# I. Introduction

Dans ce projet il est question de systèmes multi-agent (SMA). En informatique, il s’agit d’un domaine de recherche visant à modéliser le comportement d’un ensemble d’agents en interaction avec leur environnement. La partie suivante explique plus en détails les origines et les objectifs de ces modèles.

## 1) Les systèmes multi-agents (SMA)

Comme pour tout modèle, un système multi-agent a pour but de simplifier la complexité d’une situation afin de mieux la comprendre et de pouvoir l’étudier. Cependant, cette simplification ne doit pas nuire à son bon fonctionnement, ni à la qualité des résultats qu’il fournit.

Même si le terme modèle est souvent associé aux mathématiques, il y a des circonstances où la mise en place d’un modèle mathématique ne peut fournir de résultats satisfaisants. Dans ce cas, les personnes qui étudient une situation sont amenées à concevoir des modèles de données associées à un programme informatique afin de traduire au mieux cette dernière.

C’est le cas des systèmes multi-agent. Ces modèles visent à reproduire au mieux le comportement d’un ensemble d’agents qui évoluent dans un environnement et qui sont capables d’interagir avec ce dernier. Un agent est une entité, plus ou moins autonome, apte à prendre des décisions en fonction du contexte dans lequel il se trouve. Un agent peut servir à représenter une gamme très large d’entités allant d’un processus informatique à un être humain par exemple.

La conception d’un SMA soulève en général plusieurs problématiques liées au degré d’intelligence des agents. Ainsi, il faut s’interroger sur la manière dont ils prennent des décisions, sur la manière avec laquelle ils perçoivent leur environnement ou encore sur la manière dont ils peuvent collaborer. Suivant leur comportement, les agents peuvent être classés en catégorie, agents cognitifs ou agents réactifs par exemple. Il est également possible de définir une organisation entre les agents de manière à reproduire une hiérarchie, des coalitions ou encore des marchés pour simuler le commerce d’objets entre ceux-ci.

Cette introduction montre la richesse et les possibilités qu’offre la mise en place d’un SMA. Cependant, avant de développer un tel système il faut d’abord penser à ce que l’on veut modéliser. La partie suivante a pour rôle d’expliquer au lecteur en quoi consiste ce projet et permet de définir ce qui a été modélisé.

## 2) Présentation du projet

D’une manière générale, l’objectif de ce projet était de concevoir un programme permettant de simuler le développement d’une civilisation primaire dans un environnement donné. Cette idée vient de ce qui se fait dans certains jeux de stratégie en temps réel où le joueur contrôle un ensemble d’unités et doit donner des ordres à celles-ci afin de développer au maximum sa civilisation. Le développement d’une civilisation passe bien entendu par le développement économique de celle-ci mais aussi par le développement militaire. Le joueur doit alors guider ses unités tant sur le plan de la collecte des ressources et de la construction de bâtiments que sur le plan militaire où il doit adopter la meilleur stratégie, afin de vaincre ses adversaires.

Le SMA développé au cours de ce projet reprend quelques-unes des grandes lignes de ce type de jeux. En effet, il a été choisi de se focaliser uniquement sur la partie qui touche au développement économique d’une civilisation. Ce développement est bien entendu lié au cadre spatiotemporel dans lequel la civilisation évolue. Il va de soi que la stratégie de développement économique adoptée par un pays à l’air de la mondialisation n’est pas la même que celle adoptée par l’empire romain plusieurs siècles avant Jésus Christ.

Dans le cadre de ce projet, il a été choisi de faire évoluer les civilisations à une époque plus ancienne que la nôtre. En effet, dans cette simulation les agents doivent être en mesure de collecter des ressources directement accessibles dans leur environnement, comme du bois, de la nourriture ou encore des minerais. Ensuite à partir de ces ressources, ils doivent pouvoir construire des bâtiments, et se multiplier afin de faire croitre leur population. Pour donner un lieu et une époque correspondant à ce contexte on pourrait prendre l’Amérique à l’époque de sa découverte par Christophe Colomb en 1492. A cette époque, les Colomb disposaient du savoir nécessaire pour développer une civilisation assez complexe, le tout sur un territoire quasiment vierge et riche en ressources naturelles.

Si l’on ramène maintenant ce problème au cadre des SMA, le principe du projet est de faire coopérer un ensemble d’agents dans un environnement riche en ressources naturelles. Les agents doivent donc unir leur force afin de collecter des ressources qu’ils mutualisent. L’objectif final de ce système étant que les agents construisent des bâtiments - grâce aux ressources - afin d’accroitre leur population. Du point de vu organisationnel, les agents sont donc structurés en équipe, c’est-à-dire qu’ils travaillent ensemble à la réalisation d’objectifs communs, et qu’ils cherchent à maximiser les intérêts de l’équipe plutôt que leurs intérêts personnels. La simulation prend fin dès lors qu’il n’y a plus de ressources disponibles dans l’environnement.

Bien entendu, ce projet n’a pas une d’application scientifique précise. Il constitue surtout un sujet de découvertes des systèmes multi-agent et de la simulation stochastique à évènements discrets.

Avant de passer à la phase l’analyse, il est nécessaire de définir les outils techniques utilisés pour mener à bien le développement de ce programme. C’est le rôle de la partie suivante qui explique quels langages, quels IDE et autre outils de développement ont été utilisés pour la conduite de ce projet.

# II. Outils et méthodes

## 1) Présentation des outils de développement utilisés

Différents outils et technologies ont été utilisés pour mener à bien ce projet. Dans cette partie il sera question d’expliquer le rôle de chacun d’eux et de dire en quoi ils ont permis d’améliorer la qualité du programme développé.

### i) Doxygen

La documentation d’un projet en informatique est primordiale. Une bonne documentation permet d’avoir un code compréhensible et maintenable, celle-ci peut passer uniquement par des commentaires mais ce système ne permet pas d’avoir une vision globale du projet. De plus, les développeurs utilisant une API ou une bibliothèque ont besoin d’une documentation des fonctionnalités sans avoir à consulter les sources de celles-ci. Ces besoins ont menés à la conception d’un outil permettant une meilleure lisibilité des documentations et une normalisation des commentaires de documentation.

Doxygen est un outil de génération de documentation pour du code source annoté. Doxygen est initialement prévu pour la documentation de code source C++ mais il supporte également les sources C, Objective-C, C#, PHP, Java, Python, IDL, Fortran, VHDL, Tcl et D. Le logiciel analyse les commentaires et le code écrit et en extrait la documentation à partir des balises placées dans les commentaires. La documentation peut ensuite être mise en forme selon différents formats :

* HTML pour héberger la documentation en ligne ;
* PDF, LATEX et autres pour une documentation hors-ligne ;
* Pages man pour UNIX.

Doxygen peut également être utilisé pour analyser la structure du code à partir de sources non documentées. Cette fonctionnalité trouve son utilité pour la reprise de projets de taille conséquente puisque Doxygen peut générer automatiquement les diagrammes d’héritage, de collaboration et de dépendance.

Doxygen est sous licence GPLv2, son code source peut donc être utilisé, copié, modifié et redistribué. De ce fait, le logiciel est portable sur la majorité des plateformes actuelles.

Ici, nous verrons la mise en place de Doxygen et son utilisation. Nous aborderons également son fonctionnement et les extensions possibles.

#### a) Installation

Le dépôt des sources de Doxygen est disponible sur <https://github.com/doxygen/doxygen> et les installeurs sur la page <http://www.stack.nl/~dimitri/doxygen/download.html>. Le tutoriel permet une installation simple de Doxygen. Bien que le logiciel puisse être utilisé uniquement en ligne de commande, l’outil Doxywizard permet d’ajouter une interface pour l’édition du fichier de configuration de Doxygen ou plus communément appelé Doxyfile.

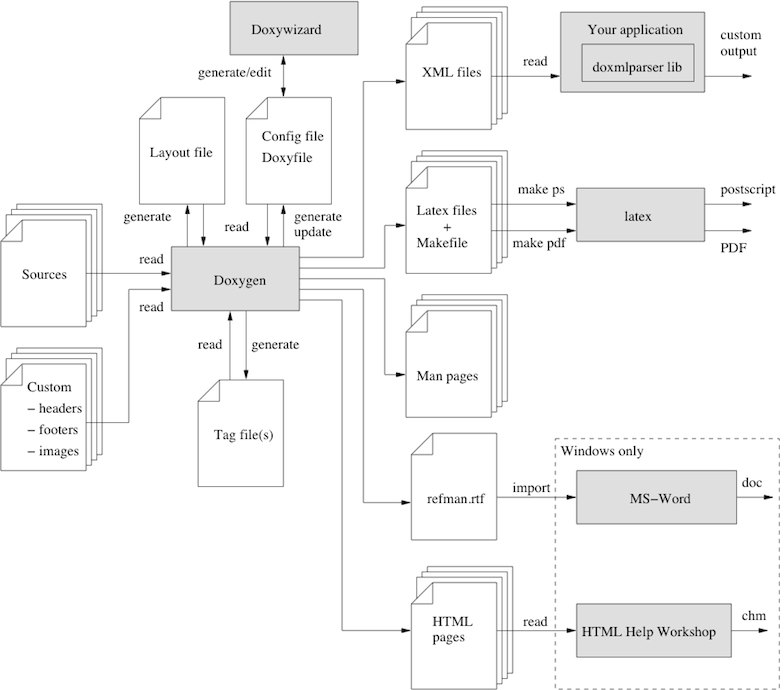


Figure 1 - Flux d'informations de Doxygen

#### b) Fonctionnement

L’exécutable doxygen est le cœur de l’outil, il permet de transformer les annotations des sources en différents formats de documentation selon le fichier de configuration ou les arguments passés. Dans cette partie, nous verrons les différents modules et leur rôle dans ces opérations.

##### Parsers de configuration

Le rôle des parsers de configuration est de récupérer les paramètres de la ligne de commande ou du fichier de configuration afin de rendre cette configuration exploitable par les autres modules de Doxygen. Les arguments non renseignés sont fixés à une valeur par défaut. Le fichier de configuration permet d’obtenir des informations telles que les sources à analyser, les formats de sorties à utiliser, si les sources non documentées sont à analyser ou encore un fichier de tags à parser. Un tel fichier sera pris en charge par un autre module de chargement de configuration.

Doxygen permettant une configuration avec de très nombreuses options, celles-ci ne seront pas toutes détaillées ici, cependant, une liste exhaustive des arguments possibles est disponible sur <http://www.stack.nl/~dimitri/doxygen/manual/config.html>.

##### Parsers de sources

Une fois la configuration chargée, un préprocesseur C va analyser les sources spécifiées dans la configuration, celui-ci s’exécutera sur les fichiers spécifiés dans les #include des sources, permettant ainsi d’avoir une connaissance de l’intégralité de la carte de code à analyser.

Les sources modifiées par le préprocesseur sont ensuite envoyées à un module unique permettant de parser tous les langages compatibles avec Doxygen via l’utilisation d’une machine à états. Le découpage de ce module en des modules spécifiques à des langages est envisagé dans l’avenir de Doxygen.

##### Organiseur de données

En sortie de ces modules, Doxygen possède un arbre d’entrées correspondant aux classes, fichiers, namespaces, variables, fonctions, packages, pages et groupes extraits. Le rôle de l’organiseur de données et de construire des dictionnaires catégorisés mais également de créer les relations et les héritages entre les entités de l’arbre.

##### Générateurs de sortie

Après la récupération et l’organisation des données, Doxygen génère des sorties dans différents formats. Les générateurs de sorties implémentent tous la classe OutputGenerator. Pour l’instant, un fichier XML peut être généré directement de la sortie de l’organiseur de données. Dans le futur de Doxygen, ce fichier XML pourra être utilisé comme langage intermédiaire pour les générateurs de sorties afin de les détacher encore plus du cœur du logiciel.

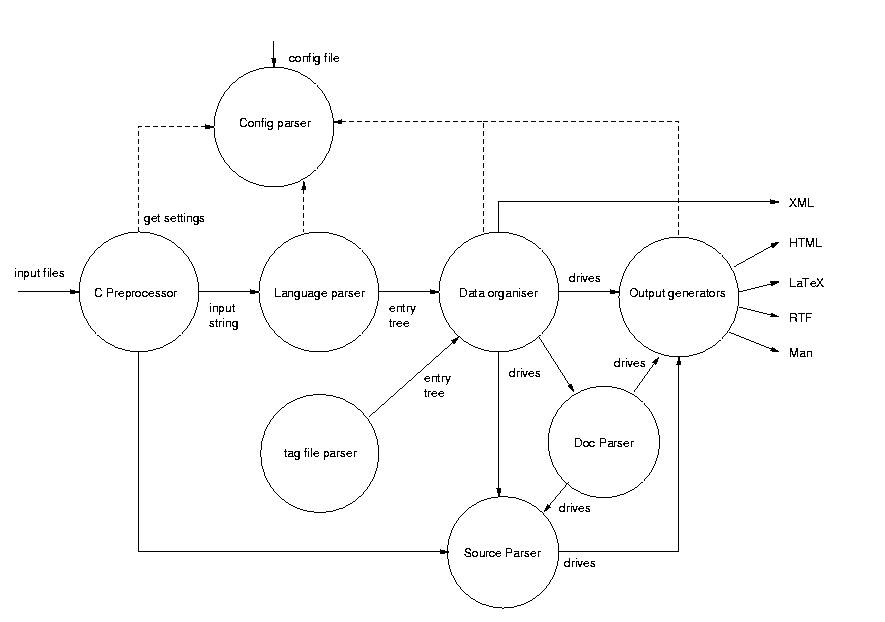


Figure 2 - Structure du noyau de Doxygen

#### c) Création de documentation

##### Syntaxe

Doxygen dispose de différentes syntaxes permettant la documentation des sources. Entre autres, l’une d’entre elle supporte la syntaxe de la Javadoc. Doxygen récupère les informations de l’utilisateur dans des blocs de commentaire spécifiques (Voir Figure 3). Ces commentaires documentent soit l’entité se trouvant à leur suite, soit une entité donnée si elle est spécifiée.

/\*\*

\* Bloc de commentaire Doxygen

\*/

/\*!

\* Autre bloc de commentaire Doxygen

\*/

/\*

\* Bloc de commentaire non traité par Doxygen

\*/

/// Ligne de commentaire Doxygen

//! Autre ligne de commentaire Doxygen

// Ligne de commentaire non traitée par Doxygen

Figure - Commentaires Doxygen

##### Balises

Doxygen récupère les textes associés aux balises dans les commentaires pour créer la documentation. Ces balises commencent soit par un backslash (\) ou par une arobase (@).

Doxygen possède de nombreuses balises, d’autres peuvent être spécifiées par l’utilisateur via des fichiers de tags. Ici, nous verrons les balises les plus communément utilisées via des exemples d’utilisation.

##### Balisage général

Certaines balises peuvent être utilisées sur toutes les entités, celles-ci donnent des descriptions d’ordre général sur les entités. La Figure 4 donne un exemple d’utilisation de ces balises.

/\*!

\* **\brief** Description de la classe

\* **\details** Cette classe est utilisée pour détailler des balises générales

\* **\author** John Doe (Auteur de la classe)

\* **\author** John Smith (Autre auteur de la classe)

\* **\version** 0.5.6b (Version de la classe/du programme)

\* **\date** 1992-2014 (Date de développement du programme)

\* **\pre** Initialiser le système (prérequis d’utilisation)

\* **\bug** Toute la mémoire n’est pas libérée (bug lié à l’utilisation)

\* **\warning** Peut causer des crashs (avertissement lié à l’utilisation)

\* **\copyright** GNU Public Licence **(**copyright d’utilisation de la classe)

\*/

class ExampleClass **{};**

Figure - Exemple d'utilisation des balises générales

##### Balisage de fonction et méthodes

Des balises plus spécifiques permettent de documenter des fonctions. Celles-ci donnent des informations sur les paramètres de la fonction et son retour. La Figure 5 donne un exemple de documentation de fonction.

/\*!

\* **\brief** foo Ce texte sera dans la description rapide de la fonction

\* \details Ce texte sera consultable dans la description détaillée

\* **\param** bar Ce texte documente le rôle du paramètre

\* **\param** src[in] Ce paramètre est un paramètre d'entrée de la fonction

\* **\param** dest[out] Ce paramètre est un paramètre de sortie de la fonction

\* **\param** mod[in,out] Ce paramètre est en entrée et en sortie de la fonction

\* **\return** Ce texte documente la sortie de la fonction

\*/

double foo **(**int bar**,** const void **\***src**,** void **\***dest**,** char **\***mod**);**

Figure - Exemple de documentation de fonction

##### Balisage de classe et énumération par notation post fixée

Le balisage d’une classe et d’une énumération nécessite parfois l’utilisation d’un balisage post-fixe pour alléger la notation et pouvoir faire la documentation sur la même ligne que la déclaration. La Figure 6 donne un exemple de cette utilisation.

/\*!

\* **\brief** Classe documentée par Doxygen

\*/

class Test

**{**

public**:**

/\*!

\* **\brief** Une énumération documentée

\*/

enum TEnum **{**

TVal1**,** //!< Valeur d'énumération TVal1 en notation post fixée

TVal2**,** //!< Valeur d'énumération TVal2 en notation post fixée

TVal3 //!< Valeur d'énumération TVal3 en notation post fixée

**};**

/\*!

\* **\brief** Constructeur de Test

\* ...

\*/

Test**();**

/\*!

\* **\brief** Destructeur de Test

\* ...

\*/

**~**Test**();**

/\*!

\* **\brief** Méthode publique

\* ...

\* **\param** a Paramètre de méthode

\* **\return** Retour de méthode

\*/

int method**(**int a**);**

int m\_Attribut**;** //!< **\brief** Attribut public en notation post fixée

**};**

Figure - Exemple de documentation de classe et énumération

##### Génération de documentation

La configuration du fichier de configuration de Doxygen peut être faite via l’utilitaire DoxyWizard. Celui-ci peut également lancer la génération. Il faut lui spécifier l’emplacement des sources, des répertoires de destinations, choisir les formats de destinations et d’autres options plus spécifiques.

Une de ces options est la génération de diagramme avec l’utilisation du plugin Graphviz/dot. Celui-ci permet une visualisation des héritages entre les différentes classes et d’obtenir un diagramme UML du projet (Voir exemple en Figure 7)

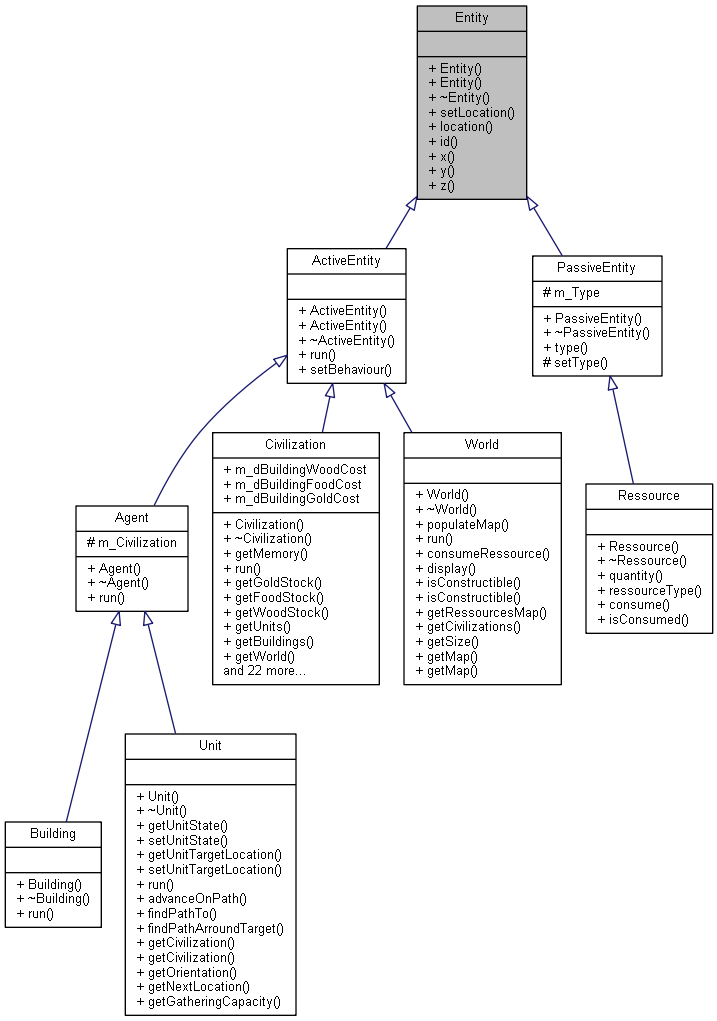


Figure - Exemple de diagramme UML généré par Doxygen avec Graphviz/dot

#### d) Conclusion

Doxygen est un outil très complet dont on découvre de nouvelles facettes à chaque nouvelle utilisation. Ce logiciel devient vite un outil indispensable à tout projet et brille par ses fonctionnalités et sa modularité. Cependant, la syntaxe utilisée pourra amener certains à ne pas apprécier son utilisation. Cette gêne peut facilement être occultée par l’installation de plugin de génération automatique de commentaires Doxygen à son environnement de développement préféré.

### ii) Github

### iii) Langage C++

Le SMA de ce projet a été développé en C++. Ce choix de langage a été motivé par différentes raisons. La première est qu’il fallait que le langage utilisé offre la possibilité de mettre en œuvre le paradigme de la programmation objet. En effet, ce type de langage offre un niveau d’abstraction suffisamment élevé pour mettre en place des systèmes complexes comme ceux de que requièrent le développement d’un SMA. De plus, la programmation objet facilite le maintien, l’amélioration et le débogage d’un code source. Enfin, de par la nature atomique des objets, elle facilite le découpage et la répartition du travail à accomplir dans une équipe afin de mener à bien un projet de développement.

Une des critiques que l’on peut faire du C++ est qu’il reste un langage d’assez bas niveau, proche de la machine et donc moins portable qu’un langage plus récent comme le C# ou le Java. Cependant, un langage de bas niveau n’a pas que des inconvénients. En effet, en termes de performances, rien n’égale la vitesse d’une instruction assembleur directement exécutée par un processeur. Lorsqu’un programme C++ est généré, de telles instructions sont écrites et assemblées dans une image exécutable que le processeur peut directement lancer. Avec les langages évolués, un certain nombre de mécanismes sont mis en place pour augmenter entre autre la portabilité du code, mais ils peuvent également ralentir l’exécution du programme. Ainsi, un programme développé en Java nécessite une phase de compilation à la volée avant d’être exécuté par le processeur, ce qui a pour effet de le rendre moins performant (même si ce constat est de moins en moins vrai de nos jours du fait de l’amélioration des machines virtuelles et des compilateurs JIT).

De plus, de par la nature du programme, on peut d’ores et déjà se douter qu’un grand nombre d’objets risques d’être alloués / désalloués en temps réel, comme c’est le cas dans de nombreuses simulations. Le fait que le C++ laisse à l’utilisateur la charge de gérer la mémoire permet d’éviter la lourdeur de certains mécanismes présents dans des langages plus évolués. En effet, en Java ou en C# lorsqu’un trop grand nombre d’objets sont alloués sur le tas, un mécanisme de « ramasse miette » est déclenché avec une haute priorité d’exécution. Ce mécanisme a pour but de libérer de la RAM afin de subvenir aux besoins du programme, mais il entraine souvent un ralentissement important de ce dernier.

Un compilateur C++ a donc été utilisé dans le cadre de ce projet. Il s’agit de celui présent dans la suite logicielle Visual Sudio 2012. Ce dernier supporte la norme C++ 11, c’est pourquoi, certaines nouveautés du langage ont pu être utilisées, comme l’inférence de types par exemple. Ce mécanisme permet de laisser au compilateur le soin de résoudre automatiquement le type de certaines variable à la compilation grâce au mot clé « auto ».

Maintenant que la question du langage de programmation a été discutée, il est temps de passer à celle qui vient logiquement derrière : le programme doit il s’appuyer sur une API et si oui laquelle choisir parmi celles du langage disponibles ? La partie suivante a pour rôle de répondre à ces questions.

### iv) Bibliothèque Qt

Un des choix effectués au moment de la rédaction du cahier des charges du programme, était de doter ce dernier d’une interface graphique. Celle-ci devait avoir pour rôle d’afficher en temps réel le déroulement de la simulation. Pour ce faire, il a été décidé de s’appuyer sur le framework Qt qui fournit les outils nécessaires au développement d’une interface homme-machine en C++.

L’utilisation de Qt garantit également une certaine portabilité du programme sur différents types d’architectures et d’OS. A défaut d’avoir choisi un langage portable comme le Java ou le C#, le choix d’utiliser une API portable permet de garantir une certaine indépendance du programme vis-à-vis de la plateforme sur laquelle il s’exécute.

En dehors de Qt aucune autre API n’a été utilisée. Le code source est donc conforme aux standards du C++ 11. C’est pourquoi, pour être compilé dans environnement donné, il suffit de posséder un compilateur à jour, et le déploiement du programme peut s’effectuer sans problèmes.

Ainsi s’achève la partie sur la présentation des outils et des technologies utilisées. Elle aura permis de préciser l’environnement de travail dans lequel s’est déroulé le projet, tout en justifiant certains choix techniques effectués. Désormais, il est temps d’aborder un point essentiel du projet : l’analyse. C’est le rôle de la partie suivante qui détaille l’étude menée, au niveau génie logiciel et algorithmique, pour répondre aux besoins du SMA développé.

## 2) Analyse

Dans cette partie, titrée analyse, il est question de réflexion sur l’étude menée pour développer concrètement le SMA du projet. De ce fait, une première sous-partie traite des techniques de génie logiciel utilisées pour implémenter les différentes spécifications présentées dans la partie présentation du projet. Puis dans un second temps, une deuxième sous-partie explique les différents problèmes purement algorithmiques soulevés par le projet.

### i) Génie logiciel

Pour rappel le SMA implémenté dans ce projet a pour but de simuler le développement d’une civilisation humaine dans un écosystème riche en ressources naturels. Pour ce faire, les humains ou plutôt les agents doivent se déplacer dans l’environnement et collecter des ressources. L’objectif final étant de collecter un maximum de ressources. Les ressources sont bien sûr consommables et servent à produire des bâtiments et de nouvelles unités (accroissement de la population). Comme le rapprochement l’a déjà était fait précédemment, on pourrait associer le déroulement de cette simulation à un scénario dans lequel des explorateurs débarqueraient sur une île déserte et devraient s’y installer. Pour survivre et prospérer, ils devraient alors tirer profit des richesses naturelles environnantes et adopter une stratégie collective afin de maximiser leurs collectes.

#### a) Le framework MAVIS

Après ce bref rappel, il faut réfléchir à la stratégie à adopter pour concevoir ce système. Etant donné que les spécifications techniques (développement en C++) du projet permettent d’utiliser les concepts de la programmation objets, il faut tirer profit au maximum de cette caractéristique. Ainsi, au-delà du paradigme objet en lui-même, un des points forts de ce type de programmation est qu’il facilite la réutilisabilité et l’export de modèles déjà développés. C’est-à-dire que face à un certain nombre de situations, il est possible qu’un modèle objets ait déjà été créé. Dans ce cas, il est possible que ce modèle soit réutilisable dans d’autres situations similaires, et grâce à l’approche objet, ce modèle s’exporte très bien d’une situation à l’autre. Ces modèles presque génériques et réutilisables dans plusieurs situations s’appellent des design patterns. C’est également une des grandes forces de la conception orientée objet. Ces patrons servent en quelques sortes de guide au moment de la phase de conception d’un programme. En effet, la plupart du temps ils sont le fruit d’une réflexion complexe concernant un problème à traiter. La solution ainsi proposée par un pattern est bien souvent une solution de qualité. C’est pourquoi, dans la mesure du possible, avant de se lancer dans la conception d’une solution novatrice, il est préférable de rechercher les éventuels patterns qui traitent de problèmes similaires.

Dans le cas des SMA justement, des patterns, et même des frameworks sont disponibles. Un framework baptisé MAVIS (Multi-Agent Visual Interactive Simulation), développé à l’institut Blaise Pascal, leur est d’ailleurs consacré. Un framework est un ensemble cohérent de composants logiciels structurels. Il diffère d’un patron de conception pour plusieurs raisons, mais les deux principales sont qu’un pattern fournie une solution plus abstraite qu’un framework et qu’un pattern concerne une plus petite architecture logicielle qu’un framework. Dans le cas de MAVIS, le framework se compose d’un jeu de packages UML (Unified Modeling Language) permettant de mettre en œuvre la simulation d’un écosystème dynamique complexe. L’un de ces packages appelé « World » fournit un patron de conception intéressant dans le but de modéliser l’ensemble des entités d’un SMA dans lequel les agents sont en forte interaction avec leur milieu. De ce fait, le développement du simulateur de civilisation de ce projet s’inspire fortement de ce qui a été conçu dans le package « World » de MAVIS, dont voici le diagramme UML :



Figure 8 - Package World du framework MAVIS

Cette architecture présente différents points remarquables qu’il est intéressant de détailler pour comprendre les choix effectués par la suite.

Le premier est qu’elle fait la distinction entre deux types d’entités, les actives et les passives. Les entités passives diffèrent des actives de par le fait qu’elles ne seront jamais transmises au flow d’exécution du programme. C’est-à-dire qu’elles n’ont pas de comportement à proprement parlé comme les agents. Elles se contentent d’exister et d’être éventuellement consommées par les agents. Pour donner un exemple concret, dans le cadre du simulateur de civilisation, les entités passives représentent les ressources que peuvent collecter les agents.

Le second point intéressant de ce pattern est qu’il permet aux entités actives de changer de comportement pendant l’exécution de la simulation. Ce changement en temps réel est rendu possible grâce à la mise en œuvre du pattern stratégie. Il s’agit pour les entités actives d’implémenter une interface appelée behaviour. Cette interface comporte une méthode appelée « execute ». Celle-ci est chargée de définir le comportement d’une entité active qui sera exécuté à chaque pas de temps de la simulation. Cette interface est bien entendu spécialisable grâce aux mécanismes de l’héritage. Ainsi elle permet de définir un ou plusieurs comportements propres à chaque spécialisation d’entité active.

Le dernier point intéressant du package se situe au niveau de la classe Group. En effet, on remarque que celle-ci hérite de la classe Agent et qu’un groupe contient lui-même des agents. Ce pattern porte le nom de composite. Mais ce qui est réellement intéressant c’est que le groupe soit considéré comme une entité active. Cela implique qu’un groupe a lui aussi un comportement et qu’il influence celui des agents qu’il contient. C’est une des caractéristiques intéressantes du package car il vrai que dans la réalité, les agents auront une tendance naturelle à être influencé par le groupe dans lequel il vive. Cette partie du package fournit donc un élément de réponse sur la manière de modéliser cette réaction chez les agents.

#### b) Modèle objet des entités du SMA

Dans le cadre du simulateur de civilisation développé, le package « World » de MAVIS a donc servi de socle pour débuter la phase de conception. Les 3 grands principes expliqués ci-dessus ont donc été repris et adaptés au SMA de ce projet. Voici le diagramme UML déduit de cette étude :



Figure 9 - Package World du simulateur de civilisations

Parmi les choses qui restent identiques à celles du package World de MAVIS, on retrouve le pattern stratégie mis en œuvre pour spécifier le comportement des entités actives. La branche des entités passives reste elle aussi la même, elle a juste été spécialisée sous la forme d’une classe Ressource. Cette dernière permet de modéliser trois types de ressources présentes dans l’environnement : du bois, des filons d’ors, et de la nourriture que les unités peuvent collecter. Toujours dans le registre de la spécialisation, on remarque que la classe Agent a été dérivée en deux sous-classes. D’un côté les unités, et de l’autre les bâtiments. Le choix d’avoir placé les bâtiments dans la catégorie des agents est justifié par le fait que les bâtiments peuvent produire des ressources, ainsi que des unités.

Cependant, certaines parties du package ont été adaptées. Ainsi, la classe Group a disparu pour laisser place à une classe Civilization. Cette dernière est chargée de gérer la répartition des tâches de collectes, d’exploration et de construction entre les unités qu’elle supervise. C’est elle qui coordonne le travail des agents et qui agit un peu à la manière d’un chef dans une équipe.

On remarque également que la mémoire des agents a été mutualisée et qu’elle appartient désormais à la Civilization. Ce choix est justifié par une volonté de faire collaborer les agents de manière à ce qu’ils puissent établir une cartographie commune de leur environnement. Ainsi, lorsqu’une unité explore l’écosystème, elle met à jour la mémoire commune de la civilisation, pour partager ce qu’elle a découvert, un peu comme le ferai des êtres humains avec un système de cartes géographiques. Grâce à cette mémoire, les unités peuvent rechercher les endroits ou d’autres unités ont localisé des ressources et s’y rendre afin de les collecter.

#### c) Hiérarchie de classes comportementales

A propos du comportement des entités actives, l’interface Behaviour fait en réalité l’objet d’un package. Celui-ci rassemble une hiérarchie de classes permettant d’implémenter le comportement de toutes les entités actives du modèle. Voici le diagramme UML de ce package :



Figure 10 - Package Behaviour du simulateur de civilisations

Ces classes sont en quelques sortes les cerveaux des entités actives. En effet, ce sont elles qui implémentent l’intelligence artificielle de chaque type d’entité active. Chaque classe du package Behaviour est liée à une classe du modèle présenté dans le package World. Ainsi chaque type d’entité active utilise l’une des classes de ce package pour agir et prendre des décisions au cours de la simulation.

Une des précisions à apporter par rapport au framework MAVIS est que ce dernier suggère de séparer la partie qui modélise les entités (package World) de celle qui gère le déroulement stochastique des évènements grâce à un package baptisé Simulator. C’est ensuite la classe World qui utilise ce package pour ordonnancer les évènements de la simulation.

Dans le cas du simulateur de civilisation de ce projet, cette fonctionnalité est directement implémentée par les classes World et Civilization à travers leurs comportements respectifs (WorldBehaviour et CivilizationBehaviour). Ainsi à chaque pas de temps de la simulation, la classe World va réactiver les civilisations qu’elle contient. C’est-à-dire qu’elle va leur transmettre le flow d’exécution du programme afin que chaque civilisation puisse exécuter son propre comportement. L’ordre de réactivation est bien entendu généré pseudo-aléatoirement. Et de la même manière chaque civilisation va alors réactiver ses agents dans un ordre pseudo-aléatoire.

Dans le cas de la classe Civilization, en plus gérer l’ordonnancement de ses agents, cette dernière a besoin de définir une intelligence artificielle (IA) permettant de gérer une équipe. C’est aussi la classe CivilizationBehaviour qui implémente ce comportement. Grâce à cette classe, la civilisation est capable de gérer le travail collaboratif de collecte de ressources des unités. L’IA en elle-même sera plus détaillée dans la suite du rapport.

On notera également la présence 3 classes permettant de gérer trois comportements chez les unités. La première classe baptisée, NopBehaviour, permet à une unité de ne rien faire. Ce comportement lui permet d’attendre que la civilisation l’affecte à une tâche de construction ou de récolte. Le second comportement appelé, GathereBehaviour permet de définir la stratégie de collecte adoptée par les unités. Cette classe utilise un algorithme pour définir la manière dont les unités cherchent et collectent des ressources. Et enfin la 3ème et dernière classe comportementale des unités s’appelle BuilderBehaviour. Cette dernière permet à une unité de choisir un emplacement de construction dans son environnement, afin de s’y rendre et de construire un bâtiment.

#### d) Gestion du système d'information géographique

#### e) Générateurs pseudo-aléatoires

### ii) Problèmes algorithmiques

## 3) Implémentation

### i) Stratégies des entités actives et des agents

### ii) Algorithme de Pathfinding

### iii) Génération aléatoire de cartes à l’aide d’un bruit de Perlin

La génération de carte a été un point important de ce projet. En effet, nous cherchions à obtenir une carte aléatoire mais réaliste. Les algorithmes cherchant à atteindre des comportements imprévisibles utilisent trop souvent à tort uniquement des générateurs de nombres aléatoires. Le problème de données complétement aléatoire est qu’elles rendent le résultat peu réaliste et peu crédible. Le bruit de Perlin est une des solutions permettant d’obtenir des résultats aléatoires et naturels.

#### a) Introduction au bruit de Perlin

Une observation de la nature nous donne l’idée directrice ayant mené au bruit de Perlin. En effet, les éléments naturels ont comme point commun leur nature fractale. Un exemple commun peut être l’observation d’une montagne qui, à une large échelle, peut être considérée comme une grande variation de hauteur (la montagne), à une moindre échelle, comme des variations moyennes (les monts), puis de petites variations (rochers), puis de minuscules variations (pierres et cailloux) et ainsi de suite. On peut également remarquer que la fréquence d’apparition des variations est inversement proportionnelle à l’amplitude de la variation. Le bruit final est donc composé d’octaves de fréquence 1/n et d’amplitude n. C’est en partant de ce principe que Ken Perlin a développé ce bruit.

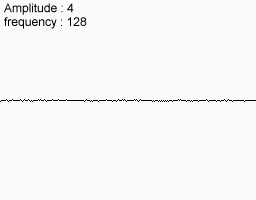
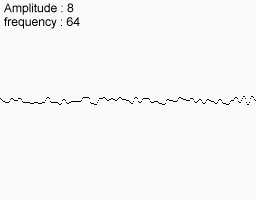
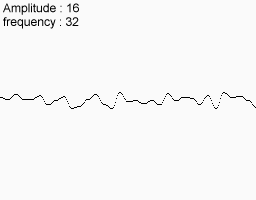
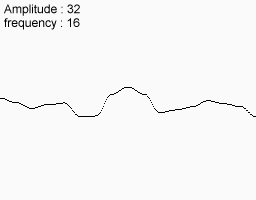
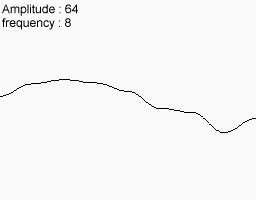
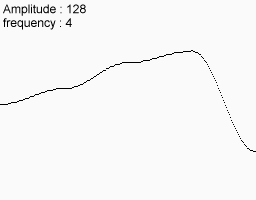


Figure 11 - Fonctions de bruit pour différentes octaves

Le bruit de Perlin nécessite deux éléments principaux : un générateur aléatoire et une fonction d’interpolation. Pour chaque octave, le générateur tirera un nombre de valeurs inverse à la fréquence et dans la plage d’amplitude de l’octave. La fonction d’interpolation sera ensuite utilisée pour obtenir la fonction de bruit associée à l’octave. La génération de bruit de Perlin repose donc sur la génération de fonction de bruit pour des octaves de fréquence différente puis à sommer ces fonctions (Figure 12).

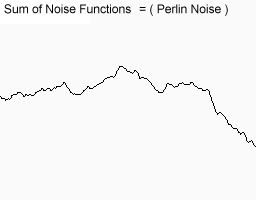


Figure 12 - Somme des octaves

Le principe de ce bruit peut se généraliser à un bruit en N dimensions. Dans notre cas, nous utiliserons une génération en deux dimensions pour la création de la carte.

#### b) Bruit Simplex

Le bruit de Perlin permet d’obtenir des rendus naturels mais le problème de sa complexité () peut poser problème lorsque l’on génère un bruit dans un espace de plus grande dimension. La variante du bruit Simplex permet d’obtenir le même résultat que le bruit de Perlin pour une complexité bien plus faible () pour un espace de plus grande dimension. C’est cette variante que nous utiliserons pour la génération de carte. Le générateur de nombre aléatoire choisi est le Mersenne Twister présent dans la STL. Afin d’optimiser la génération de bruit, nous utiliserons l’implémentation tirant profit de fonctionnalités du C++11 de Solarian Programmer[[1]](#footnote-1).

#### c) Génération de carte

La génération de carte passe par différentes étapes : la création du terrain et la population de celui-ci. Le terrain est créé avec un coefficient d’échelle définissant la fréquence du bruit de Perlin (Figure 13). Un coefficient plus grand permet d’obtenir une carte avec des variations de hauteur plus lentes. La valeur d’élévation à un point donné est obtenue par la récupération de la valeur de bruit aux coordonnées du point (mises à l’échelle). Cette génération permet d’obtenir des altitudes aux points de la carte réalistes.

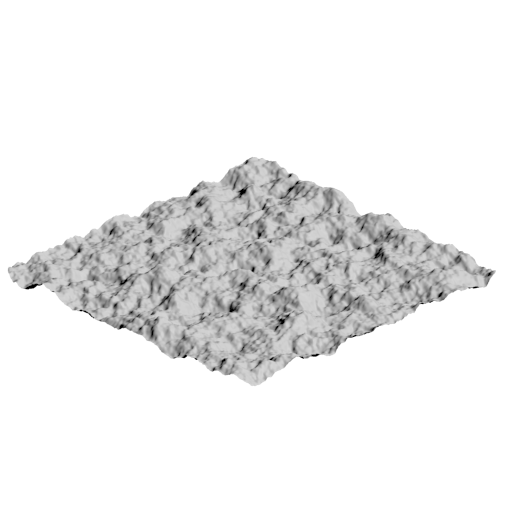
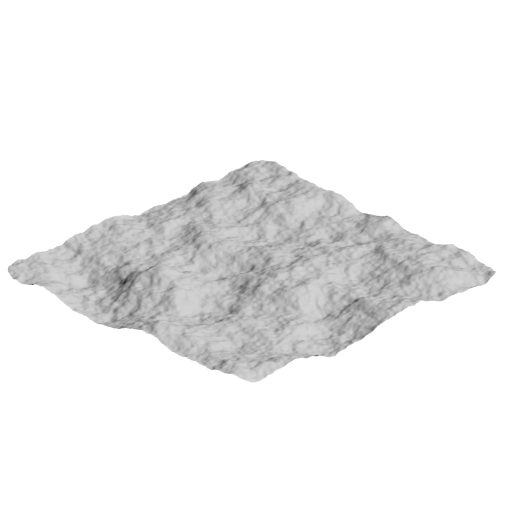


Figure 13 - Génération avec coefficient d'échelle de 0.25 (à gauche) et 0.50 (à droite)

Après cette phase de génération d’élévation, la carte est peuplée par les ressources qui y seront disponibles. Cette phase nécessite des paramètres d’échelle pour le bruit de Perlin et de densité de la ressource. La présence d’une ressource à un point donnée est déterminée par la valeur du bruit à ce point (avec les coordonnées mises à l’échelle). Si cette valeur est comprise dans l’intervalle alors la ressource est affectée. Le Tableau 1 donne les répartitions de ressources générées en fonction des paramètres de densité et d’échelle (ces évaluations sont effectuées avec le seed 3 pour le bruit de Perlin).

Table 1 - Répartition des ressources en fonction des paramètres de génération

|  |  |  |  |
| --- | --- | --- | --- |
| Densité 0.05 | Densité 0.15 | Densité 0.25 |  |
|  | | | Echelle 20.0 |
| Echelle 5.0 |
| Echelle 1.0 |

# III. Résultats

# IV. Discussion

1. http://solarianprogrammer.com/2012/07/18/perlin-noise-cpp-11/ [↑](#footnote-ref-1)