Projet **Rigobot**

Livrable L3 : Rapport final

Équipe **Rigobot**

22 décembre 2002

Résumé

Ce document a pour but d'expliquer les orientations prisent par le projet **Rigobot** [1], que se soit au niveau technique qu'à des niveaux plus abstraits. **Rigobot** est un projet pensé et développé au sein du module projet de deuxième année de MIM à l'ENS-Lyon. C'est un programme éducatif permettant l'apprentissage des bases de la programmation à des enfants de primaire et de collège.

1. `http://savannah.nongnu.org/projects/rigobot/`

Table des matières

1 Présentation

1.1 Le projet Rigobot

Le projet **Rigobot** a vu le jour dans le cadre du module de Conduite de Projet de la deuxième année de MIM à l'École Normale supérieure de Lyon.

Son but est de créer un logiciel intégré d'apprentissage de la programmation objet et événementielle, destiné à des enfants de primaire et de collège.

Dans cette optique, l'élève/utilisateur est aux commandes d'un robot plongé dans un monde aux graphismes en trois dimensions : il doit diriger le robot au moyen d'un langage de programmation impératif simple et proche du français.

1.2 Membres

L'équipe **Rigobot** est composée de : Aurélien Moreau, Antoni Gutsens Diaz, Benjamin Leveque, Damien Pous, David Roger, Lionel Vaux, Mathieu Martin, Nicolas Bernard, Samuel Thibault, Simon Capern et Thomas Gazagnaire, tous élèves du MIM-01/02.

Un intervenant externe, Jean-Michel Ouvry, est également impliqué dans l'élaboration du design 3D.

2 Description

2.1 Contexte

Le langage LOGO a été, pendant une dizaine d'années, enseigné aux élèves de primaire. La simplicité de sa syntaxe en faisait un langage idéal pour débuter, pour apprendre les concepts fondamentaux de la programmation. La compréhension et la maîtrise de ces concepts sont utiles pour de nombreux domaines autres que l'informatique (raisonnement, logique, mathématiques, *etc.*).

Mais LOGO, comparé à d'autres logiciels sur le marché, n'est plus du tout attractif pour des enfants habitués aux jeux vidéo en 3D, avec des graphismes très évolués. Le but du projet Rigobot est de créer un langage d'apprentissage de la programmation qui tienne compte de ce constat, tout en restant pédagogique.

2.2 Objectifs du projet

Dans le contexte évoqué plus haut, le projet Rigobot s'est donné les deux objectifs suivants :

- Gestion d'un monde virtuel en 3 dimensions, soumis à des lois physiques (gravité, collisions, ...). Dans ce monde, l'enfant devra gérer un robot qui va être soumis à différentes épreuves. Le but de l'enfant sera de programmer son robot afin qu'il puisse passer victorieusement les épreuves.
- Définition d'une syntaxe simple mais suffisamment puissante pour pouvoir contrôler le robot à sa guise. De plus, ce langage doit pouvoir exposer les bases de la programmation objet (héritage, méthodes) et évènementielle (notion de gestionnaire d'évènement).

3 Présentation des différents axes de développement

3.1 Organisation

Une vue d'ensemble de l'organisation est présentée en figure 1

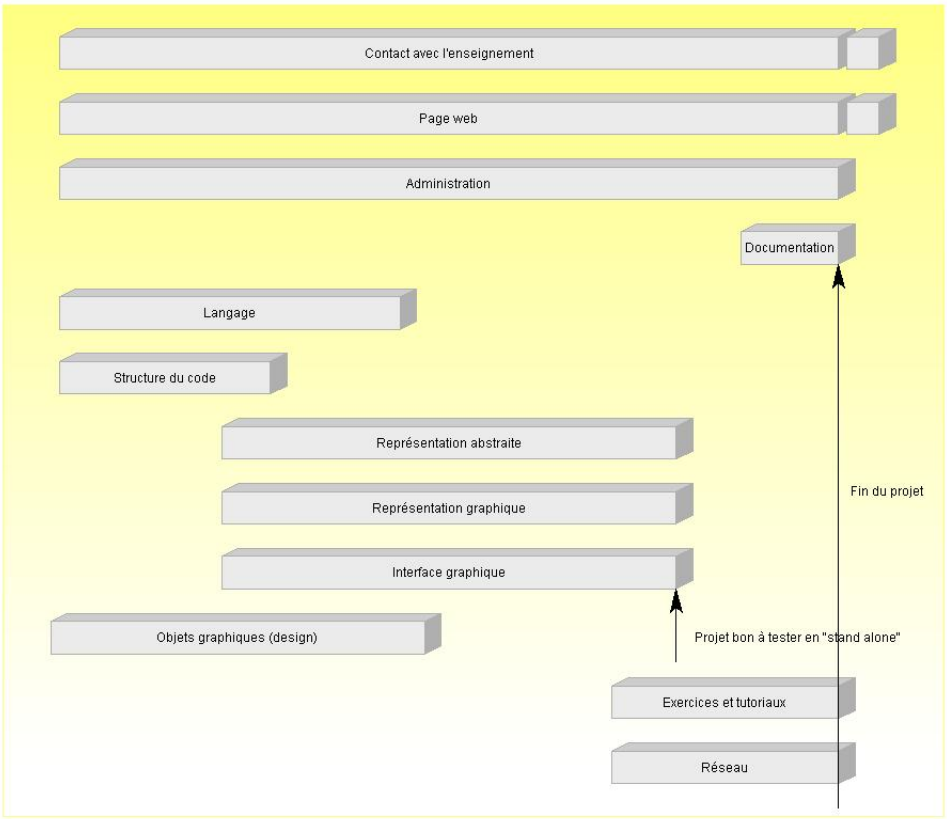


FIGURE 1 – Sous-projets Rigobot

3.2 Comment lire ce qui suit

Chacune des sous-sections suivantes décrit en détails un sous-projet. Le nom de la sous-section est le titre du sous-projet. Chaque description est structurée comme suit :

3.3 Description de cet axe de développement

Acteurs :
Les acteurs du sous-projet, *i.e.* ses membres, dont le chef de sous-projet.

Description :
Une description succinte du sous-projet.

Futur :
Le bilan du sous-projet, ce qu'il reste à faire et quelques notes sur les orientations futures possibles.

3.4 Détails techniques

Rapport technique, *i.e* le détail des réalisations effectuées.

4 Administration

4.1 Description de cet axe de développement

Acteurs :
Aurélien Moreau, Damien Pous, Lionel Vaux et Samuel Thibault.

Description :
Le sous-projet *Administration* est chargé de coordonner les efforts des membres du projet.

Futur :
Ce travail est perpétuel et ne se terminera pas avec la fin du module projet. Il sera interressant d'activer et de se servir des différents outils offerts par Savannahpour améliorer la qualité de notre logiciel.

4.2 Détails techniques

Le travail de ce sous-projet se poursuit tout au long des diverses phases de la genèse de **Rigobot**. Il est le fruit d'un effort commun qui se manifeste en particulier sur la liste de diffusion du projet et au cours des réunions hebdomadaires. Voici les différents parties supportées par le sous-projet administration :

4.2.1 Réunions

Les réunions **Rigobot** ont été suivies de comptes-rendus publiés d'abord sur la liste de diffusion, puis sur le site Internet.

4.2.2 Espace de développement

La gestion régulière de l'espace fourni par Savannahcomprennant entre autre un CVS.

4.2.3 Coordination

Elle s'est manifesté lorsqu'il falait faire un état des lieux des travaux en cours ou encore pour rassembler les contributions de chacun lors de l'édition commune de documents (livrables par exemple).

4.2.4 Collecte

Le CVS du projet contient sous l'arborescence `admin` une série de notes synthétiques résumant les réflexions échangées au cours des réunions ou sur les fils de la liste de diffusion, à destination de sous-projets particuliers ou s'adressant à l'ensemble des membres de l'équipe.

5 Contact avec l'enseignement

5.1 Description de cet axe de développement

Acteurs :
Lionel Vaux et Aurélien Moreau.

Description :
Ce sous-projet est destiné à maintenir un lien avec le monde de l'enseignement afin de mieux cibler notre logiciel. Dans un premier temps, il rapporte aux programmeurs les exigences des professeurs en ce qui concerne l'apprentissage de la programmation. Il peut ensuite trouver des classes de test pour pouvoir adapter le logiciel à son public et corriger les éventuels choix malencontrueux. Il peut aussi de façon parallèle prendre contact avec des associations de distribution de logiciels éducatifs pour proposer le logiciel fini.

Futur :
Ce sous-projet c'est trouvé être pour l'instant majoritèrement un échec, en effet les différents contacts tentés dans l' enseignement se sont trouver être infructueux mis à part monsieur Vaux (coord) qui nous à montrer un certain intérêt pour **Rigobot**. Les raisons qui ont provoqué cet échec sont multiples, d'une part les enseignants du primaire affichent en général une certaine réticense envers l'informatique trop technique et donc ne se sente pas en mesure d'apporter un enseignement dans un domaine qu'il ne maîtrise pas eu-même. Un autre point important qui a été soulevé est l'obligation aux enseignants désireux d'innover dans leur classes de proposer un projet pédagogique, chose fastidieuse et que peut d'enseignant sont près à mettre en place. Enfin notre projet ne s'inscrit pas du tout dans les lignes directrices que le ministère de l'enseignement c'est fixé en ce qui concerne l'informatique à l'école, priviligiant l'aspect pratique de l'ordinateur. Ceci vient d'un amalgame entre *l'informatique* en tant que science et l'utilisation d'un ordinateur qui sont deux choses totalement différentes mais pas contradictoire. Ainsi quand les enseignants ou le ministère de l'enseignement parle d'informatique, ils utilisent le mot non pas dans le sens de science mais dans celui d'utilisation d'un ordinateur. Mais la situation n'est pas désespéré, loin de là, il faut juste persévérer en proposant des logiciels de qualité permettant peut-être un jour de réintroduire la programmation dans les écoles non spécialisées. Nous allons donc continuer à prospecter afin de trouver d'autres enseignants motivés et essayer d'adapter notre logiciel à leurs besoins.

5.2 Détails techniques

5.2.1 Recherche d'enseignants

Cet aspect du sous-projet devait être primordial car il devait apporter des visions autres que celles des informaticiens sur la difficile question de l'apprentissage de la programmation dans le milieu scolaire. Il devait apporter des réponses sur les différentes interrogations

que le programmeur peut se poser sur l'orientation du logiciel, et pourquoi pas donner de nouvelles idées.

5.2.2 Classes de test

Un des autres objectifs de ce sous-projet était de trouver une ou plusieurs classes de test afin de valider notre logiciel, voire de le modifier pour encore mieux coller aux exigences des professeurs, et enfin, de calibrer la difficulté des exercices.

5.2.3 Distribution

Une fois le logiciel rodé sur les classes de test, on peut envisager une distribution dans les écoles intéressées et ceci au travers d'organisations spécialisées ou d'associations. Le sous-projet *contact avec l'enseignement* devait prendre contact avec ces organismes pour faire connaître le logiciel.

6 Site web

6.1 Description de cet axe de développement

Acteurs :
Thomas Gazagnaire Damien Pous

Description :
Gestion du site web du projet **Rig^{o}b^{o}t**.

6.2 Détails techniques

Le site web [2] est découpé en deux parties :

6.2.1 Partie privée

Elle est accessible uniquement aux membres du projet. Elle permet de coordonner les sous-projets en permettant à chaque membre du projet d'avoir une idée globale de l'avancement du projet. Cette partie est utile uniquement lors de la phase de développement.
Elle comprend une archive des différents documents relatifs au projet comme les comptes rendus de réunions, les documents clés retraçant l'évolution du projet, etc... On y retrouve aussi quelques renseignements sur les sous-projets (les membres, l'état d'avancement, ...). D'une manière générale, tout document relatif à l'élaboration du projet devra s'y retrouver.

6.2.2 Partie publique

Elle sera destinée au grand public et représentera la vitrine du logiciel. Y figurera une description adaptée aux enseignants et aux élèves du logiciel **Rig^{o}b^{o}t**. On pourra bien sûr télécharger les sources et les versions binaires sous de multiples plate-formes.

2. http ://www.nongnu.org/rigobot/

7 Structure du code

7.1 Description de cet axe de développement

Acteurs :
Damien Pous

Description :
Projet installant les première brique de code du projet.

Futur :
Ce sous projet avait une signification à la mise en route du projet pertmettant de bien commencer le codage. Il n'a plus vraiment de raison d'être dans la suite du projet.

7.2 Détails techniques

Ce sous-projet a étudié la structure globale du code, afin d'en extraire les différents blocs et leurs interdépendances. Cette analyse a permis de bien définir le travail et les interfaces à fournir dans chaque partie du code, et d'assurer la cohérence de l'ensemble.

La structure ainsi définie devait donc, en plus d'être adaptée au projet, être encodable dans le langage choisi (à savoir, Ocaml), et permettre une compilation aisée et modulaire du logiciel.

Ce sous-projet visait donc plusieurs objectifs :

— définir une structure adéquate au langage de programmation, dans laquelle les différents sous-projets sont bien définis, ainsi que leurs interfaces ;
— coder le squelette de cette structure, et vérifier qu'elle est cohérente ;
— écrire des Makefiles permettant de compiler le tout facilement ;
— maintenir ces Makefiles, et les porter sous MS-Windows ;
— assurer la cohérence globale du code, tout au long de sa définition.

La structure retenue est présentée sur la figure 7.2.

— Le bloc *language* définit le langage utilisé par les élèves. Sur la demande de l'affichage, ce module fournit un objet `prgm`, correspondant au programme d'un robot. Cet objet est alors *inséré* dans un robot pour définir son comportement : via une méthode `get`, un robot peut obtenir la prochaine action qu'il doit effectuer ; des évènements peuvent aussi être déclenchés par une méthode `declare`.
— Le bloc *graphic interface* s'occupe de tout le fenêtrage de l'application. À priori, pour des raisons de portabilité, cette interface est faite avec la bibliothèque `Tk`, cependant, afin de s'en abstraire, ce module fournit une interface générique.
— Le bloc *abstract world* définit le monde de manière abstraite, c'est-à-dire sans se soucier de l'affichage. Le comportement des objets évoluant dans le monde y est défini, ainsi que l'évolution globale du monde.

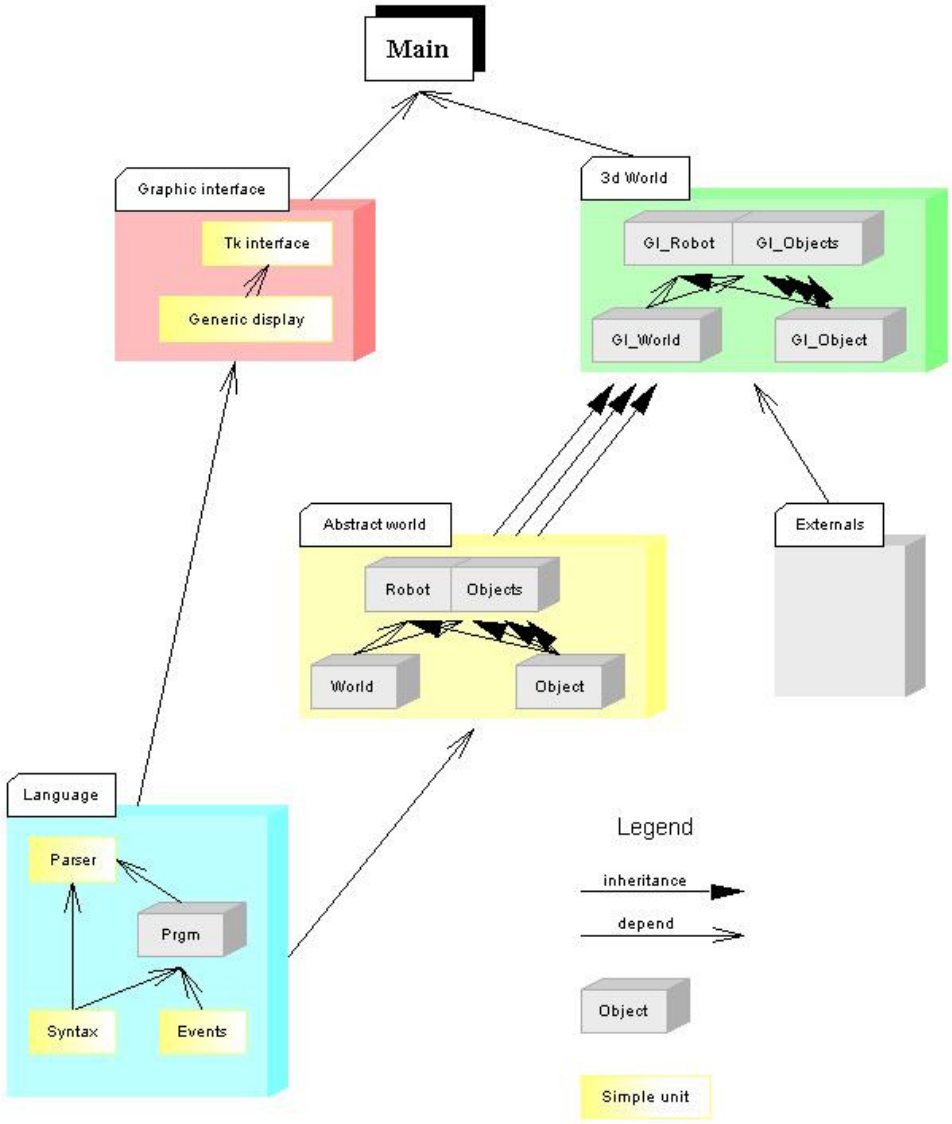


FIGURE 2 – Structure choisie pour le code

— Le bloc *3d world* définit au contraire seulement l'affichage 3d du monde et de ses objets. Les classes qui y sont définies héritent des classes de *abstract world* et définissent ce dont elles ont besoin pour être dessinées.
— *externals* est une librairie regroupant tout l'interfaçage C/Ocaml dont on peut avoir besoin. Mettre ce code à part permettra une compilation plus aisée sous Windows.

L'unité *Main* se charge de rassembler le tout : elle crée un monde, ainsi qu'un robot, et les fait tourner via l'interface graphique, en mode interactif (l'élève entre une commande puis en constate les effets) ou

pas (le monde évolue sans intervention de l'élève).

Le modèle monothread a été choisi, car il permet un meilleur contrôle, ainsi qu'une meilleure portabilité Windows/Unix. Le taux de rafraîchissement du monde, ainsi que sa vitesse d'évolution du monde sont alors déterminés par deux *timers* distincts, fournis par l'interface graphique.

8 Langage

8.1 Description de cet axe de développement

Acteurs :
David Roger, Thomas Gazagnaire, Aurélien Moreau,
Damien Pous, Lionel Vaux

Description :
Axe de développement étudiant et construisant le langage de programmation utilisé pour programmé le robot.

Futur :
Ce sous-projet avait pour objectif de définir le langage qu'utiliseraient les élèves pour programmer leur robot, puis de réaliser un interpréteur qui transformera le code écrit avec ce langage en une suite d'instructions atomiques faciles à comprendre par le monde abstrait.. Ce langage devait être en même temps simple et puissant, et extrêmement facile à déboguer.

8.2 Détails techniques

L'implémentation de l'arbre syntaxique (annexe B), du typeur(annexe A) et de l'interpréteur(annexe C) ont été repoussé en annexe pour se consacré ici seulement à la description du langage, mais ses implémentations font biensûre parti du sous-projet.

8.2.1 Contexte

Le projet **RigObOt** constitue une tentative pour élaborer un logiciel d'apprentissage de la programmation, le langage associé doit donc présenter certains aspects particuliers.

Un langage impératif s'impose : on cherche à transmettre des notions de programmation objet et évènementielle, pas fonctionnelle. Il doit être simple d'accès et assez proche du français, ou plutôt d'ordres en français : il s'agit de commander un robot. L'utilisateur « s' adresse » au robot, par des phrases construites à l'impératif.

Même si le choix a été fait de pouvoir traiter un maximum d'exercices sans variables, nous avons décidé de les conserver, afin de ne pas trop appauvrir le langage.
De même, la définition de procédures et de fonctions est possible dans le langage, et pas seulement via un bouton « définir une nouvelle fonction ».

Enfin, le langage limite l'usage des parenthèses : les arguments d'une fonction sont simplement donnés à la suite. Une telle convention ne doit toutefois pas être confondue avec un aspect fonctionnel : une fonction n'est pas une valeur et on n'autorise pas d'application partielle.

Nom	Temps	Description
`av`	1	avance d'une case
`re`	1	recule d'une case
`td`	1	tourne d'un quart de tour vers la droite
`tg`	1	tourne d'un quart de tour vers la gauche
`dt`	1	demi-tour
`avd`	1	tourne à droite et avance d'une case
`avg`	1	tourne à gauche et avance d'une case
`sa`	2	saute une case
`dm`	1	démarre le marquage au sol
`am`	1	arrête le marquage au sol
`pr`	1	prend un objet sur la case courante
`po`	1	pose un objet sur la case courante
`act`	1	active un objet (*e.g.* interrupteur) sur la case courante
`ut`	1	utilise un objet

TABLE 1 – Primitives du langage Rigobot

8.2.2 Spécifications du langage

Un programme **Rigobot** est constitué de phrases. Chaque phrase est composée d'une séquence d'instructions séparées par des virgules (`,`), suivie d'un point (`.`).

On distingue instruction et expression : une instruction « fait quelque chose », elle n'a pas de résultat ; une expression a un résultat, qui peut être typé (entier, booléen, chaîne de caractères, *etc.*).

Procédures Les procédures sont des instructions. L'appel à une procédure se fait en écrivant son nom, suivi de ses éventuels arguments. Si une procédure n'a pas d'argument, son nom suffit. Par exemple :
> `av.`
fait avancer le robot d'une case et
> `dis "bonjour".`
affiche `bonjour` dans la zone des messages. Les arguments des procédures sont des expressions.

Les procédures primitives sont présentées en table 1.

Boucles On a trois types de boucles.

Boucle conditionnelle :
> `tantque` *cond* `fais` *seq* `fin.`
répète la séquence *seq* tant que la valeur (booléenne) de *cond* est satisfaite.

Boucle itérative :
> `fais` *expr* `fois` *seq* `fin.`
calcule la valeur (entière) n de l'expression *expr* et répète n fois la séquence *seq*.

Boucle infinie :
> `repete` *seq* `fin.`
répète indéfiniment la séquence *seq* (le seul moyen de sortir d'une telle boucle est de recevoir un évènement).

Affectations On peut stocker la valeur d'une expression dans une variable :
> `soit a =` *expr*`.`
et changer la valeur d'une variable :
> `change a =` *expr*`.`
Une affectation est une instruction.

Branchement Le cas du `si` est particulier : la structure obtenue en l'utilisant peut être une expression ou une instruction. Dans :
> `si` *cond* `alors av sinon re fin.`
on a une instruction alors que dans :
> `soit a = si` *cond* `alors 4 sinon 5 fin.`
c'est la valeur d'une expression qui est affectée à `a`.

Constantes Les expressions de base sont les constantes. Un entier est un mot de $[1-9][0-9]^*$. Un booléen est `vrai` ou `faux`. Une chaîne est une séquence de caractères encadrée par des guillemets (`"`).

Variables Une variable est une expression et sa valeur est celle de la dernière expression qui lui a été affectée *au moment même de l'affectation*. Après :
> `soit a = 1, soit b = a, change a = 2.`
la valeur de `b` est 1.

Opérateurs On dispose d'opérateurs dont la liste est représentée en table 2. Dans
> `soit a =` $expr_1 \oplus expr_2$`.`
où $\oplus$ est un opérateur, $expr_1 \oplus expr_2$ est une expression dont la valeur est $v_1 \oplus v_2$, avec v_i la valeur de e_i.

Fonctions Les fonctions sont l'équivalent des procédures en termes d'expressions. Par exemple :
```
> si (f 32 52) = 1664
> alors dis "f est sûrement la multiplication"
> fin.
```

Expression précédée d'instructions Une séquence d'instructions suivie d'une virgule puis d'une expression est une expression. L'évaluation de cette expression complexe provoque l'exécution de la séquence, et sa valeur est celle de l'expression terminale après l'exécution de la séquence.

Nom	Type des opérandes	Type de retour	Description
`+`	*entier*	*entier*	addition
`-`	*entier*	*entier*	soustraction
`*`	*entier*	*entier*	multiplication
`/`	*entier*	*entier*	division
`^`	*chaîne*	*chaîne*	concaténation
`=`	*entier*	*booléen*	égalité
`!=`	*entier*	*booléen*	non égalité
`<`	*entier*	*booléen*	inférieur strict
`>`	*entier*	*booléen*	supérieur strict
`<=`	*entier*	*booléen*	inférieur
`>=`	*entier*	*booléen*	supérieur

TABLE 2 – Opérateurs binaires du langage Rigobot

Définition de procédures et fonctions On définit une procédure par :
> `soit p arg1 arg2 ... argn =` *seq*`.`
et une fonction par :
> `soit f arg1 arg2 ... argn =` *expr*`.`
où *seq* est une séquence d'instructions et *expr* est une expression. On ne sait pas syntaxiquement si on définit une procédure ou une fonction : ce sera au typeur (voir annexe A) de reconnaître que *seq* est une séquence d'instructions (*i.e.* n'a pas de valeur) ou que *expr* est une expression (éventuellement précédée d'instructions). Bien évidemment, ni *seq* ni *expr* ne sont évaluées lors de la déclaration : elles sont seulement typées.

Lors de l'appel `p` $expr_1$ $expr_2$ `...`$expr_n$, la valeur de chaque $expr_i$ est calculée, puis la séquence *seq* est exécutée, toutes les occurrences libres des arguments ayant été remplacées par les valeurs correspondantes.

La valeur de l'appel `f` $expr_1$ $expr_2$ `...`$expr_n$ est celle de l'expression *expr* au moment de l'appel, et dans laquelle toutes les occurrences libres des arguments sont remplacées par les valeurs des $expr_i$.

Pour définir une procédure ou une fonction sans argument, on remplace `arg1 arg2 ... argn` par `()` (il est à noter que `()` n'apparait que dans la définition, pas dans l'appel). Une fonction sans argument est différente d'une variable en ce sens qu'une fonction n'a pas de valeur : c'est l'appel à la fonction qui en a une. La valeur d'une variable est celle obtenue au moment de l'affectation, alors que la valeur de l'appel à une fonction est calculée au moment même de l'appel. Par exemple :
> `soit a = 32, soit b = a * 52, change a = 0.`
> `repete b fois av fin.`
fais avancer le robot alors que :
> `soit a = 32. soit f () = a * 52.`
> `change a = 0, repete f fois av fin.`
ne le fait pas bouger.

N.B. : une définition de fonction ou de procédure n'est pas une instruction et ne peut donc être incluse dans une séquence d'instructions. Elle se termine obligatoirement par un point : c'est une phrase à part entière. On ne peut donc définir de fonction ou de procédure dans le corps d'une autre fonction ou procédure.

8.2.3 Parsing

Une des difficultés du parsing dans **Rigobot** réside dans la différenciation instruction/expression : même si cette différenciation est surtout sémantique, il faut s'assurer lors de l'analyse syntaxique que ce qui est parsé en tant qu'instruction *peut* effectivement en être une, et inversement que ce qui est parsé en tant qu'expression *peut* avoir une valeur. Le doute restant sera levé au typage. D'autres difficultés de parsing sont liées à la spécification du langage : par exemple, un nom de variable peut représenter une variable même, ou un appel de fonction/procédure sans argument.

On distingue trois catégories d'expressions :
— *unit* : les instructions ;
— *non unit* : les expressions au sens « objet qui possède une valeur » ;
— *maybe unit* : les expressions pour lesquelles on ne peut pas décider syntaxiquement si elles sont des instructions *unit* ou non.

Cette distinction permet d'organiser efficacement le parsing.

Les procédures primitives sont *unit*, de même que les boucles et les définitions. Les constantes et les opérations binaires sont *non unit*. Les variables, applications de fonctions et branchements conditionnels sont *maybe unit*. Les expressions précédées d'une séquence restent dans la même catégorie.

On peut alors définir deux catégories non disjointes :
— *expected unit* : rassemble les *unit* et les *maybe unit* ;
— *expected non unit* : rassemble les *non unit* et les *maybe unit*.

Quand lors du parsing, on s'attend à une instruction, on parse en *expected unit*, et quand on s'attend à une expression représentant une valeur, on parse en *expected non unit*.

L'application de ce principe conduit aux règles présentées en annexe D.

9 Monde abstrait

9.1 Description de cet axe de développement

Acteurs :
Damien Pous, Lionel Vaux, Samuel Thibault, Nicolas Bernard et Thomas Gazagnaire

Description :
Ce sous-projet s'occupe de la partie du code qui définit le monde virtuel et son évolution. Y sont définies :

- une classe `world` : le monde abstrait, comprenant principalement un tableau de cases,
- une classe `cell` : une case, comprenant principalement une liste d'objets,
- une classe générique virtuelle `objet`, dont hériteront tous les objets évoluant dans le monde,
- une classe spéciale `robot`, descendant de `objet`, qui définit un robot et son comportement, avec l'aide de l'objet `prgm` fourni par le sous-projet langage,
- des classes diverses et variées, héritant de `objet`, afin de définir tous les objets pouvant apparaître dans le monde (champignons, murs, ...). Certains de ces objets ne seront pas visibles : ils ne serviront qu'à déclencher des évènements.

Futur :
Outre la réalisation d'un monde qui fonctionne, ce sous-projet été chargé de définir tous les objets nécessaires, fruits de notre imagination ; et de les doter de nombreuses fonctionalités, en parallèle avec le sous-projet langage, afin d'obtenir une grande expressivité / diversité au niveau des exercices que nous proposeront par la suite.

9.2 Détails techniques

Le monde abstrait est au centre du projet, dans la mesure où c'est lui qui fixe les règles du jeu, mais aussi car sa structure définit précisément les liens entre les différents objets manipulés (cartes, robots, champignons...)

Cette partie du code ne définit pas la manière dont les objets sont rendus à l'affichage. C'est la partie *monde graphique* qui s'en occupe, en étendant les classes du monde abstrait. Ce sont ces classes, qui au final, interagissent pour donner vie au monde du robot.

Cette partie ne pose pas de problèmes particuliers au niveau algorithmique, c'est surtout son organisation qui est délicate. Je commente donc les signatures des différentes classes.

9.2.1 Monde

La classe monde (`world_t`) définit un plateau de jeu, et centralise la gestion de celui-ci. Le paramètre de type (`'obj`) de la classe signifie que l'on a un monde dont les objets sont de types `'obj`. Cette astuce

est nécessaire, car, le type des objets qui seront posés sur la carte n'est défini que plus loin, dans l'arbre des dépendances du projet.

```
class type ['obj] world_t =
object
  method cell: pos_t -> 'obj cell_t
  method cell_list: pos_t list -> 'obj cell_t list
  method deplace: 'obj -> dir_t -> unit
  method register_generator: (unit -> Events.event list) -> unit
  method register_listener: (Events.event -> unit) -> unit
  method register_reacter: (unit -> unit) -> unit
  method won: unit
  method lost: unit
  method next: status_t
  method reset: unit
end
```

— `cell` : accès à une case du plateau de jeu. Si l'on sort du tableau, une case spéciale (`invalid_cell`) nous est retournée, permettant ainsi de ne pas avoir à s'occuper de la gestion des *bords*.
— `cell_list` : accès à une liste de cases, à partir d'une liste de positions
— `deplace` : déplacement d'un objet, dans la direction donnée. Si l'objet n'a *pas le droit* d'effectuer ce déplacement, une exception est levée. (l'appelant pourra alors générer un évènement...)
— `register_generator` : enregistrement d'un nouveau *générateur potentiel*. La fonction passée en argument sera appelée à chaque tour. Cette fonction peut générer des évènements sur un certain nombre d'objets, et renvoie une liste d'évènements à générer de manière globale.
— `register_listener` : enregistrement d'un nouvel *écouteur potentiel*. La fonction passée en argument sera appliquée à chaque top, sur chacun des évènements ayant été générés globalement.
— `register_reacter` : enregistrement d'un nouveau *réacteur potentiel*. La fonction passée en argument sera appelée à chaque top, permettant ainsi à un objet de faire ce dont il a envie.
— `won / lost` : ces deux méthodes servent à déclarer la partie gagnée ou perdue. Elles doivent être appelées par les générateurs d'évènements. Lorsque plusieurs robots pourront être mis en concurrence, il faudra préciser cette notion de *gagné/perdu*.
— `reset` : réinitialisation du monde
— `next` : c'est la méthode qui fait réagir le monde : elle est appelée régulièrement par un *timer*, et s'occupe de faire réagir chacun des objets de la carte. On commence par appeler tous les *générateurs potentiels*, et concaténer les listes d'évènements ainsi obtenues, puis on diffuse cette liste à tous les *écouteurs potentiels*, enfin, on appelle successivement tous les *réacteurs potentiels*

9.2.2 Cases

La classe `cell_t` définit une case du monde. Une instance de cette classe correspond à une position entière sur la carte, et contient un ensemble d'objets, dont l'un d'entre eux est éventuellement ramassable.

```
class type ['obj] cell_t =
object
  method pos: pos_t
  method objets: 'obj list
  method rem: 'obj -> unit
  method accept: 'obj -> dir_t -> unit
  method takable: 'obj option
  method obj_height: float
end
```

— `pos` : la position (entière) de la case sur le plateau
— `objets` : la liste des objets présents sur cette case
— `rem` : retrait d'un objet
— `accept` : un objet souhaite se déplacer *vers* nous, dans une direction donnée. Cette méthode vérifie que les objets présents sur la case n'empêchent pas un tel mouvement, puis effectue le déplacement (retrait de l'ancienne case, ajout sur la nouvelle). Si l'objet n'a pas le droit d'effectuer ce mouvement, une exception est levée.
— `takable` : renvoie l'éventuel objet ramassable se trouvant sur la case
— `obj_height` : certains objets modifient la hauteur à laquelle les autres objets doivent évoluer. Cette méthode fait la somme des modifications engendrées par les objets de la case.

9.2.3 Objets

La classe `objet_t` est une classe de base pour tous les objets évoluant dans le monde (champignons, murs, robots...). De nombreuses méthodes y sont définies, certaines avec des valeurs par défaut, devant être surchargées lors de la création d'objets plus spécifiques.

La paramétrisation de la classe est plus complexe :
— `'world` spécifie le type de monde dans lequel évolue l'objet
— `'coerce` est le type de l'objet, vu de l'extérieur (méthode `coerce`)

```
class type ['world, 'coerce] objet_t =
object
  method id: int

  method is: class_t -> bool

  method cell: 'coerce cell_t
  method set_cell: 'coerce cell_t -> unit
  method world: 'world
  method pos: pos_t
  method zone: pos_t list
  method dir: dir_t
```

```
  method move: dir_t -> unit

  method generate: Events.event list
  method notify: Events.event -> unit
  method react: unit

  method is_valid: 'coerce -> dir_t -> bool

  method incr_obj_height: float

  method takable: bool
  method useme: 'coerce -> bool
end
```

— `id` : identificateur unique pour l'objet
— `is` : l'objet fait-il partie d'une classe donnée (par exemple `C_Mushroom`, `C_Robot`...)
— `cell` : case sur laquelle se trouve l'objet
— `set_cell` : changement de case
— `world` : monde dans lequel se trouve l'objet
— `pos` : position sur la carte (=`cell#pos`
— `zone` : zone d'influence sur la carte
— `dir` : direction courante (entière)
— `move` : déplacement dans une direction donnée
— `generate` : évènements générés par l'objet (cf. `world#register_generator`)
— `notify` : notification d'évènements (cf. `world#register_listener`)
— `react` : réaction, lors d'un tour (cf. `world#register_reacter`)
— `is_valid` : un objet donné peut-il arriver sur la case dans la direction donnée ?
— `incr_obj_height` : modification de la hauteur à laquelle les objets doivent passer sur la case (par exemple pour un pont...)
— `takable` : l'objet est-il ramassable ?
— `useme` : si oui, que faire lorsque l'on est utilisé ?

Remarque : cette classe n'enregistre pas automatiquement ses méthodes `generate`, `notify` et `react` dans le monde, c'est aux classes descendantes de le faire, selon ce qu'elles font.

9.2.4 Robot

La classe `robot` hérite de objet, et définit un robot. Pour ce faire, on lui rajoute un *lecteur de disquettes*, dans lequel, on vient insérer l'objet `prgm` défini par le code de l'élève.

On implémente donc la méthode `react` en lisant une action dans le flot fournit par `prgm` puis on l'exécute. Dans le cas d'un déplacement, s'il échoue, on génère l'évènement, puis on donne une seconde chance au robot en relisant une action dans le flot et en l'exécutant. Cela permet par exemple un passage *fluide* d'une haie : l'élève rentre `repete av fin.` dans le gestionnaire d'évènement `Start`, et `saute.` dans le gestionnaire `onHedge` ; grâce à la deuxième chance, le robot

peut déclencher son saut au même top que la découverte de la haie.

10 Représentation graphique

10.1 Description de cet axe de développement

Acteurs :
Aurélien Moreau et Simon Capern

Description :
Cet axe de développement s'occupe de l'affichage $3D$ du monde virtuel. Les composantes graphiques étant fournies par le sous-projet *design*, nous devions mettre en place les structures permettant l'importation et l'intégration des données de ce projet.

Futur :
Les résultats obtenus sur cette première implémentation sont très prometteurs, un des gros problème est la demande en ressource de la gestion 3d. Il faut donc optimiser encore cette partie. Il reste encore a implémenter une gestion des animations de déplacement plus réaliste. Cette axe de développement sera intensivement poursuivi par la suite.

10.2 Détails techniques

10.2.1 Introduction

Rigobot étant un logiciel avant tout éducatif, nous ne pouvions nous permettre de gérer le graphisme 3d comme pour un jeux.
En effet il existe plusieurs restrictions auxquelles nous avons du faire face. Les jeux s'effectuant en mode plein écran, on peut alors modifier à loisir la résolution, de plus l'environnement de fenêtrage n'est alors plus à gérer, ce qui augmente les performances d'affichage.
Nous devons donc créer un environnemnt 3d pouvant être affiché uniquement dans une petite fenêtre avec un niveau de performances acceptable (20-30 fps). Une partie de notre travail est donc l'optimisation de tout ce qui va être rendu.

10.2.2 Moteur 3d

Le moteur $3d$ est fourni par *OpenGl*. On utilise *Lablgl* interface pour caml d'*OpenGl* version 1.0 ou 1.1 (certaines fonctionnalités d'OpenGl ne sont alors pas disponibles : multitexturages, on ne pourra donc pas faire aussi joli que $QuakeIII$...).
De plus cette bibliothèque pour caml n'est pas très complète, en effet certaines manipulations sur les textures ne sont pas implémentées comme par exemple la mise en mémoire graphique des textures. Il a donc fallu compléter lablGl par une interface $caml/C$ pour une meilleure gestion des textures.

10.2.3 Fog et clipping

Dans un souci d'optimisation, la gestion du fog (brouillard) et du clipping est quasi indispensable. Car même si le brouillard prend un

petit peu plus de ressources à l'affichage on y gagne car on se permet alors de réduire la distance au plan limite de profondeur (clipping).
Cette section est encore en développement car le brouillard est difficile à gérer. En effet, le fog est une manière de modifier les couleurs des objets pour donner l'impression de distance et de profondeur. Opengl n'ajoute pas des pixels supplémentaires pour créer un brouillard il va juste atténuer les couleurs des objets suivant différentes fonctions définies par l'utilisateur.
Trois types de brouillard sont alors disponibles :
— le mode *linear* où l'atténuation est alors linéaire par rapport à la distance oeil-objet.
— deux modes *exp* et *exp*2 qui modifient alors les couleurs de manière logarithmique.
Mais le rendu de chacun de ces modes dépend des capacités hardware de l'utilisateur.
Il nous reste donc deux choix, mettre en place un brouillard commun avec des résultats peu appréciables ou alors permettre à l'utilisateur de paramètrer le fog por l'adapter à ses ressources.
Pour l'instant la première solution était choisie, mais au vu des résultats catastrophiques sur certaines implémentations d'OpenGl (MesaGl, faibles ressources, ...), nous allons mettre en place un panel de configuration qui nous permettra d'optimiser le rendu sans contre-performances.

10.2.4 Robot

Le robot est donc composé de deux choses :
— Un squelette articulé par des noeuds de transformation à chaque articulation.
— Un ensemble d'objets qui vont composer les membres du robot.
La structure globale du robot a été relativement facile à implémenter grâce aux manipulations de matrices d'Opengl.

Articulation L'implémentation de ce modèle d'articulation se fait avec une classe très utile : *gl_node*

```
class gl_node :
  object
    val mutable obj   : model          (*objet*)
    val mutable child : gl_node list   (*liste de fils*)
    val mutable root  : gl_node option (*parent*)
    val mutable i     : 'a
    val mutable pos_x : float          (*position par rapport au pere*)
    val mutable pos_y : float
    val mutable pos_z : float
    val mutable posx  : float          (*position dans son repere local*)
    val mutable posy  : float
    val mutable posz  : float

    val mutable rot_x : float          (*rotation par rapport au pere*)
```

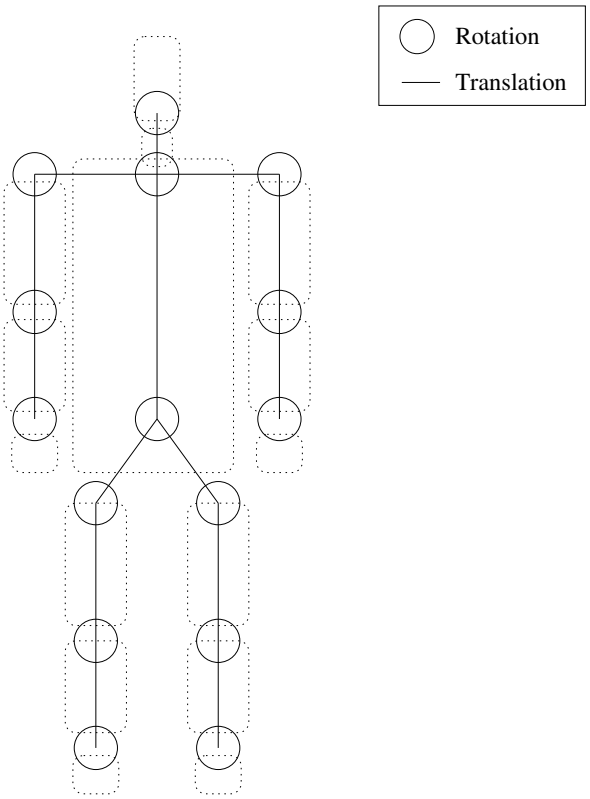


FIGURE 3 – Modèle d'articulation du robot

```
    val mutable rot_y : float
    val mutable rot_z : float
    val mutable rotx  : float          (*rotation dans son repere local*)
    val mutable roty  : float
    val mutable rotz  : float
  end
```

Chaque membre est alors un objet *gl_node* avec un unique parent (*root*) et une liste de fils (*child*).
Par exemple le tronc n'a pas de parent mais 5 fils (2 bras, 2 jambes et 1 nuque).
Pour l'affichage proprement dit on utilise le système de matrice d'OpenGl.

```
GlMat.push();

   GlMat.translate3 (pos_x,pos_y,pos_z);   (*
   GlMat.rotate3 (rot_x) (1.0,0.0,0.0);      Translation et rotation globale du membre
   GlMat.rotate3 (rot_y) (0.0,1.0,0.0);      et de tous ces fils
   GlMat.rotate3 (rot_z) (0.0,0.0,1.0);    *)

   GlMat.push();
```

```
    GlMat.rotate3 (rotx) (1.0,0.0,0.0);   (*
    GlMat.rotate3 (roty) (0.0,1.0,0.0);     Translation et rotation propre du membre
    GlMat.rotate3 (rotz) (0.0,0.0,1.0);     c.a.d les modifications du modèle de l'ob
    GlMat.translate3 (posx,posy,posz)     *)
    obj#draw                              (* affichage du modele qui subit ses propres
  GlMat.pop();                            . et les transformations globles   *)

  child#draw;                             (* affichage des fils qui subissent
                                             la transformation du père       *)
GlMat.pop();
```

On utilise $GlMat.push()/GlMat.pop()$ pour localiser les transformations sur les objets compris entre les 'push' et les 'pop'.

Membres Chaque membre associé à un objet gl_node est un objet au format *.obj* que l'on va traiter de la même manière que les objets du terrain.
Tout cela est décrit dans la section suivante.

10.2.5 Objet 3d

Les objets que nous utilisons dans **RigObOt** sont réalisés par le sous-projet design qui nous livre alors des modèles au format *.obj*.

Description du format Obj

— une liste de lignes décrivant les sommets
v a b c
a, b et c étant alors les coordonnées dans le repère courant.
— une liste de lignes décrivant les normales de chaque sommet
vn a b c
a, b et c étant alors les coordonnées du vecteur normal dans le repère courant.
— une liste de lignes décrivant les normales de chaque sommet
vt a b [c]
a, b et en option c (pour les textures 3d) étant alors les coordonnées de la texture .
— une définition de groupe
g label
— une liste de lignes décrivant les triangles
f v1[/vt1][/vn1] v2[/vt2][/vn2] v3[/vt3][/vn3]
v[nt]1 v[nt]2 v[nt]3 correspondent aux numéros des lignes des définitions des sommets.
Chaque groupe de sommets est délimité par les labels de groupe définis précédemment.
— une définition du 'material' (couleur, texture, transparence) usemlt label
le label doit être défini dans le fichier mlt.

Nous utilisons ensuite une librairie Glm (codée en C) pour charger et manipuler ces objets. Nous avons du effectuer des modifications pour de meilleurs résultats notamment pour la gestion des textures.

Avant l'utilisation de ces objets nous devons opérer plusieurs traitements :

— Unitize : On translate et on redimensionne (sans changer les proportions) le modèle pour qu'il tienne dans un cube d'une unité *OpenGl* de coté centré en zéro.
— facetNormals : On crée les normales à chaque face et on les normalise (de longueur égal à 1).
— vertexNormals : On crée les normales à chaque sommet et on les normalise (de longueur égal à 1).
— Weld : On fait une élimination de sommets pour simplifier et enlever les sommets trop proches qui ne font que ralentir le moteur $3d$.

10.2.6 La carte de jeu 3d

Introduction **Rigobot** doit offrir un monde 3d où évoleura le robot. Celui-ci se déplacera sur une aire d'exercice, appelée carte, jonchée de différents obstacles et objets. Nous nous sommes orientés vers deux solutions au début du projet, une gestion orientée plus "d'intérieur" où le robot se déplacerait dans une maison, un gymnase ou autres grandes salles, et une autre où les exercices se dérouleraient à l'air libre, à l'extérieur. La première possibilité permet de n'utiliser principalement que des surfaces planes (en ce qui concerne la carte) ce qui produit un rendu simple et rapide, la deuxième, quant à elle, n'offre pas une telle simplicité que se soit pour la description de la carte (et donc son stockage) ou son rendu. Nous avons pourtant choisi la seconde, malgré les difficultés qu'elle entraînait, misant sur son côté plus attractif. Un point important qui a conditionné certains des différents choix pris par la suite est la volonté d'avoir une création de carte simple et pouvant se faire aléatoirement.

Champ de hauteurs : height fields La génération du terrain de jeu se base dans un premier temps sur une matrice de flottants, celle-ci représentant un échantillonnage des mesures de l'altitude d'un terrain.
Ce type d'entrée permet une grande liberté dans la conception de l'exercice, ainsi il n'est pas nécessaire d'avoir un éditeur de niveau pour créer sa topologie de carte, un simple éditeur de fichier bitmap suffit. Chaque point de couleur du fichier image est ainsi interprété comme une altitude lors du calcul de la carte.
Les données fournies par un champ de hauteurs sont assez générales pour reconstruire presque toutes les configurations de terrain voulues. Une solution alternative aurait été une approche que l'on pourrait nommer de *vectorielle* en opposition avec le bitmap (définition en tous points), c'est-à-dire qu'on aurait eu des objets avec une topologie spécifique, prédéfinis, qu'on aurait ensuite déplacés et mis à l'échelle sur une carte plane pour former la configuration finale. L'avantage d'une telle approche est le gain au niveau stockage et calcul. En effet la technique du champ de hauteurs nécessite une construction de triangulation pour

former l'objet en trois dimension final, alors que la technique *vectorielle* dispose d'objets pré-calculés sur lesquel il ne reste plus que quelques transformations à faire. Au niveau du stockage, la technique *vectorielle* constitue là aussi un avantage n'ayant besoin que du type d'objet introduit, sa position sur la carte, et quelques paramètres autres comme la taille ou une pente.
Seulement avec quelques efforts on peut transformer une carte décrite vectoriellement en un champ de hauteur alors que le contraire n'est pas faisable. Tout de même, il faut noter que certaines configurations de terrain ne sont pas possible à reproduire comme des pentes totalement verticales ou encores des grottes, ceci étant du au fait que le champ de hauteur ne contient qu'une seule information de hauteur par position dans le plan. Mais cette restriction n'est pas génante pour un terrain de base, de plus nous nous sommes donnés les moyens par la suite de rajouter de véritables objets en trois dimensions directement sur la carte.
L'objet champ de hauteurs est représenté dans le code par le module `HeightField` dans lequel on peut trouver les fonctions nécessaires à sa manipulation comme le chargement à partir d'un fichier bitmap, ou l'introduction d'objets *vectoriels* déformant l'objet initial, des objets comme des collines, des vallées, ou encore des plateaux ont été implémentés.

Les zones de la carte Une fois la topologie de la carte définie, il faut pouvoir décrire l'aspect de ses différents points (les textures). Nous voulions pouvoir introduire différentes zones qui seraient distinguées par des textures différentes, ainsi il serait possible d'avoir de l'herbe sous les pieds puis de la terre, puis des gravillons, rendant ainsi la carte moins monotone et permettant de délimiter à l'intérieur d'une même carte d'exercice plusieurs zones d'évolution.
Une des premières approches était la définition pour chaque case de jeu d'une texture spécifique, or cette technique a été un échec en ce qui concerne la vitesse de rendu, en effet pour chaque case de jeu le moteur OpenGl devait charger une texture (même si c'était la même que précédement utilisée) ce qui était problématique au niveau de la vitesse de rafraîchissement, on peut noter qu'on peut voir plusieurs centaines de case de jeux en même temps.
Nous nous sommes alors orientés vers une approche plus globale où nous décrivions réellement plusieurs zones avec pour chacune d'elle une seule texture. On peut alors décider de la précision des zones sachant que plus celle-ci sont petites, plus l'affichage sera lent.
La représentation des cartes se trouve dans le module `Map_look`, celui-ci ne dispose que de peu de fonctions pour l'instant, représentant principalement un type.
La triangulation se fait sur chaque aire de manière indépendante pour ne pas avoir de problème de raccord entre les textures.

La triangulation

Introduction La technique du champ de hauteurs nécessite un calcul important avant l'affichage, en effet on ne dispose à l'origine que de points isolés distribués régulièrement, mais l'affichage en trois dimensions d'OpenGl nécessite des triangles permettant de définir les surfaces de notre objet. Les données de départ sont organisées selon un maillage régulier comme montré en figure 4

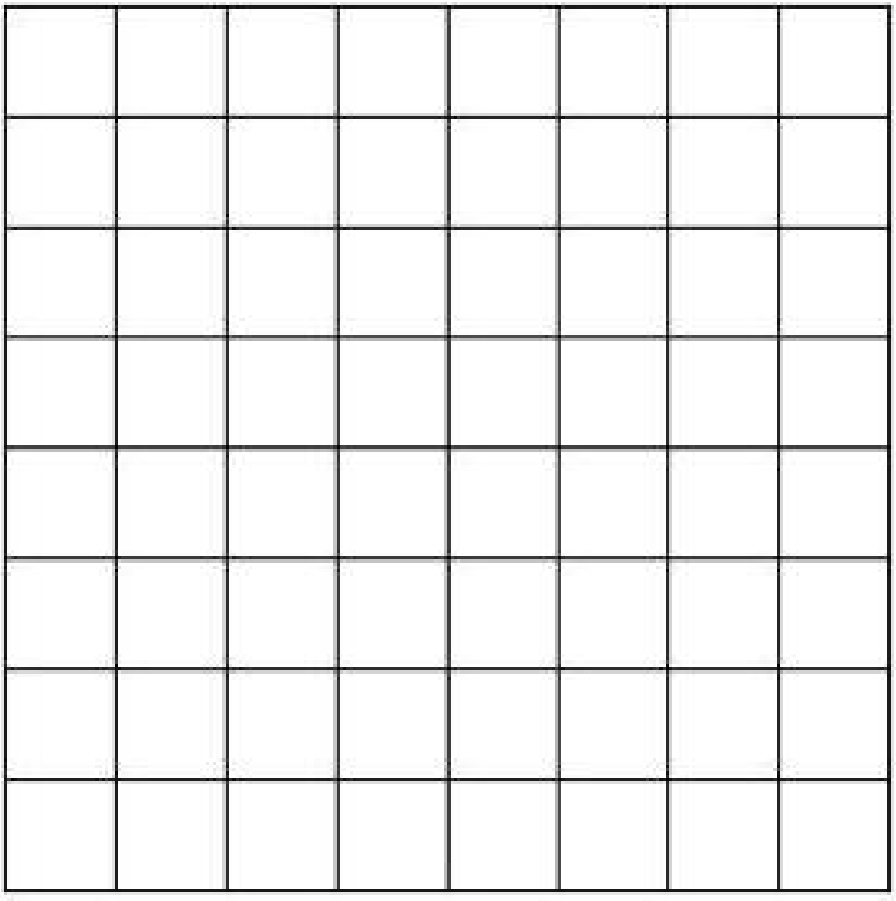


FIGURE 4 – Maillage d'origine

Triangulation simple La première approche a été de trianguler notre semis (ensemble de points) régulièrement, en faisant correspondre aux quatre sommets d'un petit carré du champ de hauteurs deux triangles comme montré dans la figure 5. Cette technique a pour avantage d'être simple et de fournir une des plus fidèles représentations de notre champ de hauteurs, de plus elle nécessite un temp de calcul en $O(n)$ pour une carte disposant de n points ce qui est très apréciable. On obtient le résultat de la figure 6. Seulement ne faisant

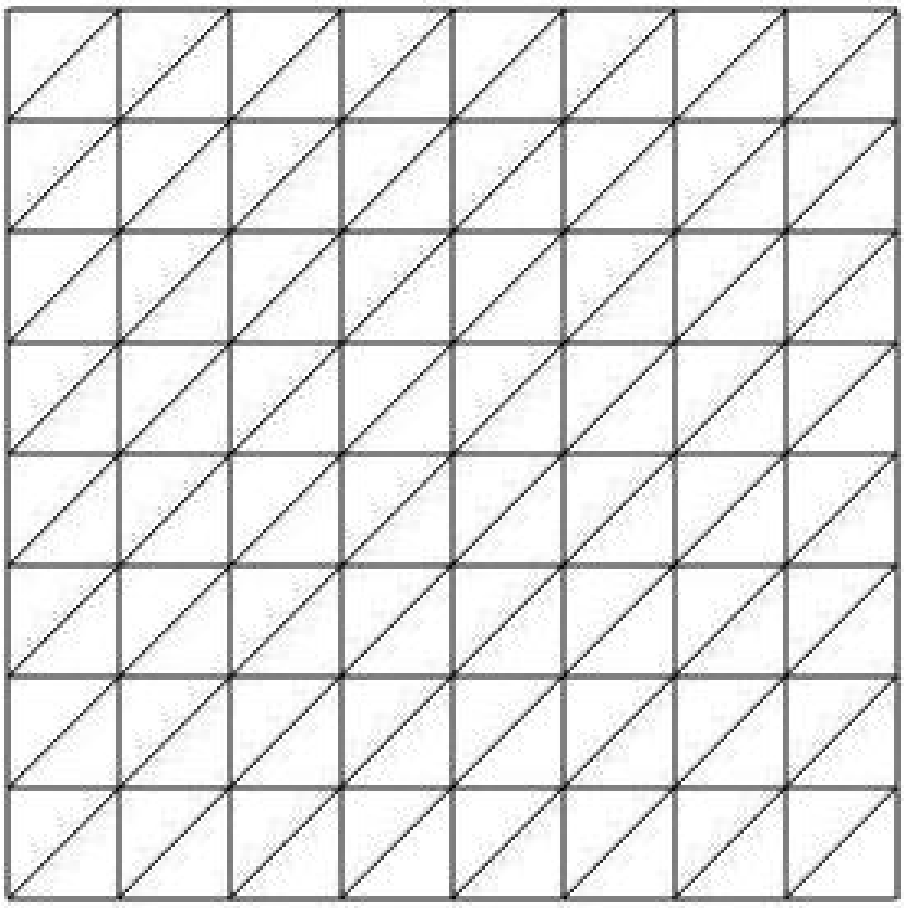


FIGURE 5 – Maillage d'origine

ce travail qu'une seule fois au chargement de l'exercice (nous n'avons pas choisi une solution dynamique qui est coûteuse en temps et peu nécessaire dans notre cas) nous pouvons nous permettre de calculer un peu plus de choses pour pouvoir afficher moins de triangles (accélérant ainsi l'affichage). Un des gros problèmes de la triangulation précédente était son résultat en nombre de triangles : $2 * (n - 1)$ ou n est le nombre de points du semis. On peut décrire une grande surface plane par seulement deux triangles, alors que l'algorithme précédent le divisera en une multitude de petits triangles comme dans la figure 7.

Triangulation optimisée Nous avons alors deux travaux à effectuer : premièrement il faut décider des points les plus importants, c'est-à-dire ceux sans lesquels l'approximation de la carte est fausse. Puis dans un deuxième temps il faut pouvoir créer des triangles entre ces différents points. Nous nous sommes donc appuyés sur l'article [**?**]

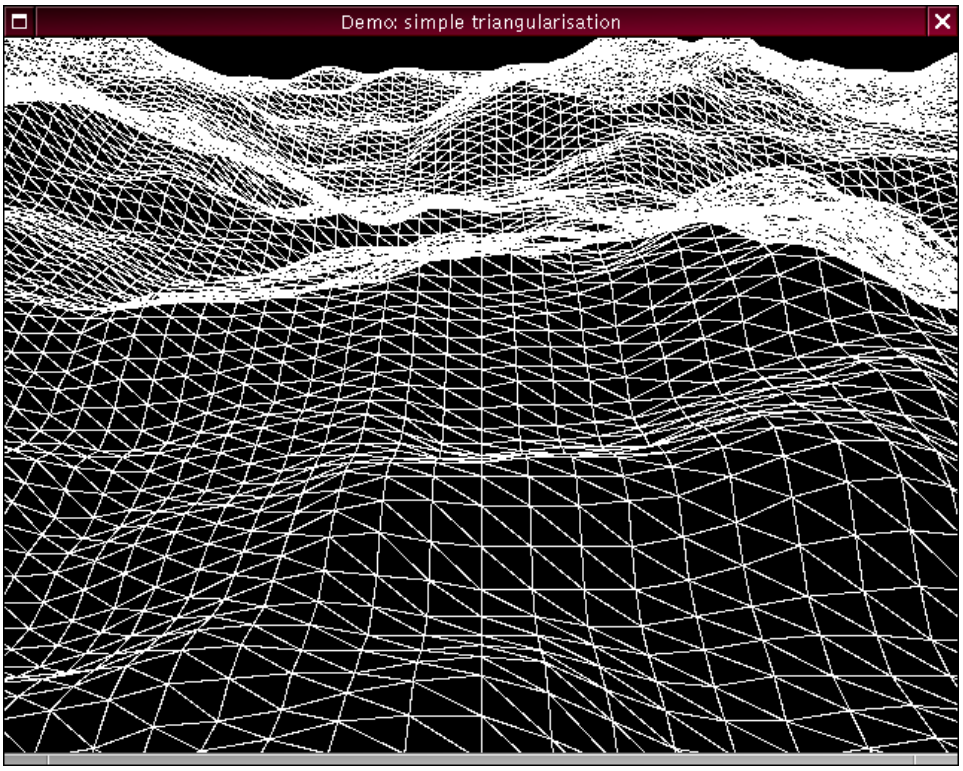


FIGURE 6 – Triangulation simple

qui donne quatre algorithmes d'approximation d'une carte définie par un champ de hauteurs. Ces algorithmes reposent sur la triangulation de Delaunay (qui s'effectue en deux dimensions), qui permet d'avoir un pavage qui maximise le minimum des angles de chaque triangle, ce qui permet d'avoir de beaux triangles (c'est-à-dire qu'ils ne sont pas trop allongés). De plus l'article donne les différentes possibilités pour le choix des points à prendre en compte. Nous allons par la suite expliquer notre implémentation qui se base sur le troisième algorithme proposé dans l'article.

Triangulation de Delaunay Notre premier travail était l'implémentation de la triangulation de Delaunay incrémentale, celle qui serait par la suite utilisée par les algorithmes. Ceci a été fait dans le module `Delaunay` de façon totalement indépendante du code de génération de carte (ainsi ce module peut être réutilisé pour d'autres applications). L'algorithme comme expliqué précédement est de nature incrémentale, on part d'une surface d'origine (un triangle) puis on ajoute des points, pour chacun on recalcule la triangulation pour que celle-ci respecte les conditions de Delaunay (qui peut s'exprimer par une cocircularité). Nous nous sommes concentrés sur une implémentation de l'algorithme local qui donne de très bons résultats en terme de complexité. Un point

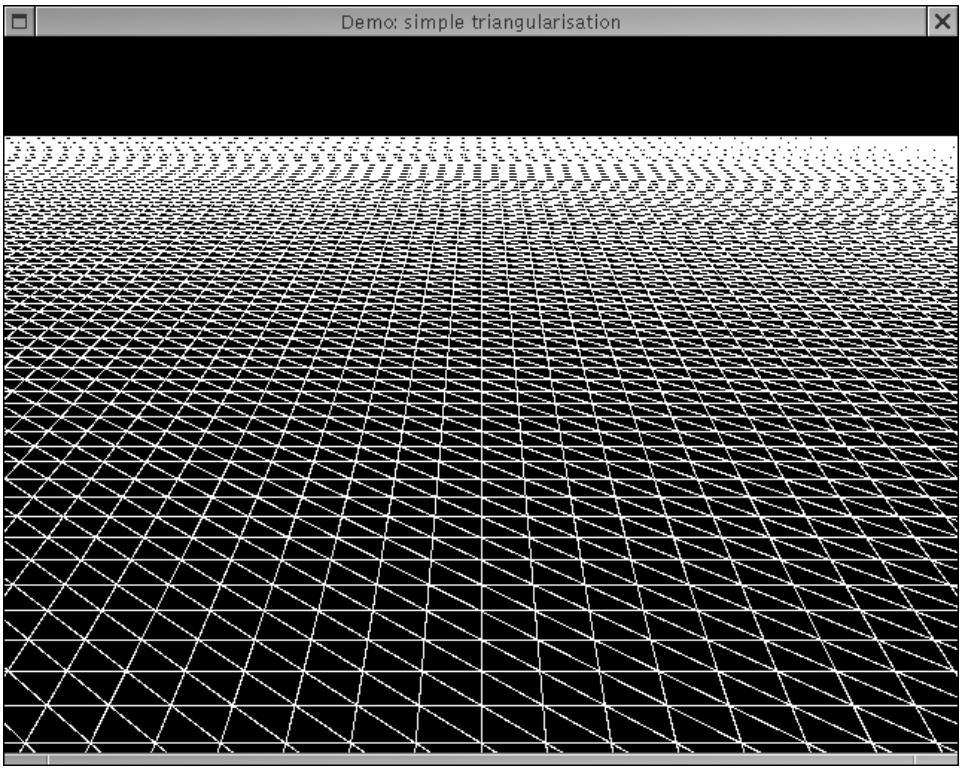


FIGURE 7 – Triangulation simple, grande surface plane

remarquable de l'implémentation est la possibilité d'associer à chaque triangle manipulé une structure d'un type quelconque. Cette structure est ensuite utilisée grâce aux fonctions données lors de la création du support d'origine. Ces fonctions concernent la création (appelée pour créer la structure associée au triangle lors de sa construction) et la destruction. Ce détail d'implémentation est très important pour avoir un algorithme final très performant.

Choix des points à insérer Le choix du point se base sur un calcul d'erreur local, pour chaque point on calcule l'erreur entre son altitude réelle et celle extrapolée de la triangulation actuelle, on ajoute alors le point qui maximise cette erreur. On peut alors arrêter l'algorithme selon au moins deux critères, le nombre de points ajoutés ou le passage de l'erreur maximale sous une certaine borne. Il s'agit ensuite de manipuler une structure de données qui permet un choix rapide du prochain point à ajouter ; comme conseillé dans l'article nous avons utilisé un tas.

Implémentation Au final donc d'après ce qui a été expliqué précédemment sur l'implémentation de la triangulation, nous devons définir une structure, deux fonctions sur celle-ci (une pour la créa-

tion d'un triangle et une pour sa destruction). La structure comporte trois champs, le premier donne l'erreur maximale calculée pour tous les points du triangles, le deuxième donne le point pour lequel cette erreur est atteinte, et le troisième un pointeur sur le triangle concerné :

```
structure: heap_ptr = {key: float;
                       candpos: fpos_t;
                       triangle : heap_ptr Delanay.triangle_t option}
```

Le type `fpos_t` est un type position flottante, et `'a Delaunay.triangle_t` est le type des triangles manipulés par le module `Delaunay`. La fonction de création récupère alors un pointeur sur le triangle qui vient d'être construit, calcule le point de ce triangle qui maximise l'erreur, range dans la structure les informations et dépose sur le tas cette même structure. Voici la fonction en pseudo-code (`H(p)` est la hauteur d'après le champ de hauteurs du point p) :

```
(heap_ptr Delaunay.triangle_t) funcc(triangle:heap_ptr Delaunay.triangle_t):
   plan = Trouver_plan_triangle(triangle)
   meilleurPoint:=(0,0)
   maxErr:=0
   PourTout point p dans triangle Faire:
      tmpErr:=|H(p) - interpoler(p,plan)|
      Si tmpErr>maxErr Alors
         maxErr:=tmpErr
         meilleurPoint:=p
      FinSi
   FinPourTout
   Si maxErr>0 Alors
      structure:= {key=maxErr; candpos=meilleurPoint; triangle=triangle}
      Deposer_sur_tas structure
      renvoyer structure
   Sinon
      renvoyer rien
   FinSi
```

La fonction de destruction se contente de récupérer la structure et de la détruire du tas.

```
rien funcd(triangle:heap_ptr Delaunay.triangle_t;structure: heap_ptr) :
   Si triangle!=rien Alors
      Retirer_du_tas structure
   FinSi
```

Il suffit alors de créer une surface de départ avec nos fonctions précédemment définies et d'itérer en prenant dans le tas le maximum des structures (selon le champ key), de l'ajouter à la triangulation en le déposant dans le triangle qui le contient (le champ `triangle` de la structure évite sa recherche dans tous les triangles).

```
pCandidat:=Retirer_max_du_tas()
TantQue But_non_atteind(pCandidat) Faire
   Inserer_dans_triangle(pCandidat.candpos, pCandidat.triangle)
   pCabdidat:=Retirer_max_du_tas
FinTantQue
```

La fonction `But_non_atteind()` se base dans notre programme sur un critère de qualité. Quant à la fonction `Inserer_dans_triangle()` elle est fournie par le module `Delaunay`.
On obtient alors une complexité globale pour le pire cas en $O(mn)$ ou n est le nombre total de points du champ de hauteurs sur lequel on calcule et m le nombre de points qu'on sélectionne pour l'approximation. On obtient pour le cas moyen une complexité en $O((m+n)logm)$.

Résultats Pour l'instant nous nous appuyons sur une approximation avec une erreur de 0, nous obtenons alors une carte fidèle tout en diminuant les triangles inutiles. Les figures 8 et 9 nous montrent les résultats sur les deux champs de hauteurs précédents.

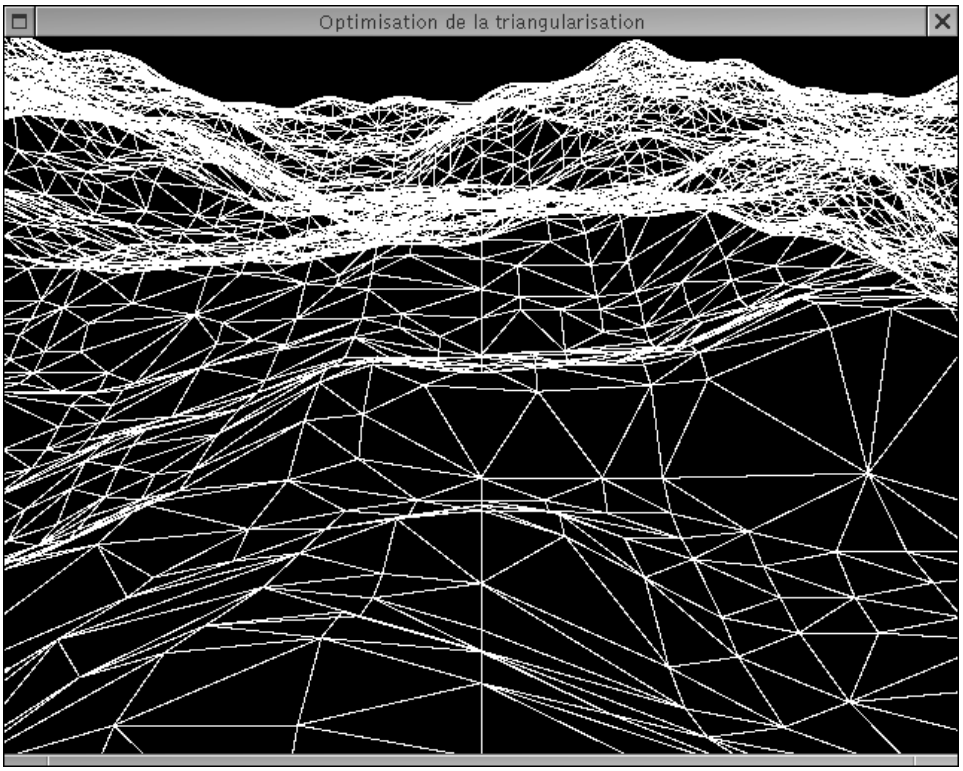


FIGURE 8 – Triangulation optimisée, paysage de la figure précédente

Problèmes Nous rencontrons quelques problèmes avec l'optimisation de carte contenant de brusques variations d'altitudes, les résultats sont souvent totalement incohérents (souvent dus à un triangle qui ne contient aucun point du champ autre que ces sommets car trop penché). De plus les rebords des zones ne correspondent parfois pas totalement, mais sans être absurde. Il faut donc faire attention aux cartes qu'on donne en exercice.

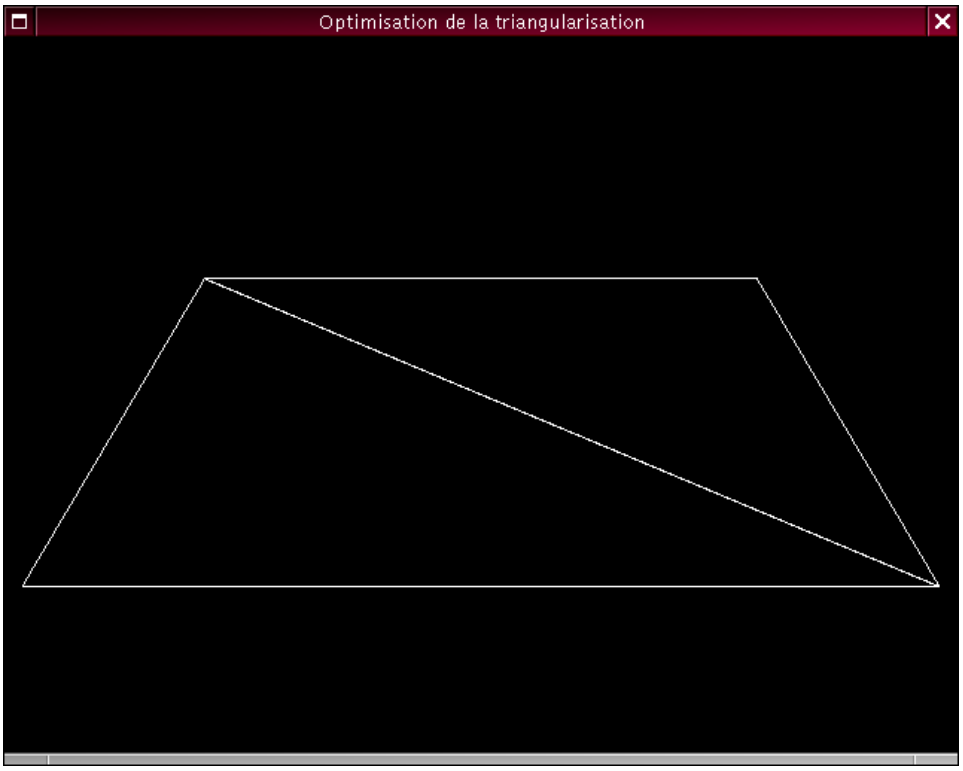


FIGURE 9 – Triangulation optimisée, grande surface plane

10.2.7 Textures

Une partie importante de l'environnement $3d$ est la gestion des textures. En effet, ce sont elles qui vont apportér du réalisme et des détails sans surcharger le moteur de gestion des sommets. On obtient de bien meilleurs résultats de performance et de qualité avec un modèle très simple et une texture de bonne qualité qu'avec un modèle coloré très complexe.

Chargement Pour l'instant nous avons développé une bibliothèque de chargement de texture supportant trois formats : BMP, PPM et JPG.
Pour cela nous utilisons dans le cas des JPG la librairie jpeg qui (miracle) est compatible Windows/Linux et pour les autres nous avons nous-même géré la gestion du chargement des données.
Nous n'allons tout de même pas utiliser des textures trop grosses ou de très haute résolution car on suppose que les stations de travail sur lesquelles vont travailler les enfants ne sont pas forcément dotées de carte $3d$ avec beaucoup de mémoire vive. Si la taille des textures excède cette mémoire, on va les retrouver dans la mémoire du pc, ce qui ferait effondrer les performances à cause du temps de chargement.

Utilisation Les textures offrent de très nombreuses possibilités de rendu. Mais dans notre cas nous devons nous contenter des utilisations les plus légères pour ne pas faire exploser le temps de calcul d'*OpenGl*. Nous n'utiliserons alors pas le système *Mipmap* de textures multirésolution qui permet un meilleur rendu de la texture suivant la distance. L'utilisation classique d'une texture consiste donc à charger un tableau de pixels dans un objet texture d'*OpenGl* puis de le paramètrer (répétition (true/false), gestion des couleurs (remplace/module/...)). Il ne nous reste plus qu'à le mettre sur le haut de la pile de textures quand on en a besoin pour texturer tel ou tel objet.

10.2.8 Options d'affichage

Suite au développemnt et aux différents tests effectués sur diverses configurations, il s'avère nécessaire d'offrir la possibilité de paramétrer l'interface *OpenGl*. Pour cela nous mettrons en place avec l'aide du projet interface graphique un setup de l'affichage.
Il devrait contenir différentes options :

— Niveau de détail des objets (Variation du paramètre lors de l'utilisation de Glm.Weld pour le chargement de l'objet).
— Distance du clipping (Variation du paramètre Far. Utilisé pour définir la matrice de projection du moteur).
— Utilisation du Fog (Booléen permettant l'affichage ou non du fog)
— Définitions des textures (Utilisations de différentes tailles de fichier pour les textures.)

11 Design

11.1 Description de cet axe de développement

Acteurs :
David Roger, Thomas Gazagnaire, Benjamin Leveque et Jean-Michel Ouvry (intervenant exterieur en BTS design)

Description :
Ce sous-projet a pour but la réalisation des objets 3D composant le monde virtuel.

Futur :
La modélisation d'objets 3d destinés à un moteur 3d temp réel est une chose beaucoup plus dure que la création de modèle pour le rendu d'une scène précalculé. Il reste quelques astuces à acquérir pour fournir au moteur 3d des objets à la fois beau et simple (au niveau du nombre de triangle). Beaucoup de problème ont été rencontré aussi lors des exportations de textures. Le robot actuel n'est pas définitif et sera surement amené à changer.

11.2 Détails techniques

Après une demande du sous-projet 3D, les objets sont tout d'abord dessinés sur papier, sous différents angles de vue. Ils sont ensuite modélisés en 3D à l'aide d'un logiciel de modélisation. Puis ils sont mis à la disposition de l'équipe 3D sous forme de fichier .obj (objet 3D) et .mtl (texture).

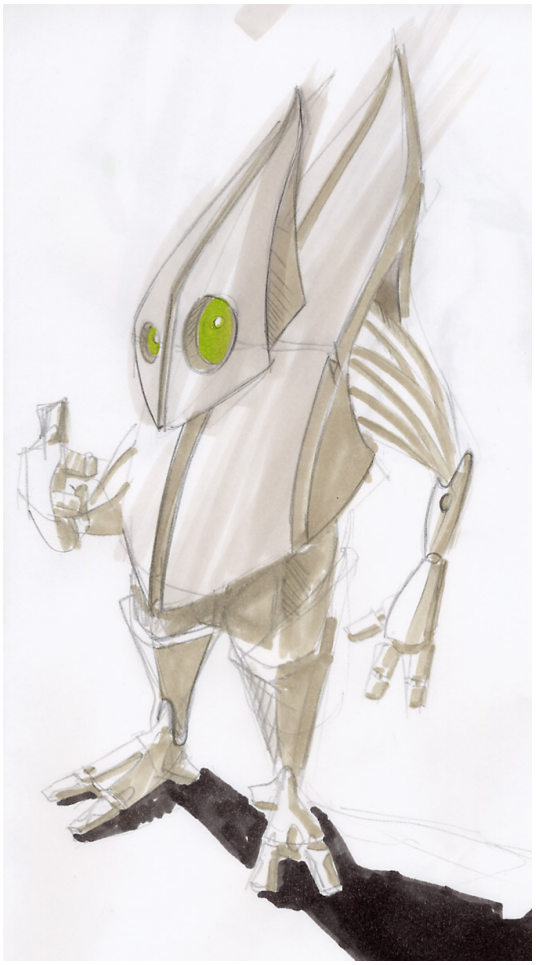
11.2.1 Tests

Le travail des membres de ce sous-projet commence tout d'abord par la réalisation rapide d'un objet 3D simple comportant des textures. Cet objet permet à l'équipe 3D de tester rapidement la compatibilité des formats de fichiers et les possibilités graphiques offertes par les différents langages de programmation Java, C, Caml.

11.2.2 Dessins 2D

Dans le même temps, les artistes du sous-projet ont commencé à dessiner sur papier, en 2D, certains des objets qui apparaîtront dans le monde virtuel. Les directives principales étaient la réalisation d'un monde paisible, doux et coloré. Une réflexion et un soin particulier devait être apportés à la réalisation du robot, objet de toutes les attentions. Ca été le premier objet qui a été réalisé.

FIGURE 10 – Le robot dessiné



11.2.3 Modélisations 3D

Une fois dessinés sur papier, sous différents points de vue, les objets ont été modélisés en 3D sur ordinateur. C'est probablement l'étape la plus difficile dans la chaîne de production d'objets 3D, et aussi celle où les membres du sous-projet possèdent le moins d'expérience. Un apprentissage important devait donc être réalisé pour acquérir les bases nécessaires à l'utilisation d'un logiciel de modélisation 3D. Les aides et tutoriaux disponibles sur la toile ont été une aide précieuse pour ce travail long et fastidieux.

FIGURE 11 – Le robot modélisé

12 Interface graphique

12.1 Description de cet axe de développement

Acteurs :
Mathieu Martin

Futur :
Seulement une interface graphique de base et de débuggage a été développé, il est indispensable qu'une interface solide soit développé dans les plus bref délai.

Description :
L'interface graphique de débuggage a été développé en lablTk et implémente le minimum néscessaire pour l'utilisation du programme.

13 Réseau

13.1 Description de cet axe de développement

Acteurs :
Samuel Thibault, Damien Pous, Nicolas Bernard

Description :
Ce sous-projet s'occupe de l'implémentation de la gestion réseau dans **RigobOt**.

13.2 Détails techniques

13.2.1 Introduction

Afin d'apporter encore un peu plus d'attrait à **RigobOt**, une certaine gestion du réseau est faite. En effet, la plupart des écoles ont un réseau local assez rapide que l'on peut mettre à profit pour que les élèves puissent non seulement travailler sur leur propre machine, mais aussi échanger leurs programmes, idées, et organiser des championnats pour favoriser l'émulation entre élèves.

13.2.2 Architecture générale

Les réseaux locaux tels qu'on peut les trouver dans des écoles sont en général très simple : une dizaine de machines connectées par câble coaxial, par exemple. Sur ce genre de réseau, le broadcast est possible, et permet donc de «voir» qui est disponible sur le réseau. Ensuite, il faut établir un lien sûr pour pouvoir transférer les données nécessaires.

C'est pourquoi on utilise une architecture s'appuyant à la fois sur UDP et sur TCP :

— une socket UDP reste en permanence à l'écoute de paquets qui contiennent le nom de la machine qui l'a envoyé. Il suffit donc d'ajouter à la liste des machines disponibles cette machine, accompagnée de l'adresse source du paquet, qui servira pour les connexions ultérieures. Si cette machine n'était pas encore connue, on en profite pour lui envoyer un tel paquet, car elle ne nous connaissait sûrement pas. On ne le fait que si la machine n'était pas encore connue, afin d'éviter tout effet «ping-pong». Cependant, si le paquet est un paquet d'initialisation (le premier charactère du paquet l'indique), on y répond dans tous les cas, car cela veut dire que l'autre machine vient de redémarrer **RigobOt** ;

— une socket TCP à l'écoute des connexions entrantes : ce peut être pour récupérer la liste des partages disponibles : «disquettes» (programmes donnés aux robots), ou «match» (ses paramètres, ce qui permet de le lancer ici aussi, puisque notre monde est déterministe).

À l'initialisation, on envoie donc un paquet UDP à toutes les machines en utilisant l'adresse ip spéciale 255.255.255.255, ce qui permet à la fois de se faire connaître, et de recevoir en retour les noms de toutes les machines disponibles.

Ensuite, les communications sur TCP se font avec un protocole simple, au format texte pour ce qui est des commandes (les sous-projet récupèrent ensuite les `in_channel` et `out_channel` pour échanger leurs propres données.). Le client envoie une commande, et le serveur envoie une réponse : (liste susceptible d'être étendue)

— `LIST` permet de récupérer la liste des noms des partages avec leur type, un nom de partage par ligne ;
— `GET partage` permet de récupérer le partage nommé `partage`.

13.2.3 Au niveau utilisateur

On obtient ainsi à l'écran une liste des machines présentes sur le réseau local. On ajoute également la possibilité de contacter une machine qui n'aurait pas été jointe automatiquement : si elle est sur un autre réseau avec un routeur entre les deux (les routeurs ne propagent pas les broadcasts).

On peut alors cliquer sur l'une des machines pour en récupérer la liste des partages. On peut ensuite cliquer sur un fichier pour le télécharger.

13.2.4 Détails de l'implémentation

Pour relier le réseau à la fois à l'interface et au monde abstrait, on utilise un objet «network» qui contient une série de méthodes pour communiquer avec eux :

— `idle` regarde à la fois la socket UDP et la socket TCP. Elle effectue la mise à jour de la liste des machines en lisant UDP, puis répond à une éventuelle connection TCP ;
— `hosts` renvoie la liste des machines, sous forme de paires (nom, adresse)
— `announce_to` et `announce_to_host` permettent d'envoyer un paquet UDP à la machine précisée, pour implémenter la fonction de «recherche de machine» ;
— `files` demande à la machine précisée la liste de ses partages, qui est renvoyée sous forme d'une liste de `string` ;
— `file` demande à la machine précisée le fichier précisé ; une fois la requête transmise, le `in_channel` est renvoyé à l'appelant, pour qu'il traite lui-même le format du partage. Ce sera à lui de le refermer lorsqu'il a lu ce qu'il voulait, ou que `End_of_file` a été levé ;

Pour gérer l'ensemble des fichiers, on utilise une classe `container`, très simple, qui se limite aux méthodes :

— `files` qui renvoie la liste des noms des fichiers contenus,
— `load` qui renvoie le fichier dont le nom est donné
— `save` qui sauvegarde le fichier dont le nom est donné, écrasant éventuellement une ancienne version.
— `write` qui écrit le fichier dont le nom est donné sur le canal donné.
— `read` qui lit un fichier depuis le canal, et l'enregistre sous le nom donné.

Afin d'implémenter `write` et `read`, il faut donner des fonctions de lecture et d'écriture depuis et vers un canal lors de la création de la

classe. Au passage, cela indique le type effectif des fichiers manipulés. Pour l'instant, la sauvegarde est faite en mémoire, mais l'on pourra les sauvegarder sur le disque dur à l'aide de `read` et `write`.

C'est ainsi une instance de cette classe que l'on donne en paramètre à la création de l'objet network.

Une fois l'interface et le réseau créés, on peut alors enregistrer le réseau auprès de l'interface à l'aide d'une méthode `register_network`. C'est à ce moment-là que les éléments de l'interface pour le réseau peuvent être créés, avec les fonctions évènements correctement associées aux boutons.

On enregistre également un timer qui appelle `idle` et met à jour la liste des machines sur l'interface. On le programme toutes les secondes par exemple. Cela permet d'éviter de surcharger la machine uniquement pour la gestion du réseau, mais d'avoir tout de même une bonne réponse.

13.3 Sécurisation

Il est important de pouvoir protéger les utilisateurs contre d'éventuelles interférences extérieures qui peuvent exister si le réseau local de l'école est relié à Internet. Nous avons pour ce faire implémenté un mécanisme basique d'authentification, qui est, l'installation passée, totallement transparent pour l'utilisateur. Un autre usage, secondaire sur le plan de la sécurité mais d'un usage plus immédiat pour les enseignants, est de permettre l'utilisation par plusieurs classes de Rigobot simultanément sur un même réseau local sans interférences (par exemple les machines d'une classe apparaissant dans la fenetre réseau de l'autre) si celles-ci ne sont pas désirées.

13.3.1 Principe

Lors du déploiement de l'application sur les machines, l'administrateur (*i.e.* : l'enseignant) entre un mot de passe qui doit être partagé par toutes les machines devant pouvoir jouer ensemble. Ce mot de passe est stocké (sous forme d'empreinte, à la manière d'Unix) dans un fichier. Ensuite, dans la phase de découverte du réseau en UDP expliquée précédemment, les ordinateurs incluent un champ contenant un MAC (Message Authentification Code) pour lequel l'empreinte du mot de passe sert de secret partagé.

13.3.2 Notes sur l'implémentation

L'implémentation de ce schéma de sécurité en est actuellement au stade alpha, aussi son utilisation n'est pas activée par défaut. Pour compiler une version de Rigobot l'incluant, il faut d'abord positionner à **true** la variable **enable_security** dans le fichier *net/network.ml*.

Sans entrer dans les détails de l'implémentation (qui se trouve, outre son intégration dans *net/network.ml*, dans *net/security.ml*) indiquons que nous avons une fonction chargée de signer un message et une autre chargée de vérifier une signature sachant de quel hote il est censé provenir. Si la signature se révèle correcte, le message reprend alors un trajet normal, sinon on le jette.

D'autre part les définitions de, par exemple, la fonction de hachage (qui est pour l'instant celle du module **Digest** de Caml (MD5)), sont faites au début du fichier de manière à pouvoir en changer facilement, ce qui est toujours important en sécurité pour pouvoir réagir rapidement lors de l'éventuelle découverte d'une faille dans un algorithme...

13.3.3 Limites et prospective

Une des limites du schéma actuel est d'être vulnérable au rejeu de paquets et au vol de connexion, l'authentification se faisant seulement lors du contact initial. Dans un futur proche, l'ajout respectivement d'un système de *défis* et la généralisation de l'ajout d'un MAC à tous les messages transmis permettra d'éviter ces écueils.

Par la suite, il est imaginable d'utiliser la bibliothèque **Cryptokit** de Xavier Leroy, dont Mike Lin a récemment fait un port pour Windows, pour remplacer l'utilisation de l'empreinte du mot de passe comme secret partagé par une clef générée à chaque session selon le mécanisme Diffie-Hellman...

14 Conclusion

Le logiciel **Rigobot** à ce point de l'implémentation est opérationnel mais il lui manque une interface graphique pratique et attrayante de plus les animations ne sont pas encore très belles. C'est encore donc une version de test.
Le projet s'est très bien déroulé laissant à chacun la possibilité de s'exprimer dans le domaine qui lui plaisait et nous faisant profiter de l'expérience du développement en grand nombre.
Rigobot devrait continuer sa route avec les membres du projet qui le désire et peut-être même incorporer de nouveaux développeurs. Tout ceci en espérant que notre logiciel pourra servir dans l'enseignement.

A Inférence de type

A.1 Problématique

Afin de faciliter l'écriture du code par l'enfant, nous avons opté pour un langage fortement typé, c'est-à-dire, où les types des variables ou fonctions sont inférés de manière automatique.

Ainsi, lors de la déclaration d'une fonction, il suffit d'écrire

```
soit f a b = a + b
```

et le système en déduit que f est une fonction de $\mathbb{N}^2 \to \mathbb{N}$. Sans le typage, l'élève devrait donner lui-même cette information et il devrait écrire :

```
soit f: int (a: int) (b: int) = a + b
```

Il en va de même pour les variables : elles doivent être déclarées (et initialisées) par l'élève, mais ce dernier n'a pas besoin d'en donner le type.

Le typage permet de plus d'obtenir un système d'évaluation robuste. Si une expression est bien typée, son exécution ne posera pas de problème... Par exemple, si l'élève tape

```
soit a = ‘‘toto’’ + 51
```

L'erreur sera détectée lors du typage, (et sera expliquée clairement à l'élève), évitant ainsi l'erreur qui serait survenue à l'évaluation.

Au niveau de la structure du code, le typeur permet de transformer un arbre fourni par le parseur (de type p_unit_expr) en un arbre bien typé, évaluable (de type unit_expr).

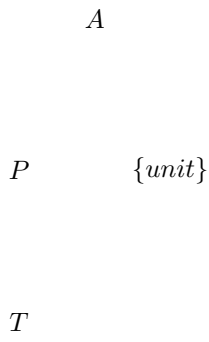
A.2 Définition du système de types

On se donne un ensemble fini de types de base : $T = \{int, bool, string, ...\}$, deux ensembles infinis de variables de type : $P = \{'a, 'b, ...\}$ et $A = \{“a, “b, ...\}$, et un type particulier : *unit*.

En caml, cela donne

```
type type_t =
| T_int | T_bool | ... (* T *)
| T_unit
| T_Poly of int        (* P *)
| T_Any  of int        (* A *)
```

Avec les mains, on définit un système de types respectant la relation notée $\preceq$ suivante :



Un type de A (Any) peut devenir n'importe quoi, un type de P (Poly) ne peut pas devenir *unit*. En effet, on souhaite que l'élève puisse définir des *procédures* ie. des fonctions qui ne renvoient rien, c'est à dire la valeur de type *unit*, mais on ne veut pas qu'il puisse manipuler cette valeur (l'affecter, la passer en argument d'une fonction...), d'où la nécessité de bien séparer ce type des autres types de base.

Afin de ne pas perturber l'enfant, le système doit accepter le polymorphisme : si l'enfant définit f par `soit f x = x` en pensant très fort que x est une chaîne de caractères, le typeur ne doit pas dire "je ne sais pas comment typer x, disons que c'est un entier...". il doit donc typer cette fonction $'a \to' a$. (comme x est un argument, son type doit être restreint à P).

Les règles d'inférence sont standards, je ne m'étendrai donc pas dessus.

A.3 Implémentation

L'implémentation du typeur commence par la définition d'un objet *environnement de typage*, permettant lors du parcours de l'arbre à typer, d'accéder aux symboles définis, ainsi qu'à leurs types. Cet environnement est défini dans le fichier `language/environment.ml`.

La seconde partie consiste à parcourir l'arbre, en tenant l'environnement à jour, en unifiant les types au fur et à mesure, et en reconstruisant l'arbre évaluable correspondant. Ce code est écrit dans le fichier `language/typer.ml`

A.3.1 Environnement de typage

Symboles, blocs. En un point d'une expression, trois choses nous intéressent : l'ensemble des symboles définis, leurs types inférés, et la pile des blocs du code qui contiennent ce point. Les deux premières sont stockées dans une table de hachage qui, à un symbole, associe son

type, [3] et la troisième est stockée dans une pile de listes de symboles.

Lorsque l'on entre dans un nouveau bloc, on doit empiler une liste de définitions vide (aucun symbole n'est pour l'instant défini dans ce bloc). Au contraire, lorsque l'on sort d'un bloc, il faut retirer de la table des symboles les symboles de la liste se trouvant au sommet de la pile, et dépiler celle-ci (les symboles correspondant n'étant plus définis). Lorsqu'un nouveau symbole est défini, on doit l'ajouter à la table des symboles, ET à la liste du haut de la pile (le bloc courant).

Unification, alias. Lors du parcours de l'arbre, des types vont devoir être unifiés. C'est à dire qu'il va falloir calculer des classes d'équivalence sur les types. Ces classes d'équivalence seront représentées par leur plus petit élément : si un type de base fait partie de la classe c'est lui, sinon, c'est le type de P puis A ayant le plus petit indice.

On maintiendra donc deux tables de hachage, maintenant pour chaque variable polymorphe définie, le plus petit élément de sa classe d'équivalence (*alias*). Si, lors d'un accès à une de ces tables, on tombe à nouveau sur un type polymorphe, on recherche son éventuel alias et on met à jour la table en conséquence. Ainsi, lors de la fusion de deux classes on peut se contenter de faire pointer l'alias de l'une sur celui de l'autre.

Lorsque l'on doit unifier deux types t et t', on regarde donc leurs alias :

— Soit ce sont deux types de base ($T \cup \{unit\}$), il suffit alors de vérifier qu'ils sont égaux, et de lever une exception le cas échéant.

— Soit l'un d'eux est polymorphe ($A \cup P$), il faut alors le rajouter dans la classe de l'autre, (sauf dans le cas $t \in P, t' = unit$ où l'on lève une exception) en rajoutant l'entrée correpondante dans la table de hachage adéquate.

Clonage, normalisation. Des fonctions éventuellement polymorphes peuvent être définies dans l'environnement. Par exemple, $f\,'a \to' a$. Si, lors du parcours de l'arbre, le typeur demande le type de f, on ne peut pas lui fournir tel quel : sur l'entrée `f 3, f ‘‘abc’’`, la variable $'a$ devra être unifiée à *int*, puis *string*, ce qui est impossible !

On doit donc assurer que lorsque l'on renvoie le type d'une fonction, celui-ci ne contient que des variables de type 'fraîches', ie n'ayant jamais été utilisées.

Pour ce faire, on utilise un compteur, indiquant le plus grand indice utilisé. On peut donc créer de nouvelles variables ou cloner des signatures de fonctions selon ce compteur. Cependant, afin de ne pas risquer d'atteindre MAXINT (on est pessimistes :), les signatures de fonction sont stockées sous forme *normalisée* : les indices des variables de type sont compris entre 1 et le nombre de variables type apparaissant. Lorsque l'on a fini de typer une expression, on peut donc vider en sécurité les tables d'alias, et redescendre le compteur.

3. un symbole pouvant avoir plusieurs valeurs associées, c'est toujours la dernière définition que l'on considère

A.3.2 Typage

Instructions, expressions. Le typage est effectué en parcourant l'arbre fourni par le parser, et en unifiant les types au fur et à mesure. On écrit donc deux fonctions mutuellement récursives `type_unit_expr` et `type_expr`, qui filtrent respectivement les *instructions* et les *expressions*. `type_expr` prend comme argument supplémentaire le type présumé de l'expression à typer.

— Lorsque l'on rencontre une définition de variable, `soit v = e`, on commence par typer e en partant d'un type polymorphe *frais* t, puis on rajoute la définition (v, t) à l'environnement
— Lorsque l'on a une application de fonction, on récupère un clone de la signature de celle-ci dans l'environnement, puis on type récursivement chacun des arguments, et on unifie le type de retour, avec le type passé en argument.
— Lorsque l'on a affaire à du contrôle de flot (if, while...), il faut entrer dans un nouveau bloc, avant de typer la sous-expression.
— Les autres cas sont relativement compréhensibles, la lecture du code devrait suffire...

La difficulté majeure de ce parcours consiste à assurer qu'à chaque entrée dans un nouveau bloc correspond une sortie, notamment lorsque des exceptions sont levées dans un appel récursif !

Fonctions, procédures. On l'a dit, les procédures sont des fonctions qui renvoient *unit*. Il nous suffit donc de savoir typer une fonction.

Les fonctions sont typées *par groupe* afin d'autoriser la récursion mutuelle :

```
soit pair n = si n<2 n==0 sinon impair (n-1)
et impair n = si n<2 n==1 sinon pair (n-1)
```

En effet, ces deux fonctions doivent être typées *ensemble*, sans quoi le symbole *impair* ne sera pas défini au niveau de *pair*.

Pour typer un ensemble de fonctions, on commence par les déclarer dans l'environnement, avec les types les plus généraux possibles : $'a_1 \to ... \to' a_n \to$ "a (seul le type du résultat peut-être unifié à *unit*).

Puis, on type chacune d'entre elles, en prenant soin de déclarer dans l'environnement les symboles correspondant à leurs arguments, typés selon les signatures précédentes ($'a_0, ...,' a_n$)

Lorsque toutes les fonctions ont bien été typées, leurs signatures sont normalisées et stockées dans l'environnement, en écrasant les anciennes.

B Structures syntaxiques

Lors l'analyse du code, le parser produit un arbre syntaxique qui est transformé par le typeur.

B.1 Implémentation : l'arbre de retour du parser

Le code de l'élève est transformé par le parseur en un objet de type `p_unit_expr_t` défini par :

```
type p_expr_t =
  | P_Value      of value_t * p_info_t
  | P_FunApp     of id_t * p_expr_t list * p_info_t
  | P_IfThenElse of p_expr_t * p_expr_t * p_expr_t * p_info_t
  | P_Var        of id_t * p_info_t
  | P_TSeq       of p_unit_expr_t * p_expr_t * p_info_t
and p_unit_expr_t =
  | P_Nothing
  | P_Action     of action_t * p_info_t
  | P_Expr       of p_expr_t * p_info_t
  | P_VarDef     of id_t * p_expr_t * p_info_t
  | P_VarSet     of id_t * p_expr_t * p_info_t
  | P_FunDef     of p_fun_def_t * p_info_t
  | P_Seq        of p_unit_expr_t * p_unit_expr_t * p_info_t
  | P_Repeat     of p_unit_expr_t * p_info_t
  | P_RepeatExpr of p_expr_t * p_unit_expr_t * p_info_t
  | P_While      of p_expr_t * p_unit_expr_t * p_info_t
and p_fun_def_t = id_t * p_expr_t * id_t list
```

Globalement, si on revient aux considérations du 8.2.3, le type `p_unit_expr_t` représente les expressions *expected unit* et le type `p_expr_t` représente les expressions *expected non unit*.

La valeur de type `p_info_t` dans chaque constructeur est utilisée par le parser pour transmettre des informations sans caractère syntaxique (numéros de ligne, intervalle de caractères, ...), on n'en tiendra pas compte dans la suite.

B.1.1 Le type `p_expr_t`

L'expression représentée par un objet de type `p_expr_t` correspond à l'un des constructeurs suivants.

- `P_Value` C'est une constante, de type `value_t` défini par :

  ```
  type value_t =
    | V_Int    of int     (* entiers  *)
    | V_Bool   of bool    (* booléens *)
    | V_String of string  (* chaînes  *)
    | V_Class  of string  (* classes d'objets   *)
    | V_Object of int     (* instances d'objets *)
    | V_Unit              (* laseule valeur de type unit *)
  ```

- `P_FunApp`
  C'est l'application d'une fonction (ou d'une procédure) à la liste (non vide) de ses arguments. Le nom de la fonction est une chaîne (`type id_t = string`). Ses arguments sont des expressions *expected non unit*.
- `P_IfThenElse`
  C'est un branchement conditionnel, dont la condition comme les branches sont de type `p_expr_t`.
- `P_Var`
  C'est la donnée d'un seul nom de variable : le parser n'a pas accès à l'environnement d'évaluation, et ne peut donc déterminer s'il s'agit là d'une variable normale, ou de l'appel à une fonction (ou procédure) sans arguments.
- `P_TSeq`
  C'est une expression précédée d'instructions de type `p_unit_expr_t`.

B.1.2 Le type `p_unit_expr_t`

L'expression représentée par un objet de type `p_unit_expr_t` correspond à l'un des constructeurs suivants.

- `P_Nothing`
  L'expression est vide : elle n'a aucun effet.
- `P_Action`
  C'est une action du robot, de type `action_t` défini par :

```
type action_t =
  | Nop
  | Step
  | Rotate_Left
  | Rotate_Right
  | UTurn
  | Take
  | Put
  | Act
  | ...
```

- `P_VarDef`, `P_VarSet`
  C'est une affectation de variable.
- `P_FunDef`
  C'est une définition de fonction ou de procédure, suivant le type de la composante `p_expr_t`.
- `P_Seq`
  C'est la mise en séquence de deux instructions de type `p_unit_expr_t`.
- `P_Repeat`, `P_RepeatExpr`, `P_While`
  C'est une boucle infinie, itérative ou conditionnelle. Dans `P_RepeatExpr` (boucle itérative), la composante de type `p_expr_t` doit s'évaluer en un entier qui donne le nombre d'évaluations. Dans `P_While` (boucle conditionnelle), elle doit s'évaluer en un booléen qui conditionne la répétition de la boucle.

B.2 Implémentation : l'arbre après typage

L'arbre est sensiblement le même : les composantes de type `p_info_t` ont disparu et quelques transformations ont eu lieu, mais surtout on est assuré que les divers sous-arbres de type `unit_expr_t`. s'évalueront en des valeurs dont les types sont compatibles avec la structure de l'arbre ainsi qu'avec les signatures des fonctions et procédures présentes.

Parmi les transformations effectuées, on peut en particulier citer la transformation de `P_Var` en `FunApp`, lorsque le nom de variable concerné est celui d'une fonction sans arguments.

On obtient les types Caml suivants :

```
type expr_t =
  | Value      of value_t
  | FunApp     of id_t * expr_t list
  | IfThenElse of expr_t * expr_t * expr_t
  | Var        of id_t
  | TSeq       of unit_expr_t * expr_t
  | TLeave_    of expr_t    (* uniquement pour l'évaluation *)
and unit_expr_t =
  | Nothing
  | Action     of action_t
  | Expr       of expr_t
  | VarDef     of id_t * expr_t
  | VarSet     of id_t * expr_t
  | FunDef     of fun_def_t
  | Seq        of unit_expr_t * unit_expr_t
  | Repeat     of unit_expr_t
  | RepeatN    of int * unit_expr_t
  | RepeatExpr of expr_t * unit_expr_t
  | While      of expr_t * unit_expr_t
  | Leave_               (* uniquement pour l'évaluation *)
and fun_def_t = id_t * expr_t * id_t list
```

C Évaluation du code

C.1 Problématique

À chaque top du monde abstrait, le robot de l'élève doit agir selon le code entré par ce dernier. Il faut donc pouvoir passer de *l'arbre évaluable* fourni par le typeur à un flot d'actions, exécutées par le robot.

Ce flot d'actions dépend non seulement du code entré par l'élève, mais aussi des évènements générés dans le monde à chaque instant...

De manière plus formelle, on doit :

— pouvoir rechercher la première action apparaissant dans un arbre évaluable, et calculer le résidu de ce dernier. Par exemple, pour le code suivant,
```
soit f x = repete x fois saute, avance fin.
soit c = 50 + 1, f c
```
l'évaluateur doit, en une étape, calculer $50 + 1$, le stocker dans c, appliquer f sur c, commencer à dérouler la boucle *repete*, tomber sur la première action (*saute*) et la renvoyer, accompagnée de *ce qu'il reste à faire* :
```
avance, repete 4 fois saute, avance fin.
```
— gérer une liste de gestionnaires d'évènements, ainsi qu'une pile d'appels, et empiler les gestionnaires, lorsque les évènements correspondant sont générés.

Le code s'occupant de tout ça est dans `language/prgm.ml`, on y définit un objet *prgm* que l'on insère ensuite dans le robot, un peu à la manière d'une disquette.

C.2 Implémentation

Comme pour le typage, on se sert d'un *environnement d'évaluation*, permettant d'accéder aux valeurs des variables et aux corps des fonctions.

C.2.1 Environnement d'évaluation

La structure initiale de cet objet (table de symboles, pile de blocs) étant la même que celle de l'environnement de typage, ils héritent tous deux d'une classe générique (définie dans `environment.ml`).

La différence entre ces deux objets provient du fait que l'on associe ici non plus des types mais des valeurs aux symboles : à une variable est associée sa valeur, à une fonction, son corps (et les noms de ses arguments)

C.2.2 Évaluation d'une expression

Ici encore, la structure est similaire à celle du typage : on définit deux fonctions mutuellement récursives `eval_expr` et `eval_unit_expr`, dont les types sont les suivants :

— `eval_expr: eval_env -> eval_expr_t -> eval_t` où l'on a

```
type eval_t =
| Evaluated      of value_t
| Action_Reached of action_t * expr_t
```
Soit on a pu calculer la valeur de l'expression, on la renvoie ; soit on est tombé sur une action avant, et on renvoie le couple formé de l'action et de ce qu'il reste à faire pour calculer la valeur de l'expression.

— `eval_unit_expr: eval_env -> eval_expr_t -> action_t * unit_expr_t`
On renvoie la première action rencontrée, et ce qu'il reste à faire.

Le gros problème réside ici aussi au niveau de l'ouverture et de la fermeture des blocs de définition, d'autant plus qu'il arrive très fréquemment qu'un bloc soit ouvert lors d'une évaluation pendant un top, mais refermé seulement plusieurs tops plus tard, lors d'un second appel à l'évaluateur... Par exemple, pour

`repete soit a=3, avance, saute, avance, saute, p_int a, fin.`

On est donc obligé de stocker dans l'arbre une information indiquant le nombre de blocs ayant été ouverts. Pour cela, on utilise une feuille spéciale dans les arbres d'instructions (`unit_expr_t`) : `Leave_`. Lorsque cette feuille est évaluée, on fait une sortie de bloc. On fait de même pour les arbres d'expressions. (`expr_t`)

Mis à part les point sus-mentionnés, le parcours est relativement intuitif. Je précise tout de même deux points : l'application de fonction et le déroulement des boucles.

Application de fonction Lors de l'application d'une fonction, il faut commencer par évaluer ses arguments. Ceux-ci sont évalués de droite à gauche. Si une action est rencontrée durant cette série d'évaluations, le résidu est l'application de la fonction à la liste partiellement évaluée. Sinon (on a une valeur pour chaque argument), on récupère le corps de la fonction dans l'environnement d'évaluation,

— si c'est une fonction utilisateur (définie par un arbre f), on l'*applique* aux arguments : on construit l'arbre

$$Seq(Def(a_1, v_1), Seq(Def(a_2, v_2), ...Seq(Def(a_n, v_n), f)...))$$

puis on l'évalue dans un nouveau bloc.

— sinon, c'est une fonction prédéfinie ("+", "p_int"...) définie par du code caml, on applique la fonction caml à la liste des arguments.

Boucles L'idée de base est que `repete e fin.` est équivalent à `e, repete e fin.`, ou encore, pour les boucles finies, `repete n fois e fin.` est équivalent à `e, repete n-1 fois e fin.`. On repousse donc l'évaluation de la boucle jusqu'à sa disparition éventuelle !

Think Le language contenant des boucles, l'enfant peut écrire des programmes dont la première action est longue à atteindre (`repete 5151515151515151 fois soit a=1 fin, av.`), voire inatteignable (`repete soit toto=33 fin, saute`).

Or, le monde abstrait ne peut pas se permettre de rester bloqué dans de telles situations (gel de l'affichage dans le premier cas, mort de tout le monde dans le second...)

La solution adoptée est de déclencher l'action *Think*, lorsque les calculs au cours d'un top deviennent trop longs. Cette action sera alors interprétée à l'affichage par une animation adéquate.

Ainsi, le monde abstrait ne se retrouve pas bloqué et suit son chemin, et l'enfant, dont le programme est *lourd* ou *faux* a les moyens de s'en apercevoir, et de toucher du doigt les problèmes de complexité (pub pour Marianne :).

C.2.3 Gestionnaires d'évènements

La gestion des évènements est relativement simple : les gestionnaires d'évènement sont des procédures définies par l'élève, mais nommées de telle sorte qu'il ne puisse pas les appeler lui-même. On maintient à l'exécution une pile permettant de gérer des appels imbriqués aux gestionnaires : Lorsqu'un évènement est déclenché, on empile le gestionnaire correspondant ; à chaque top on dépile l'expression se trouvant en haut de la pile, on l'évalue, ce qui nous donne une action et un résidu, si l'on a une vraie action, on la renvoie et on rempile le résidu pour la fois d'après, sinon, si le résidu est vide, on continue en dépilant, sinon, on continue d'évaluer le résidu.

D Règles du parser pour le langage de Rigobot

start :
| phrase DOT
;
phrase :
| expected_unit_expr_seq
| fundef
;
fundef :
| LET VARNAME fundef_arg_list EQ
any_expr_seq
;
/* expression dont on espère qu'elle sera typée unit */
expected_unit_expr_seq :
| unit_expr_seq
| maybe_unit_expr_seq
;
seq_prefix :
| unit_expr
| maybe_unit_expr
;
/* expression dont on est sûr qu'elle sera typée unit */
unit_expr_seq :
| unit_expr
| seq_prefix COMMA unit_expr_seq
;
unit_expr :
| simple_unit_expr
;
simple_unit_expr :
| atomic_action
| JMP
| LPAR unit_expr_seq RPAR
| LET VARNAME EQ expected_non_unit_expr
| CHANGE VARNAME EQ expected_non_unit_expr
| REP expected_unit_expr_seq END
| DO expected_non_unit_expr TIMES expected_unit_expr_seq END
| WHILE expected_non_unit_expr DO expected_unit_expr_seq END
;
atomic_action :
| move
| rot
| START_MARK
| STOP_MARK
| TAKE
| PUT
| ACT

```
| USE
;
   rot :
| ROT_L
| ROT_R
| UTURN
;
   move :
| STEP
| STEP_B
| STEP_L
| STEP_R
;
   /* expression qui peut être typée unit ou non */
   maybe_unit_expr_seq :
| maybe_unit_expr
| seq_prefix COMMA maybe_unit_expr_seq
;
   maybe_unit_expr :
| simple_maybe_unit_expr
| funapp_with_args
;
   simple_maybe_unit_expr :
| VARNAME
| LPAR maybe_unit_expr_seq RPAR
| ite
;
   funapp_with_args :
| VARNAME arg_list
;
   /* expression dont on est sûr qu'elle ne sera pas typée unit */
   non_unit_expr_seq :
| non_unit_expr
| seq_prefix COMMA non_unit_expr_seq
;
   non_unit_expr :
| simple_non_unit_expr
| binop
| CALC simple_expected_non_unit_expr
;
   simple_non_unit_expr :
| constant
| LPAR non_unit_expr_seq RPAR
;
   binop :
| arithm_expr
| bool_expr
| string_expr
;
```

```
   constant :
| INT
| BOOL
| STRING
;
   arithm_expr :
| simple_expected_non_unit_expr PLUS simple_expected_non_unit_expr
| simple_expected_non_unit_expr MINUS simple_expected_non_unit_expr
| simple_expected_non_unit_expr MULT simple_expected_non_unit_expr
| simple_expected_non_unit_expr DIV simple_expected_non_unit_expr
;
   bool_expr :
| simple_expected_non_unit_expr LT simple_expected_non_unit_expr
| simple_expected_non_unit_expr LE simple_expected_non_unit_expr
| simple_expected_non_unit_expr GT simple_expected_non_unit_expr
| simple_expected_non_unit_expr GE simple_expected_non_unit_expr
| simple_expected_non_unit_expr EQ simple_expected_non_unit_expr
| simple_expected_non_unit_expr NEQ simple_expected_non_unit_expr
;
   string_expr :
| simple_expected_non_unit_expr CIRCUMFLEX simple_expected_non_unit_expr
;
   /* expression dont on espère qu'elle ne sera pas typée unit */
   expected_non_unit_expr_seq :
| maybe_unit_expr_seq
| non_unit_expr_seq
;
   expected_non_unit_expr :
| maybe_unit_expr
| non_unit_expr
;
   simple_expected_non_unit_expr :
| simple_maybe_unit_expr
| simple_non_unit_expr
;
   arg :
| simple_expected_non_unit_expr
;
   arg_list :
| arg
| arg arg_list
;
   fundef_arg_list :
| NOTHING
| var_list
;
   var_list :
| VARNAME
| VARNAME var_list
```

```
;
    /* expression qui peut indifféremment être typée unit ou non */
    any_expr_seq :
| unit_expr_seq
| expected_non_unit_expr_seq
;
    ite :
| IF expected_non_unit_expr_seq THEN any_expr_seq ELSE any_expr_seq
END
| IF expected_non_unit_expr_seq THEN any_expr_seq END
;   inputbiblio
```