;
  URL : String;

begin
    URL := Concat('http://'+IP+'/anno/'+Partie+'/'+Joueur+'/save.txt');
    HTTPEnvoi := THTTPSend.Create;                                      //adresse ULR de la sauvegarde
    HTTPGet := HTTPEnvoi.HTTPMethod('GET', URL);                        //Crée d'un processus d'envoi de requête HTTP
    HTTPEnvoi.Document.SaveToFile('saves/multi/'+Partie+'/'+Joueur+'/save.txt');//défini la requête de type GET, dirigé vers l'adresse URL de la sauvegarde
    HTTPEnvoi.Free;                                                     //Telecharge le ficher à cette adresse et l'enregistre dans un fichier local
end;                                                                    //termine le processus
```

Figure 1: procédure permettant de télécharger la sauvegarde d'un joueur depuis le serveur HTTP vers un fichier dans le dossier du programme.

Téléverser des données dans le serveur (unitHTTP)

C'était une question un peu plus complexe et dont la solution a demandé un léger contournement. Au départ je souhaitais pouvoir envoyer un fichier texte local vers le serveur, mais je n'ai pas réussi à trouver une solution à ce problème. Cependant, en prenant du recul, on s'aperçoit que la vraie problématique était d'envoyer du texte au serveur et qu'il était possible que ça soit lui qui se charge d'enregistrer ce texte dans un fichier.

Pour cela j'ai mis en place une page PHP permettant d'entrer du texte dans un formulaire :

```
<body>
  <form method="post">
    <input type="text" name="textdata"><br>
    <input type="submit" name="submit">

  </form>
</body>
```

Et d'enregistrer le contenu de ce formulaire dans un fichier texte :

```php
<?php
if(isset($_POST['textdata']))
{
  unlink('save.txt');
$data=$_POST['textdata'];
$fp = fopen('save.txt', 'a');
fwrite($fp, $data);
fclose($fp);
}
?>
```

Ainsi, cela simplifiait le problème, maintenant il suffisait de créer une procédure Lazarus permettant de remplir un formulaire en ligne automatiquement. Pour ce faire j'ai utilisé la librairie internettools (GNU) mettant à disposition la fonction query qui répondant à la problématique posée.

```
query('form((doc("http://'+IP+'/anno/'+Partie+'/'+Joueur+'/save.php"))//form)[1], {"textdata": $_1})', [save]).retrieve();
```

Figure 2: exemple de la fonction query qui envoie la string save à l'adresse précédente dans l'input de type textdata du formulaire de la page.

Maintenant qu'il était possible pour deux programmes de communiquer, il fallait l'adapter au jeu Anno1700, pour commencer il fallait pouvoir exploiter les données téléchargées et préparer les données à l'envoi.

Extraction des données téléchargées vers des variables (unitTraitementDeDonnees)

Il fallait commencer par lire le fichier sur lequel étaient stockées les données.

```
adresseTxt := 'saves/multi/'+Partie+'/'+Joueur+'/save.txt';          //lecture du fichier de sauvegarde
AssignFile(txtSaveFile, adresseTxt);
reset(txtSaveFile);
readln(txtSaveFile, strSave);
CloseFile(txtSaveFile);
```

Figure 3 : programme enregistrant le contenu d'un fichier texte dans une variable de type string.

Maintenant il fallait découper ce string pour séparer chaque donnée. Pour cela la librairie strUtils allait m'être utile ; elle contient la fonction ExtractDelimited qui permet d'extraire un mot d'un string, en indiquant sa position et le caractère qui sépare chaque mot. Ensuite il ne reste plus qu'à transcrire la variable string en variable entière puis à l'assigner à la variable du jeu souhaitée.

```
Setargent(StrToInt(ExtractDelimited(pargent,strSave,StdWordDelims)));
```

Cette ligne de commande extrait le mot à la position « pArgent » (qui équivaut à la position 2), provenant du string strSave est séparé par les caractères délimitant par défaut (ici : est utilisé). Puis la fonction setArgent assigne la valeur précédente à la variable Inv.Argent.

J'ai ainsi écrit la procédure ExtraireSesDonnesRessources permettant d'extraire toutes les données nécessaires au bon fonctionnement du jeu.

L'extraction des bâtiments nécessitait une itération pour remplir chaque case du tableau ile.

```
for ileItt := DayfellCay to VolcanoIsland do
begin
    for batItt := maison_colon to chantier_naval do
    begin
    //maison_colon
    ile[ileItt, batItt] := StrToInt(ExtractDelimited(pos,strSave,StdWordDelims));
    pos := pos + 1
    end;
end;
```

```
//extraction des batiments sur chaque iles vers le tableau ile
```

Compression des données pour l'envoi (unitTraitementDeDonnees)

La compression des données est principalement composée d'une concaténation des variables, utilisées par le jeu :

```
save :=  Concat(
Perso.nom+';'
+(IntToStr(getargent))+';'
+(IntToStr(gettissu))+';'
+(IntToStr(getbois))+';'
+(IntToStr(getpoisson))+';'
+(IntToStr(getlaine))+';'
+(IntToStr(getoutils))+';'
+(IntToStr(Pop.colon))+';'
+(InttoStr(Pop.citoyen))+';'
+(InttoStr(Pop.conscrit))+';'
+(InttoStr(Pop.soldat))+';'
+(InttoStr(Pop.fusilier))+';'
);
```

Puis encore un fois une itération permettant d'ajouter chaque bâtiment à la suite de la sauvegarde :

```
for ileItt := DayfellCay to VolcanoIsland do
begin
    for batItt := maison_colon to chantier_naval do
        save := concat(save+IntToStr(ile[ileItt, batItt])+';');

end;
```

```
adresseTxt := 'saves/multi/'+Partie+'/'+Joueur+'/save.txt';
AssignFile(txtSaveFile, adresseTxt);
rewrite(txtSaveFile);
writeln(txtSaveFile, save);
CloseFile(txtSaveFile);
```

Et enfin le tout est enregistré dans le fichier de sauvegarde.

Le prochain problème à résoudre était la synchronisation des deux programmes.

Pour cela j'ai mis en place le même système que pour la sauvegarde mais sous le nom de « prêt ». Les fonctions principales seront : rejoindre un serveur, attendre que l'autre joueur rejoigne et gérer les choix de fin de tours des joueurs (continuer ou quitter).

Attendre que l'autre joueur se connecte (unitMultijoueur)

```
function attendreJoueurs(IP, Partie : string):Integer;

var
   nbJoueurs, debut, nbAttente : Integer;
   nbAttenteStr : string;

begin
     nbJoueurs := StrToInt(Partie);
     nbAttente := nbJoueurs - 1;
     nbAttenteStr := IntToStr(nbAttente);
     debut := ChoixFinTour(IP, Partie);
     writeln('En attente de '+nbAttenteStr+' joueur(s)...');
     While (debut <> nbAttente) do
           begin
                sleep(1000);
                writeln('En attente de '+nbAttenteStr+' joueur(s)...');
                debut := ChoixFinTour(IP, Partie);
             end;
     ResetPret(IP, Partie, 'J2');

     EnvoyerPret(IP, Partie, 'J1');
     attendreJoueurs := 1;
end;
```

Cette fonction (utilisée uniquement par le joueur 1) appelle la fonction « choixFinTour » qui lie la valeur de « prêt » de l'autre joueur, jusqu'à ce que le joueur ait envoyé son signal de départ. (Cette fonction est adaptée à un mode de jeu comprenant plus de deux joueurs, mais finalement cela n'a pas abouti).

attendreJoueur renvoie la valeur 1 lorsque l'autre joueur est prêt à commencer la partie.

Rejoindre le serveur (unitMultijoueur)

```
procedure rejoindreServeur(IP, Partie, Joueur : String);
begin
     RemiseAZeroRessources(IP, Partie, Joueur2);
     CompresserDonneesRessource(Partie,Joueur2);
     EnvoyerSave(IP,Partie, Joueur2);
     EnvoyerPret(IP, Partie, Joueur);
  attenteServeur := attendreServeur(IP, Partie);
  while (attenteServeur = 0) do
        begin
          attenteServeur := attendreServeur(IP, Partie);
        end;
  writeln('Vous etes bien connecté et la partie peut commecer !');
  attenteServeur := 0;
  TelechargerSave(IP,Partie,Joueur1);
  ExtraireDonneesRessourceJ1(Partie);
end;
```

Cette fonction (utilisée uniquement par le joueur 2) remet à zéro les ressources du joueur 2, envoie ces données au serveur puis envoie le signal de départ pour indiquer au joueur 1 que le joueur 2 est prêt (c'est ce signal qu'attend la fonction précédente). Ensuite il appelle la fonction attenteServeur (qui renvoie 1 lorsque qu'il détecte le signal de départ du joueur 1) tant que le signal n'a pas été détecté. Une fois fait, la sauvegarde du joueur 1 est téléchargée.

Synchronisation des fins de tours joueur 1 (unitMultijoueur)

À la fin de chaque tour les joueurs ont le choix de continuer ou quitter la partie. Tant que les deux joueurs n'ont pas fait le choix la partie s'arrête. Si les deux joueurs souhaitent continuer alors la procédure FinNouveauTour est lancé, elle téléverse la sauvegarde du joueur utilisateur et télécharge le joueur du joueur adverse.

```
procedure ContinuerOUQuitter_S(IP, Partie : String);
var
   continue : integer;
begin
   CompresserDonneesRessource(Partie,Joueur1);
   EnvoyerSave(IP, Partie, Joueur1);
   writeln('Continuer : 1 | Quitter : 2');
   readln(continue);
   while ((continue <>1) AND (continue <> 2)) do
        begin
             writeln('Veuillez entrer une valeur correcte');
             readln(continue);
        end;
   case (continue) of
   1 : FinNouveauTourServeur(IP, Partie);
   2 : begin
        EnvoyerQuitter(IP, Partie, Joueur1);
        setQuitterlaPartie(TRUE);
        end;
   end;

end;
```

Maintenant que tout est réglé, il suffit de créer un mode de jeu semblable au mode solo mais en ajoutant les sous-programmes de synchronisation. Les changements sont :

La remise à 0 des fichiers présents sur le serveur (unitModesMulti)

```
Ra0Commerce(IP,Partie);                                  //
RemiseAZeroRessources(IP, Partie, Joueur1);              //Reset les données en prenant cette fois ci en compte le pseudo du joueur
MiseAJourDonnees_verser(IP,Partie, Joueur1);             //Televerse la sauvegarde du joueur 1
RemiseAZeroRessources(IP,Partie,Joueur1);                //Remises à zéro des fichier qui contenaient les données d'une précédente parti
RemiseAZeroRessources(IP,Partie,Joueur2);                //
```

Cela concerne les sauvegardes, le signal de départ et mais aussi le module prochain.

Des conditions spéciales en cas de nouveau tour (unitModesMulti)

```
if (getNvTour)=TRUE then
   begin
        DebutNouveauTour2J_S(IP,Partie);
        popUpCommerce(IP,Partie, Joueur1);
        Ra0Commerce(IP,Partie);
        MiseAJourDonnees_verser(IP,Partie,Joueur1);
        setNvTour(FALSE);
        ResetEchangePropose;
   end;
```

Ici DebutNouveauTour2J_S permet de remettre à zéro le signal de départ du joueur 1 et de mettre à jour les données des deux joueurs.

Fin de tour (unitModesMulti)

```
case choix of
    PASSER_AU_TOUR_SUIVANT : begin
                             TourSuivant;
                             DateSuivante;
                             AttaqueNaval();
                             ProgressionPlusUN;

                             setGainA0;
                             Prod_Globale;
                             SoustrairePoisson;
                             ConditionPirate;
                             ShowMenuMarchant;
                             ShowMenuFinDeTour;
                             TestVieBateau;
                             setEtatEmbarcation(false) ;
                             ContinuerOUQuitter_S(IP, Partie);
                             end;
```

Le seul ajout ici est la fonction continuer ou quitter qui attend le choix de l'autre joueur.

Les options spécifiques au multijoueur (unitModesMulti)

Une nouvelle option est disponible depuis le menu de l'ile :

```
ALLER_SUR_L_ILE_ADVERSE : begin
                          MiseAJourDonnees_charger(IP,Partie,Joueur1);
                          ShowMenuIleJ1(Partie, Joueur2);
                          end;
```

Il permet à un joueur de jeter un œil aux ressources et aux bâtiments de l'autre (les données de autres joueurs sont mises à jour juste avant.

Ce menu donne accès à une autre fonctionnalité : le commerce entre joueurs. Une fois de plus, le module commerce utilise le même système d'échange de donnés que les sauvegardes. Il fonctionne en trois parties : l'unité pour formuler un échange, l'unité pour communiquer avec le serveur et l'unité de réponse qui permet d'accepter ou refuser un échange puis de le résoudre.

Proposer un échange (proposerEchange)

```
procedure menuProposerEchange(joueur : string);
begin
    effacerEcran;
    DessinercadreCommerce(joueur);
    afficherSesRessources(90,13);
    couleurTexte(9);
    case joueur of
    'J1' : afficherRessourcesJ2(130,13);
    'J2' : afficherRessourcesJ1(130,13);
    end;
    illustrationCommerce(60,35);
    couleurTexte(11);
    optionCommerce;
end;
```

Cette procédure affiche les ressources des deux joueurs, puis la procédure optionCommerce donne plusieurs choix aux joueurs. Tout d'abord quelle ressource veut-il échanger :

```
repeat // ressource donnee                                    //Entrée du choix de la ressource à donner
  readln(choix);

  case choix of
  0 : begin   ...
  1 : begin
      ressourceDonnee := 1;
      y := 23;
      end;
  2 : begin   ...
  3 : begin   ...
  4 : begin   ...
  5 : begin   ...
  6 : begin   ...
  end;

  if ((choix < 0) OR (choix>6)) then                           //vérification de l'entrée du joueur
     begin
           deplacerCurseurXY(17,57);
           couleurTexte(12);
           write('/!\');
           couleurTexte(15);
           write('Veuillez écrire un choix valide!');
           readln;
     end;
  if (ressourceDonnee <> 0) then
  begin
  if (checkRessource(ressourceDonnee,1)=FALSE) then            //vérification que le joueur possède cette resource
     begin
          deplacerCurseurXY(17,57);
          couleurTexte(12);
          write('/!\');
          couleurTexte(15);
          write('Vous ne pouvez pas échanger une ressource dont vous ne disposez pas.');
          choix := -1;
     end;
```

Cela assigne la valeur ressourceDonnee au choix du joueur. Le joueur ne peut que donner une ressource qu'il possède, cela est vérifié grâce à la fonction checkRessource qui renvoie faux si un joueur a moins d'unité que voulu de la ressource choisie.

Ensuite combien de cette ressource veut-il donner :

```
if not annule then                                             //Entrée de la quantité à donner
   begin
   //Qte donnee
   deplacerCurseurXY(80,21);
   write('Quantité à proposer :');
   repeat
         deplacerCurseurXY(90,y);
         write('     ');
         deplacerCurseurXY(90,y);
         readln(qteDonnee);

         if (checkRessource(ressourceDonnee,QteDonnee)=FALSE) then   //vérification que le joueur possède au moins la quantité à donner
         begin
              deplacerCurseurXY(17,57);
              couleurTexte(12);
              write('/!\');
              couleurTexte(15);
              write('Vous ne pouvez pas proposer plus de ressource que vous n''en disposez.');
         end;
   until(checkRessource(ressourceDonnee,QteDonnee)=TRUE);
```

Tout d'abord on remarque que ce programme se lance uniquement si le joueur n'a pas décidé d'annuler l'échange. Ensuite tout comme à la première étape, la condition checkRessource(ressourceDonnee,QteDonnee) = FALSE vérifie que le joueur propose bien de donner une quantité de ressource qu'il possède.

Les deux prochaines étapes sont très semblables aux deux première mais cette fois ci sans les vérifications.

Figure 4 : Une proposition d'échange de 5 poissons contre 100 d'argents

Ensuite on prélève du joueur qui a proposé l'échange les ressources qu'il souhaite donner. Ainsi il ne peut pas dépenser des ressources qu'il n'aura peut-être plus si l'échange est accepté (la caution sera rendue si l'échange est refusé).

Une fois sorti de ce menu (échange validé ou annulé).

```
    //caution des ressources
    case RessourceDonnee of
1:   begin
     nvPerdu := (getPoisson - QteDonnee);
     SetPoisson(nvPerdu);
     end;
2:   begin
     nvPerdu := (getBois - QteDonnee);
     SetBois(nvPerdu);
     end;
3:   begin
     nvPerdu := (getOutils - QteDonnee);
     SetOutils(nvPerdu);
     end;
4:   begin
     nvPerdu := (getLaine - QteDonnee);
     SetLaine(nvPerdu);
     end;
5:   begin
     nvPerdu := (getTissu - QteDonnee);
     SetTissu(nvPerdu);
     end;
6:   begin
     nvPerdu := (getArgent - QteDonnee);
     SetArgent(nvPerdu);
     end;
end;
```

```
case choix of
RETOUR : deplacerCurseurXY(17,57);
PROPOSER_UN_ECHANGE : begin
                        menuProposerEchange(joueur);
                        if getRessourceDonnee<>0 then
                        begin
                             envoiEchange(IP, Partie, Joueur);
                             echangePropose := TRUE;
                        end;
                      end;
end;
```

L'échange est envoyé s'il a été formulé grâce à l'unité CommerceBackEnd.

Fonctionnalités de CommerceBackEnd

Envoyer un échange, tout comme pour les sauvegardes, ce programme envoie à l'aide la procédure envoyerCommerce, une querry. Cette fois ci les éléments contenus dans l'échange à envoyer sont les variables précédemment assignées lors de la phase proposer un échange.

```
procedure envoiEchange(IP, Partie, joueur : string);
var
  echange : string;

 begin
      echange := concat('1;'+IntToStr(getRessourceDonnee)+';'+IntToStr(getQteDonnee)+';'+IntToStr(getRessourceVoulue)+';'+IntToStr(getQteVoulue)+';0;');
      write('Proposition bien envoyée');
      envoyerCommerce(IP, Partie, joueur, echange);
 end;
```

```
//Envoie la proposition d'écange d'un joueur au serveur HTTP
```

Résumer un échange, permet simplement de créer un string contenant une phrase résumant l'échange proposé à partir de des données du serveur.

```
LireCommerce(IP,Partie,Joueur);
adresseTxt := 'saves/multi/'+Partie+'/'+Joueur+'/commerce.txt';
AssignFile(txtSaveFile, adresseTxt);
reset(txtSaveFile);
readln(txtSaveFile, strSave);
CloseFile(txtSaveFile);

RessourceDonnee := ExtractDelimited(2,strSave,StdWordDelims);
QteDonnee := StrToInt(ExtractDelimited(3,strSave,StdWordDelims));
RessourceVoulue := ExtractDelimited(4,strSave,StdWordDelims);
QteVoulue := StrToInt(ExtractDelimited(5,strSave,StdWordDelims));
```

```
resumeEchange := concat(nom,' propose un échange : il vous donne ', IntToStr(QteDonnee), ' ', RessourceDonnee, ' contre ', IntToStr(QteVoulue), ' ', RessourceVoulue,'.');
```

Attendre la réponse de l'autre joueur. checkReponse est une fonction qui renvoi la réponse du joueur choisi. Si c'est 0 alors le joueur n'a pas encore fait son choix, si c'est 1, le joueur a accepté et enfin si c'est 2 il a refusé. Une fois une de ces deux dernières valeurs obtenues, la fonction attendreAouR prend la valeur 1 si l'autre joueur a accepté et 2 s'il a refusé.

```
function attendreAouR(IP, Partie, Joueur : string):integer;
var
 reponse : integer;
 moi : string;
begin
     attendreAouR := 0;
     repeat
          reponse := checkReponse(IP, Partie, Joueur);
     until((reponse = 1) OR (reponse = 2));

     case joueur of
     'J1' : moi := 'J2';
     'J2' : moi := 'J1';
     end;

     case reponse of
     1 : attendreAouR := 1;
     2 : attendreAouR := 2;

     end;
end;
```

Envoyer Accepter/Refuser

```
procedure envoyerAccepter(IP, Partie, joueur : string);                    //Envoie au serveur les données de l'échange formulé (0 si aucun formulé
var
echange : string;

begin
     echange := concat('1;'+IntToStr(getRessourceDonnee)+';'+IntToStr(getQteDonnee)+';'+IntToStr(getRessourceVoulue)+';'+IntToStr(getQteVoulue)+';1;');
     envoyerCommerce(IP, Partie, joueur, echange);
end;
```

Ces deux procédures similaires envoient une valeur à la fin du fichier commerce du joueur choisi, cette valeur témoigne du choix du joueur (Refuser = 2 et Accepter = 1)

Répondre à l'échange (reponseEchange)

```
if ((checkEchange(IP, Partie, 'J2') = TRUE) AND (getRessourceDonnee_S(IP, Partie, 'J2')<>0)) then //si J2 veut échanger
begin
  if moi = 'J2' then
  begin
    afficherAttenteReponse('J1');
    acceptee := attendreAouR(IP, Partie, 'J1');
    case acceptee of
    1 : afficherRecapAccepter(IP, Partie, 'J2', FALSE);
    2 : begin
        remiseCaution(IP,Partie,'J2');
        afficherRefus('J1');
        end;
    end;
  end
  else //si moi = J1 et J2 veut échanger
  begin
    afficherEchange(IP, Partie, 'J2');
    reponse := accepterOuRefuser(Ip, Partie, 'J2');
    case reponse of
    1 : begin
        envoyerAccepter(IP,Partie,moi);
        afficherRecapAccepter(IP, Partie, 'J2', TRUE);
        end;
    2 : begin
        envoyerRefuser(IP, Partie, moi);
        afficherRecapRefuser;
        end;
    end;
  end;
  end;
  //personne ne veut échanger = il ne se passe rien
end;
```

Tiré de la procédure popUpCommerce, ce programme permet de vérifier si un joueur a proposé un échange, afficher un message à l'autre joueur lui demandant s'il souhaite l'accepter ou non (si le joueur n'a pas assez de ressource pour l'accepter cela sera aussi affiché) puis l'échange est résolu. S'il est accepté, un récapitulatif des ressources échangées est affiché, sinon la caution est rendue.

Figure 5 : Pop-up de la proposition d'échange (début du tour)

Figure 6 : récapitulatif de l'échange accepté (côté joueur client)

Insertion dans le programme principal

```
if choix=3 then
begin
     modePartie2J(TRUE);
end;

if choix=4 then
begin
     rejoindrePartie2J();
end;
```

Il est possible de choisir de jouer en multijoueur depuis le menu principal (choix 3 et 4)

Voici donc tous les sous programmes permettant à deux joueurs de jouer ensemble.

Liste des unités

Programme principal :

ProgrammePrincipal

Le programme principal va gérer toutes les initialisations, mises en relation des unités et va appeler certaines procédures en fonction des choix du joueur.

Unités générales :

AchatArmee

Unité contenant les procédures pour acheter différente unité pour défendre ses îles, tel que le conscrit, le soldat ou encore le fusilier. Chacun des achats vont consommer des ressources.

AchatBatiments

Unité contenant les procédures pour acheter différent bâtiment sur l'île choisie, tel que les maisons, les bâtiments sociaux et militaire ou encore les industries. Chacun des achats vont consommer des ressources.

AchatNavire

Unité contenant les procédures pour acheter différent navire, tel que le navire de colonisation, d'exploration ou encore de transport. Chacun des achats vont consommer des ressources.

AmeliorationBateau

Unité contenant la possibilité au joueur de pouvoir effectuer l'amélioration du mat, des voiles et de la coque de ses différents navires pour augmenter la résistance de ces derniers. Chaque amélioration effectuée va offrir +1 point de vie au bateau et va utiliser des ressources au joueur. Chaque navire a 9 améliorations possibles sur chaque bateau.

CombatNaval

Unité qui gère les combats navals pouvant infliger des dégâts aux navires d'exploration. Cette unité va donc calculer la vie restante pour les navires en exploration et effectuer des actions en fonction de leurs points de vie. Chaque navire a 15% de chance de se faire attaquer.

CombatPirate

Unité qui gère les combats ponctuels face à des pirates. Le nombre de pirates qui apparait est proportionnel au nombre de tour effectué. Les Pirates affrontent vos troupes, chaque opposant possèdes ses points de vie et ses points d'attaque.

Difficulte

Unité qui va définir un niveau de difficulté en fonction du choix du joueur.

GestionArmees

Unité qui va offrir une interface au joueur l'option de gérer ses troupes et en acheter s'il le souhaite. En cas d'achat, cette unité fera appelle à l'unité « AchatArmee ».

GestionBatiments

Unité qui va offrir une interface au joueur l'option de gérer ses bâtiments en fonction de l'île choisi par ce dernier et en acheter s'il le souhaite. En cas d'achat, cette unité fera appelle à l'unité « AchatBatiments ».

GestionEcran

Unité qui permet de stocker plusieurs procédures permettant d'être appelé pour gérer l'interface graphique.

GestionNavires

Unité qui va offrir une interface au joueur l'option de gérer ses navires, en acheter s'il le souhaite ou les améliorer. En cas d'achat, cette unité fera appelle à l'unité « achatNavires ». En cas d'amélioration, cette unité fera appelle à l'unité « AmeliorationBateau »

IleSuivante

Unité qui va permettre au joueur, s'il possède un bateau d'exploration arrivé à destination, de changer d'île et de ainsi de la découvrir pour pouvoir construire des bâtiments dessus.

Marchant

Unité permettant l'apparition aléatoire d'un PNJ (personnage non joueur) pour pouvoir vendre et acheté des ressources avec ce dernier en fonction des ressources détenues et des prix fixés.

MenuConnexion

Unité qui gère l'affichage du logo du jeu et du logo de l'Equipe. Elle possède aussi la procédure permettant au joueur de quitter.

MenuIle

Unité affichant la liste des ressources, des bâtiments possédés et de la population (Habitants et Armées). Fait office d'interface générale pour circuler sur les différentes interfaces de gestion.

MenuNouvellePartie

Unité possédant le choix initialise pour lancer soit une nouvelle partie seule ou en multijoueur, ou la possibilité de quitter.

Population

Unité possédante et calculant le nombre de chaque classe de population.

Production

Unité calculant le rendement de chaque bâtiments industriels possédés par le joueur et ajoutant la production des ressources dans les ressources du joueur.

Ressource

Unité possédant l'inventaire de ressource du joueur.

ResumeFinDePartie

Unité d'affichage des ressources consommés par la population et produite par les industries. Elle va aussi afficher l'avancement des navires d'exploration et résumé les points de vie perdu par ces derniers en cas d'attaque navale.

Temps

Unité qui va changer la date afficher dans le MenuIle et comptabilisé le nombre de tour effectué depuis le lancement de la partie.

Unités pour le Multijoueur :

CommerceBackEnd

Unité qui gère les échanges d'informations entre le module de commerce du joueur et le serveur.

MenusMulti

Unité qui s'occupe d'afficher les menus du joueur adverse.

ProposerCommerce

Unité permettant à un joueur de proposer un échange au joueur adverse via un menu.

ReponseCommerce

Unité qui s'occupe de la réponse d'un joueur de la proposition d'échange de l'autre.

unitHttp

Unité qui gère les échanges entre le programme et le serveur http.

unitModesMulti

Unité contenant les programmes principaux du mode multi.

unitMultijoueur

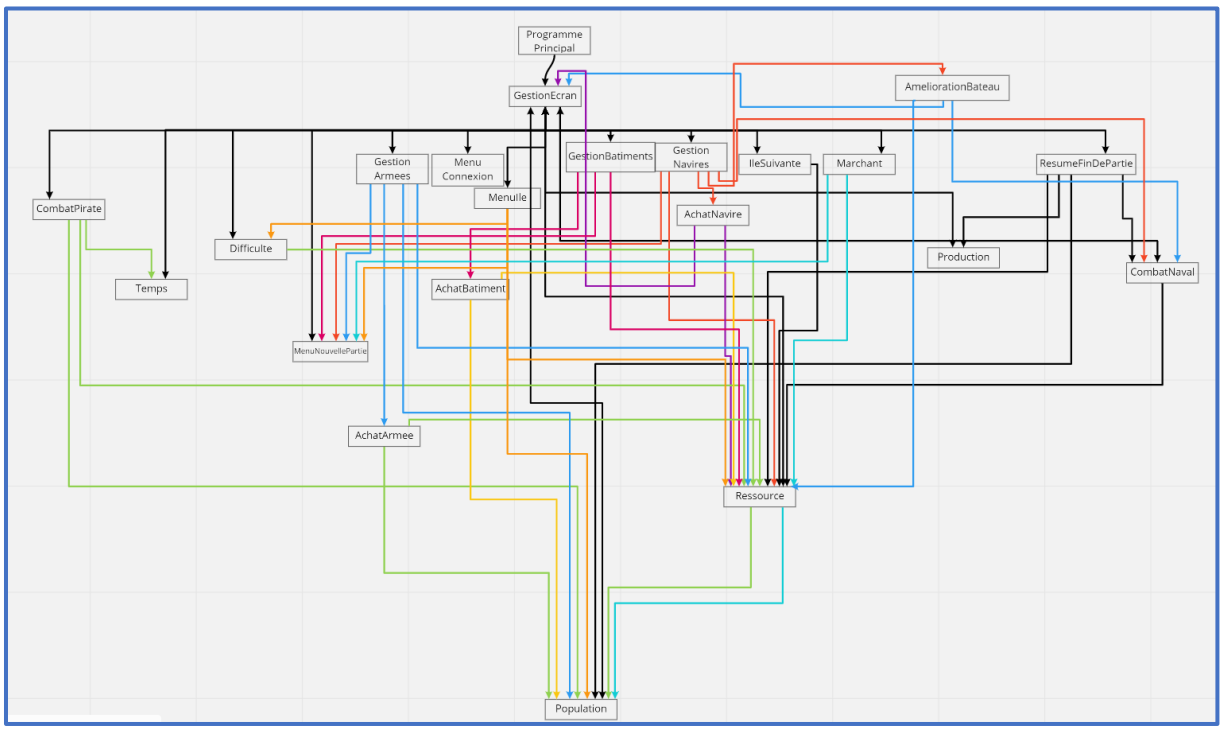
Unité qui s'occupe de la synchronisation des deux programmes (joueur 1 et joueur 2).

unitTraitementDonnees

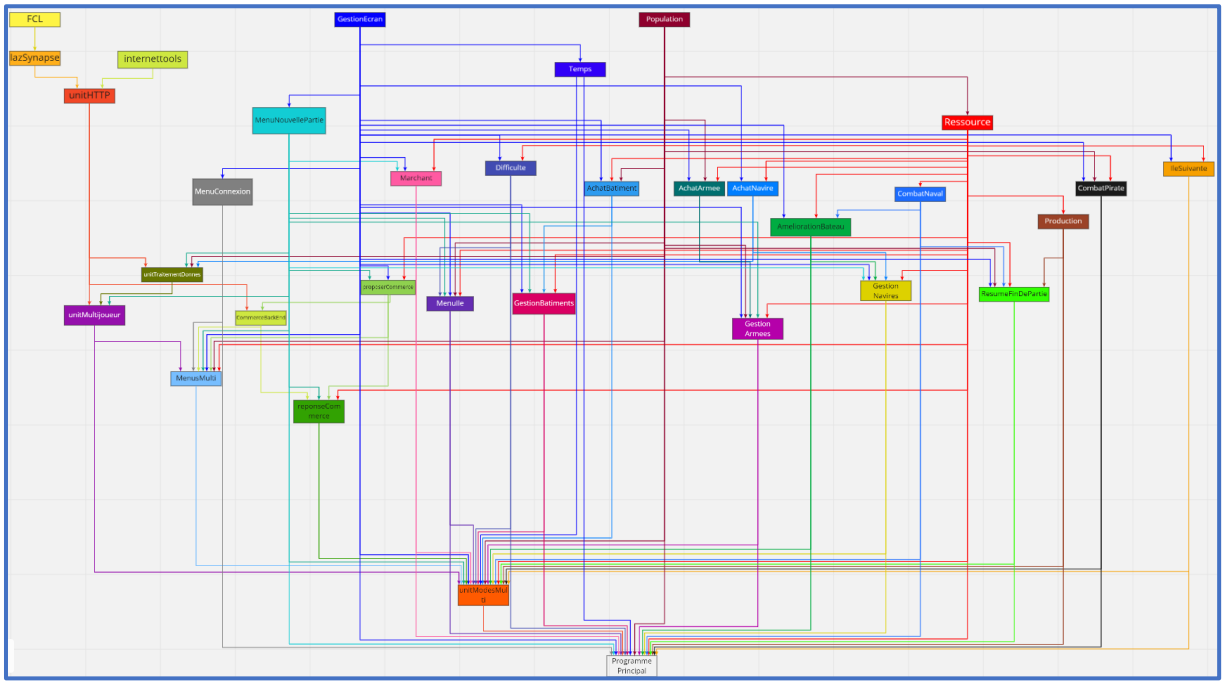
Unité qui gère le traitement des données des fichiers de sauvegarde pour enregistrer les variables dans le fichier ou enregistrer les données du fichier dans les variables du programme.

Graphe de dépendance

1er version (sans le multijoueur)



2ème version (avec le multijoueur)



Présentation d'une fonction et d'une procédure

Fonction getTotal

```
function getTotal(b : batiments) : integer;
var
  ileItt : iles;
begin
   getTotal := 0;
   for ileItt := DayfellCay to VolcanoIsland do
   begin
     getTotal := getTotal + ile[ileItt,b];
   end;
end;
```

PROGRAMME : fonction getTotal(b : bâtiments) : integer;
ROLE : Cette fonction renvoie le nombre d'un bâtiment choisi.

Glossaire

DONNEES ENTRANTES :
b : un bâtiment. Le type de bâtiment que l'on souhaite dénombrer.
Voici la déclaration du type bâtiments :

```
batiments = (maison_colon, maison_citoyen, entrepots, centres_villes, chapelle, cabanes_de_bucheron,
            cabanes_de_pecheur, bergeries, ateliers_de_tisserands, armureries, chantier_naval);
```

Ce type liste tous les bâtiments disponibles sur une ile.
ile[iles,bâtiments], un tableau de type batimentIle (des iles x les bâtiments d'entiers).
Déclaré au début de l'unité :

```
batimentIle = array [iles, batiments] of integer;
```

Ce tableau permet de stocker le nombre de chaque bâtiment par île.

VARIABLE DE TRAVAIL :
ileItt de type iles. Il va servir à itérer le tableau iles

```
iles = (DayfellCay,Soupex,GreyTerminal,WaterSeven,Amazonia,Luden,Croomelt,Skeld,VolcanoIsland);
```

Le type iles liste les iles disponibles.

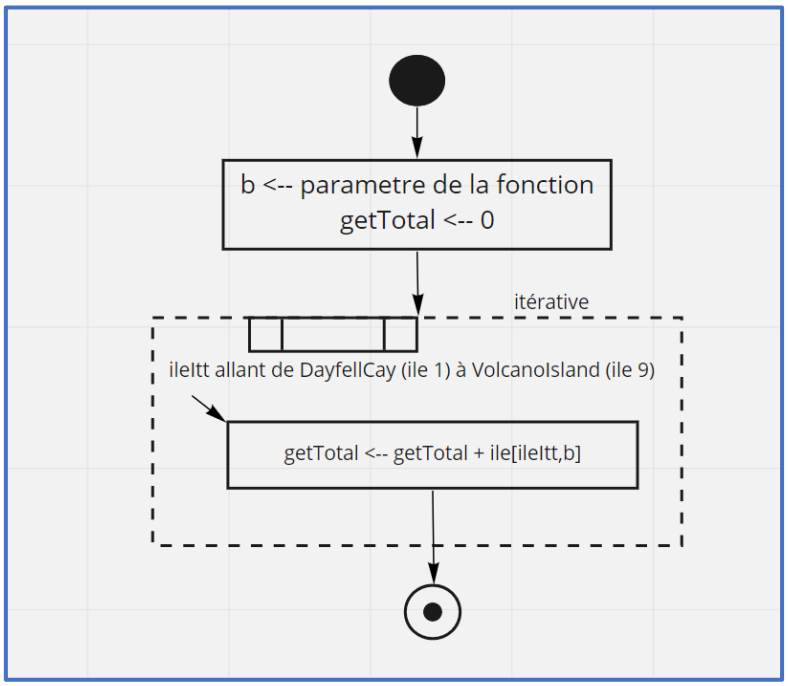
RESULTAT RECHERCHE :
getTotal : integer.
Il stockera le nombre d'un bâtiment.

Principe

```
b : batiment appelé dans la fonction
getTotal <- 0
Pour ileItt de ile1 à ile9 :
      getTotal <- getTotal + ile[iltItt,b];
```

Pour ileItt de ile1 à ile9 va parcourir toutes les iles, ensuite on ajoute au total le nombre de ce bâtiment sur l'ile itérée.

Diagramme d'activité



Jeux d'essais

Ile	Nb de maison_colon	getTotal(maison_colon)
DayfellCay	3	3
Soupex	2	5
GreyTerminal	1	6
WaterSeven	5	11
Amazonia	3	14

Luden	4	18
Croomelt	2	20
Skeld	0	20
VolcanoIsland	1	21

Je valide

Cette fonction est principalement utilisée pour afficher le nombre de bâtiments total d'un joueur dans le résumé.

Procédure achat_maison_colon

```
procedure achat_maison_colon(i : iles);

var

   diff:integer;

begin
   if ((getbois) >= 2) then
     begin
          diff := (getbois()-2);
          setBois(diff);
          ile[i,maison_colon]:= ((ile[i,maison_colon])+1);

          setColon(getColon)+1));

          effacerecran;
          dessinerCadreXY(10,3,190,25,Simple,11,0);
          dessinerCadreXY(80,1,125,5,Simple,11,0);
          deplacerCurseurXY(99,3);
          CouleurTexte(10);
          write(' MAISON ');
          deplacerCurseurXY(15,11);
          write('Achat réussi');
          deplacerCurseurXY(27,11);

     end
   else
       begin
         effacerecran;
         dessinerCadreXY(10,3,190,25,Simple,11,0);
         dessinerCadreXY(80,1,125,5,Simple,11,0);
         deplacerCurseurXY(99,3);
         CouleurTexte(11);
         write(' MAISON ');
         deplacerCurseurXY(15,12);
         write('Vous ne pouvez pas acheter cette structure');
         deplacerCurseurXY(57,11);
       end;



end;
```

PROGRAMME : procedure achat_maison_colon(i :iles);

ROLE : Cette procédure permet d'acheter une maison de colon sur une ile donnée

Glossaire

DONNEES ENTRANTES :
i de type iles (vu précédemment), l'ile sur laquelle on souhaite acheter la maison.
Le tableau ile contenant le nombre de chaque bâtiment sur chacune des iles.

VARIABLE DE TRAVAIL :
diff : integer, le nombre de bois que possède un joueur (obtenu grâce à la fonction getBois) auquel on soustrait le cout du bâtiment (ici 2)

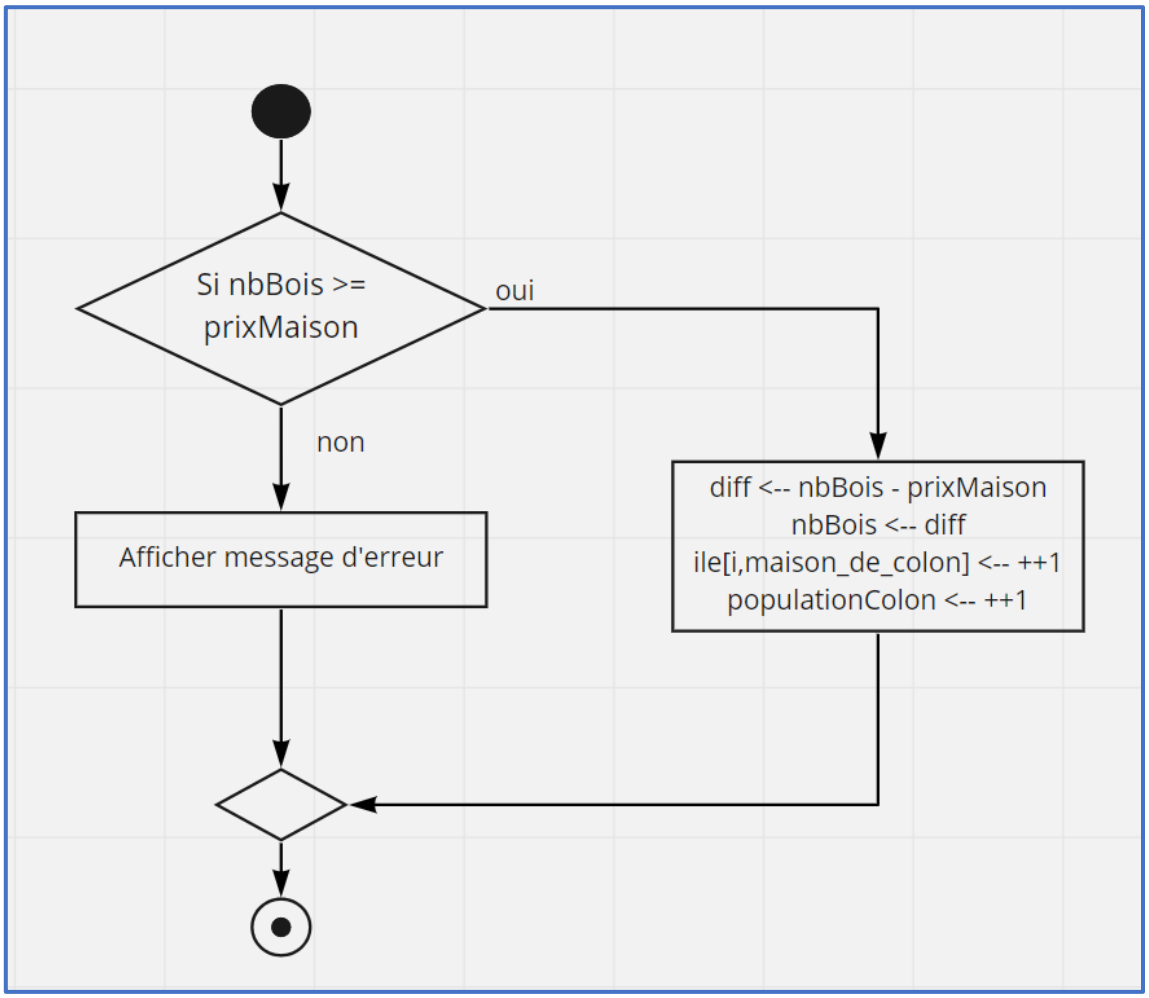
RESULTAT RECHERCHE : savoir s'il est possible pour le joueur d'acheter une maison de colon, si oui ajouter +1 maison et +1 colon, sinon afficher un message d'erreur.

Principe

```
Si nbBois >= prixMaison
      diff ← nbBois - prixMaison
      nbBois ← diff
      ile[ile,maison_de_colon] ← ile[ile,maison_de_colon] +1
      populationColon ← populationColon + 1
sinon
      afficherMessageDerreur
```

Pour travailler avec les variables qui ne viennent pas de la procédure, nous avons utilisé des sous-programmes adaptés (ici par exemple nbBois := la fonction getBois)

Diagramme d'activité



Jeux d'essais

Sur une ile donnée

Avant			*Après*		
Nombre de Bois	Nombre de maison de colon	Population de colon	Nombre de bois	Nombre de maison de colon	Population de colon
5	0	0	3	1	1
1	2	1	ERREUR : achat impossible		

Le jeu d'essai est validé.