

Institut Supérieur d’Informatique de

Modélisation et de leurs Applications

Campus des Cézeaux

24 avenue des Landais

BP 10125

63173 AUBIÈRE Cedex

Rapport d’ingénieur

Projet de Simulation Multi-Agents 2ème année

Filière Génie Logiciel et Systèmes Informatiques

Simulateur de civilisation



Projet réalisé par Nicolas Prugne et Antoine Colmard

Projet demandé dans le cadre du cours d’UML avancé de M. Hill

# Remerciements

Nous tenons à remercier M. Hill pour la liberté qu’il nous a offert sur ce projet. Nous avons ainsi pu travailler sur une thématique de notre choix tout en apprenant de multiples choses à propos des simulations multi-agents.

# Résumé

d

Table des matières

[Remerciements i](#_Toc381707857)

[Résumé ii](#_Toc381707858)

[Table des figures et illustrations iv](#_Toc381707859)

[1 Introduction 1](#_Toc381707860)

[1.1 Les systèmes multi-agents 1](#_Toc381707861)

[1.2 Présentation du projet 1](#_Toc381707862)

[2 Outils et méthodes 3](#_Toc381707863)

[2.1 Présentation des outils de développement utilisés 3](#_Toc381707864)

[2.1.1 Doxygen 3](#_Toc381707865)

[2.1.2 Git & Github 8](#_Toc381707866)

[2.1.3 Langage C++ 10](#_Toc381707867)

[2.1.4 Bibliothèque Qt 11](#_Toc381707868)

[2.2 Analyse 11](#_Toc381707869)

[2.2.1 Génie logiciel 12](#_Toc381707870)

[2.2.2 Problèmes algorithmiques 19](#_Toc381707871)

[3 Résultats 31](#_Toc381707872)

[4 Conclusion 34](#_Toc381707873)

[Références bibliographiques 35](#_Toc381707874)

[Annexes I](#_Toc381707875)

# Table des figures et illustrations

[Figure 1 - Flux d'informations de Doxygen 4](#_Toc381707465)

[Figure 2 - Structure du noyau de Doxygen 5](#_Toc381707466)

[Figure 3 - Commentaires Doxygen 6](#_Toc381707467)

[Figure 4 - Exemple d'utilisation des balises générales 6](#_Toc381707468)

[Figure 5 - Exemple de documentation de fonction 6](#_Toc381707469)

[Figure 6 - Exemple de documentation de classe et énumération 7](#_Toc381707470)

[Figure 7 - Exemple de diagramme UML généré par Doxygen avec Graphviz/dot 8](#_Toc381707471)

[Figure 8 - Interface du client GitHub 10](#_Toc381707472)

[Figure 9 - Package World du framework MAVIS 13](#_Toc381707473)

[Figure 10 - Package World du simulateur de civilisations 14](#_Toc381707474)

[Figure 11 - Package Behaviour du simulateur de civilisations 15](#_Toc381707475)

[Figure 12 - Deux approches possibles pour gérer le système d'information géographique 16](#_Toc381707476)

[Figure 13 - Package Graph du simulateur de civilisation 17](#_Toc381707477)

[Figure 14 - Structures de graphe utilisées. 18](#_Toc381707478)

[Figure 15 - Package Generators du simulateur de civilisation 19](#_Toc381707479)

[Figure 16 - Gestion des listes d'unités d'une civilisation 20](#_Toc381707480)

[Figure 17 - Illustration de l'algorithme utilisé pour trouver un emplacement de construction 22](#_Toc381707481)

[Figure 18 - Trajectoires d'une unité. 23](#_Toc381707482)

[Figure 19 - Comparaison d'algorithmes de recherche de plus courts chemins 24](#_Toc381707483)

[Figure 20 - Cas mal géré par l'algorithme de recherche best-first 25](#_Toc381707484)

[Figure 21 - Fonctions de bruit pour différentes octaves 28](#_Toc381707485)

[Figure 22 - Somme des octaves 29](#_Toc381707486)

[Figure 23 - Génération avec coefficient d'échelle de 0.25 (à gauche) et 0.50 (à droite) 29](#_Toc381707487)

[Figure 24 - Capture d'écran du programme 31](#_Toc381707488)

[Figure 25 - Zoom sur une zone du monde 31](#_Toc381707489)

[Figure 26 - Evolution des quantités de ressources 32](#_Toc381707490)

[Figure 27 - Evolution de la population d'unités 33](#_Toc381707491)

# Introduction

Dans ce projet il est question de systèmes multi-agents (SMA). En simulation 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.

## Les systèmes multi-agents

Comme pour tout modèle, un système multi-agents 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-agents. 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.

## 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’aire 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 colons 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 environnements et outils de développement ont été utilisés pour la conduite de ce projet.

# Outils et méthodes

## 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.

### 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.

#### 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.

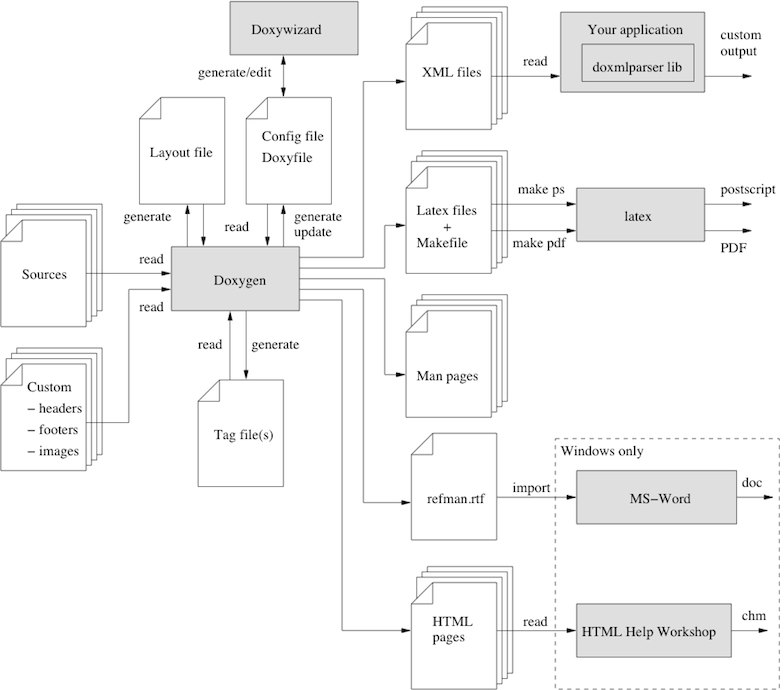


Figure 1 - Flux d'informations de Doxygen

#### 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.

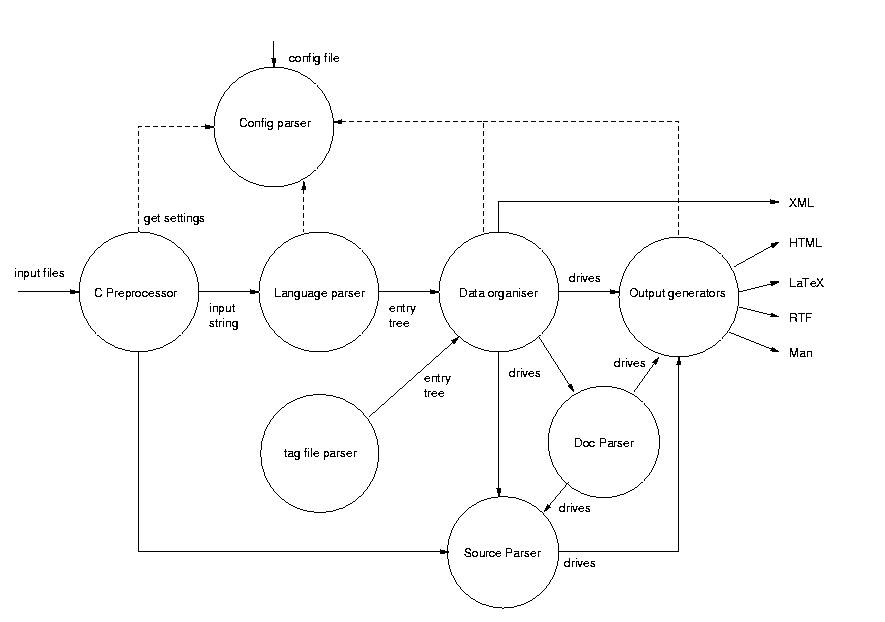


Figure 2 - Structure du noyau de Doxygen

#### 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 3 - 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 4 - 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 5 - 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 6 - 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)

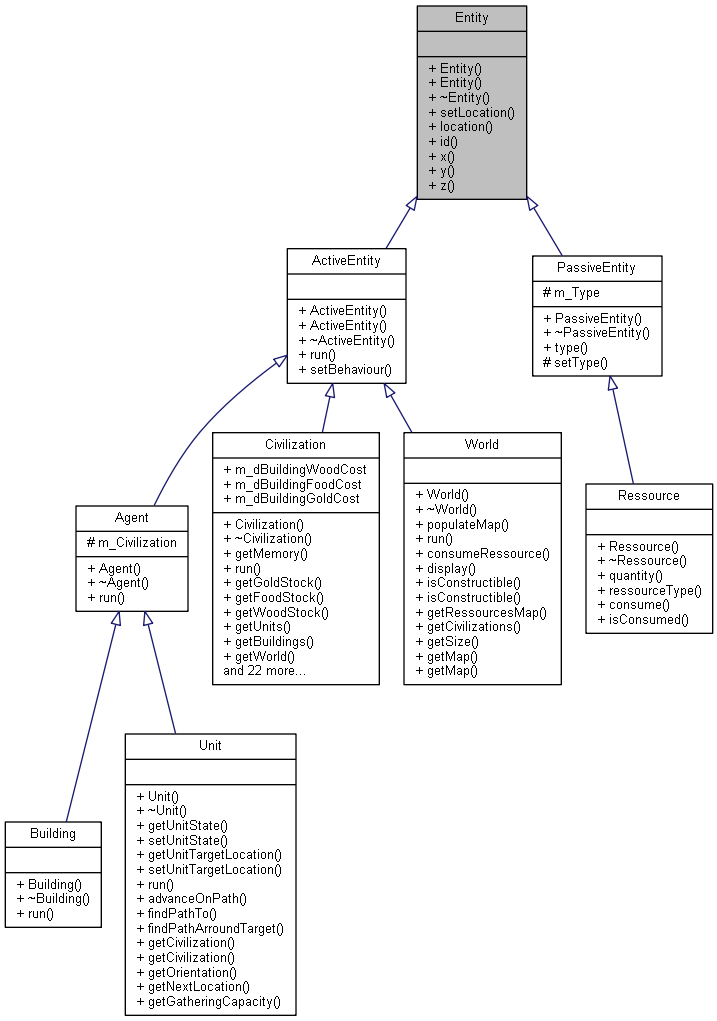


Figure 7 - Exemple de diagramme UML généré par Doxygen avec Graphviz/dot

#### 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.

### Git & Github

Lorsque l’on développe un projet assez volumineux, il peut être intéressant d’avoir recours à un logiciel de gestion de versions. Ces outils fournissent un ensemble de fonctionnalités qui permettent le suivi et la gestion d’un code source. La plupart du temps, en plus de garder l’historique des modifications sur chaque fichier source d’un projet, ils proposent des fonctionnalités destinées à faciliter le travail en collaboratif d’une équipe de développeurs. C’est pourquoi dans le monde du développement informatique, ils sont presque devenus incontournables.

#### Logiciel de gestion de version décentralisée

Un logiciel de gestion de version décentralisée, fournit des fonctionnalités classiques de versioning. Cependant, le terme décentralisé indique que le suivi des modifications est local à la machine sur laquelle il s’exécute. Effectivement il existe des logiciels de gestions de versions centralisé comme SVN. Dans ce cas, l’arborescence et l’historique du projet suivi sont stockés sur un serveur appelé dépôt. Les développeurs qui souhaitent récupérer et télécharger les dernières modifications du code source doivent alors se connecter au serveur. Puis ils doivent exécuter une commande afin de déclencher le téléchargement des dernières modifications. Vice versa, lorsqu’ils souhaitent partager des modifications, ils doivent se connecter au serveur et les uploader. Ce qu’il faut bien remarquer, c’est que seul le serveur conserve l’historique des modifications. Les développeurs se contentent de récupérer les dernières versions des sources et ne possèdent pas de copie locale de l’historique sur leurs machines.

Dans le cas de Git, le fonctionnement du logiciel est totalement différent. Celui-ci est un logiciel de gestion de versions décentralisé. Ainsi pour fonctionner, il enregistre l’historique des modifications directement dans le répertoire racine du projet qu’il traque. Pour ce faire, Git indexe les fichiers d'après leur somme de contrôle calculée avec la fonction SHA-1. Quand un fichier est modifié Git le détecte automatiquement. Il faut ensuite exécuter la commande commit pour valider ces modifications et mettre à jour l’historique du projet. Dès lors les deux versions du fichier sont stockées sur le disque de la machine. Une première version est directement visible dans l’arborescence du projet. C’est celle qui vient d’être modifiée. Tandis que l’ancienne version est stockée dans le répertoire caché qu’utilise Git pour l’historique. Ce répertoire est nommé .git. Ainsi chaque développeur possède un historique des modifications du projet directement sur sa machine. De cette manière chaque développeur peut à tout moment revenir en arrière si une des spécifications qu’il a implémentée pose problème. Et pour ce faire, nul besoin de se connecter à un serveur distant comme c’est le cas avec des logiciels de gestion de versions gestion centralisé, Git utilise directement l’historique local.

Bien entendu, Git possède également des fonctionnalités permettant de mutualiser du code source. On parle de logiciel de gestion de version décentralisé distribué. Avec ce type d’architecture, les modifications peuvent échangées entre développeurs à la manière d’un réseau peer to peer en partageant l’historique du projet. Cependant, ce type d’infrastructure peut vite devenir ingérable avec une équipe composée de nombreux développeurs. De ce fait, les logiciels de gestion de versions distribuées utilisent en général un point de rencontre afin de mutualiser le code. Ce point de rencontre est matérialisé par un serveur. Celui-ci stocke uniquement l’historique des modifications d’un projet. Bien entendu, l’historique permet de reconstituer l’arborescence du projet en intégralité et dans sa toute dernière version. Dès lors, à chaque fois qu’un développeur souhaite partager ses modifications, il lui suffit de partager son historique sur le serveur. Pour ce faire il utilise la commande push. Dans le cas où il souhaite récupérer des modifications depuis le serveur, il utilise la commande pull.

#### Présentation de Github

A l’origine Git a été développé par Linus Torvalds le créateur de Linux. C’est pourquoi il s’utilise principalement OS. Cependant, pour les utilisateurs de Windows des solutions sont disponibles. Il est possible d’installer un programme qui émule des commandes Unix sous Windows et qui par la même occasion installe Git. Cependant, le recours à un émulateur pose souvent des problèmes de performances, et il peut être agréable d’utiliser un programme avec une interface graphique plutôt qu’en lignes de commandes.

Face à ce constat, le site web GitHub propose différents outils. Tout d’abord il s’agit d’un service web permettant d’héberger des projets Git. Ainsi le code source d’un programme peut être partagé par toute une équipe de développement. De plus, pour les utilisateurs sous Windows, GitHub propose un programme client permettant d’utiliser Git sur sa machine. Ce programme fournit à la fois l’interface en ligne de commande habituellement utilisée sous Git, mais aussi une interface graphique permettant de gérer ses projets de manière intuitive (Figure 8).

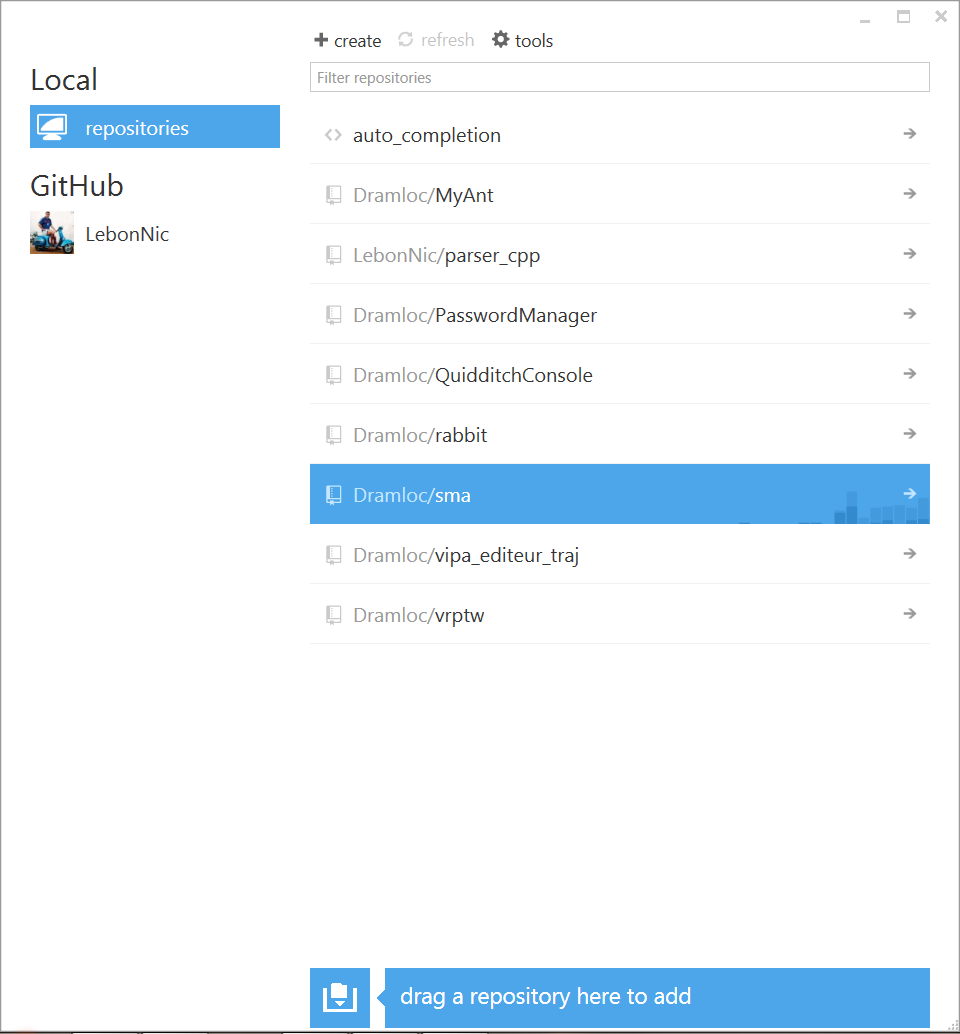


Figure - Interface du client GitHub

A cela, il faut également rajouter une add-on qui peut être installée sous Visual Studio, l’IDE de Microsoft et qui facilite grandement le versioning d’un programme.

Dans le cadre du projet de SMA, c’est cet outil de gestion de version qui a été utilisé. Le Git du projet est hébergé sur GitHub et la gestion de modification est effectuée directement depuis l’IDE grâce à l’add-on de Visual Studio.

### 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 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 générant des programmes moins portables 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 et 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 Studio 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.

### 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.

## Analyse

Dans cette partie, 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 la résolution des différents problèmes purement algorithmiques du projet.

### 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.

#### 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 objet 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 (Figure 9).



Figure 9 - 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 flot d’exécution du programme. C’est-à-dire qu’elles n’ont pas de comportement à proprement parler 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.

#### 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 (Figure 10).

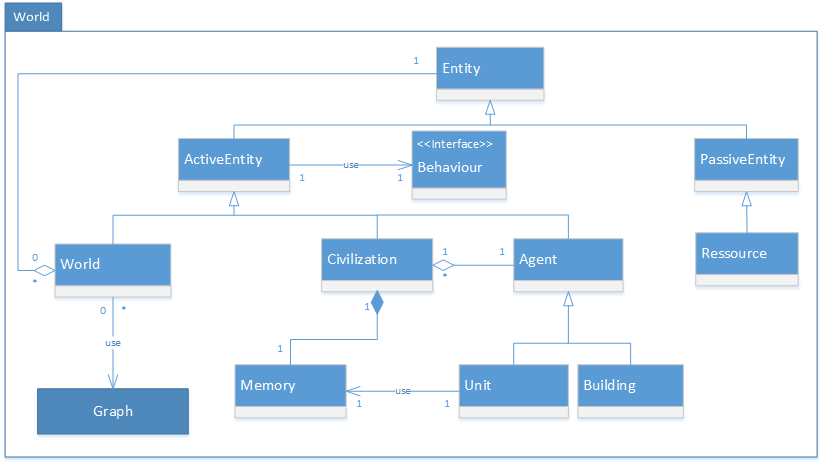


Figure 10 - Package World du simulateur de civilisations

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.

#### 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 (Figure 11).

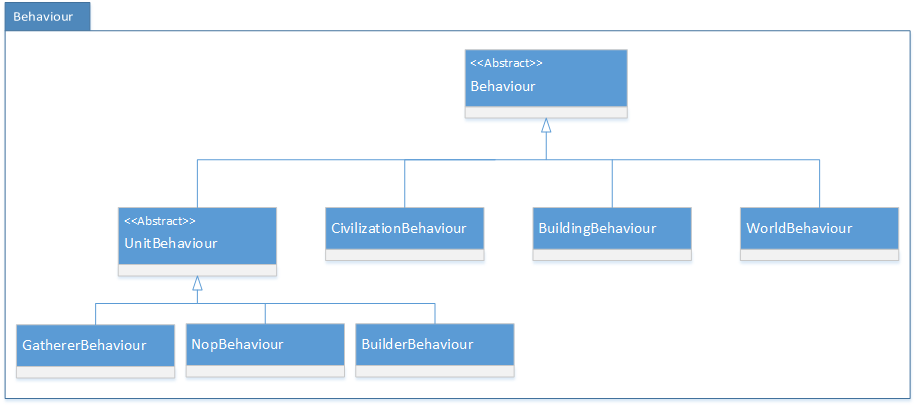


Figure 11 - Package Behaviour du simulateur de civilisations

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 flot 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, GathererBehaviour 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.

#### Gestion du système d'information géographique

Un point du package World n’a pas encore été abordé et pourtant il est au cœur de la simulation. Il s’agit de la manière avec laquelle sont référencées les entités dans l’espace. On parle de système d’information géographique. Dans les cas présentés ci-dessous, les modèles proposés sont décrits en deux dimensions, mais les principes restent les mêmes pour des cas avec plus de degrés de liberté.

C

A

E

D

B

SIG sous forme de Grille

SIG lié aux entités

A

B

C

D

E

Figure - Deux approches possibles pour gérer le système d'information géographique

Plusieurs approches sont possibles afin de modéliser l’espace dans lequel évoluent les entités. Une première solution consiste à mémoriser la position des entités dans une grille (Figure 12 – SIG sous forme de grille). Les entités sont alors référencées par l’index de ligne et l’index de colonne de la cellule dans laquelle elles se trouvent. Ce genre de représentation est proche de ce qui se fait dans le cadre des automates cellulaires. C’est une approche envisageable pour des cas où la simulation contient un grand nombre d’entités et pour des cas où le temps d’exécution est privilégié par rapport à l’utilisation de la mémoire. Ainsi l’accès au contenu d’une cellule de la grille se fait en puisqu’il suffit de spécifier deux indices et pour directement l’atteindre.

La deuxième approche possible pour gérer le système d’information géographique du programme est de stocker les coordonnées d’une entité directement dans la structure de cette dernière (Figure 12 – SIG lié aux entités). Cette approche peut être intéressante dans le cas où l’utilisation de la mémoire est restreinte. En contrepartie, l’accès aux données sera moins performant. Par exemple, si l’on souhaite savoir quelle entité occupe la position, il est nécessaire de parcourir tous les conteneurs du programme qui référencent les entités afin de rechercher celle qui possède ces coordonnées. Dans le pire des cas, il faudra parcourir toutes les entités pour trouver celle recherchée.

Dans le cas du SMA de ce projet les deux approches ci-dessus ont été utilisées. Tout d’abord, la classe Entity du package World contient des attributs qui référencent son positionnement géographique. De ce fait, chaque entité du SMA mémorise sa propre position au sein même de sa structure. Cependant, cette approche pose des problèmes de performances à certains au moment de l’exécution. En effet, au cours de la simulation, les agents et en particulier les unités vont se déplacer et rechercher des ressources. Au cours de cette phase de recherche, ils vont devoir observer leur environnement et mémoriser ce qui les entoure. Au moment où une unité va devoir interroger le SIG du monde pour savoir quelle entité se situe à proximité, la recherche risque d’être couteuse en termes en termes de complexité. Dans le cas des unités de ce SMA, c’est surtout la position des ressources qui sera demandée. C’est pourquoi, dans la classe World, une grille de ressources a été ajoutée de manière à réduire la complexité de ces accès.

Le monde du simulateur pris dans globalité est donc un espace à deux dimensions, un plan, sur lequel les entités sont référencées en utilisant le SIG défini précédemment. Pour les ressources, leur localisation s’effectue grâce à une grille et pour les autres entités la position géographique peut être connue grâce à leurs attributs.

#### Déplacement des unités

La question du positionnement géographique des unités vient d’être traitée, cependant, le SIG mis en place ne permet aux unités de se déplacer de manière autonome dans l’environnement. Il permet simplement aux entités de se repérer les unes par rapport aux autres dans le monde.

Etant donné que ce SMA a pour objectif de simuler le comportement d’êtres humains en quête de développement, l’un des objectifs du projet était que les déplacements des unités soient le plus réalistes possibles. De manière naturelle, lorsqu’une personne doit se rendre d’un point A à un point B sur une carte, elle choisit le plus court chemin. Par conséquent, lorsqu’une unité se déplace dans l’environnement en direction d’un objectif, elle suit le plus court chemin de sa position actuelle jusqu’à son objectif. En termes de génie logiciel, cela se traduit par la nécessité de concevoir un modèle permettant de calculer des plus courts chemins sur une carte.

C’est pourquoi un nouveau package a été développé dans ce sens (Figure 13). L’objectif de ce composant logiciel est de fournir une interface aux unités afin de rendre leurs déplacements le plus réaliste possible. Face à ce genre de situation, une approche particulièrement efficace consiste à modéliser l’espace sous forme de graphe. En effet, dans la théorie de l’informatique et des mathématiques, c’est une structure de données qui fait l’objet de bon nombre de recherches. Ainsi de nombreux algorithmes permettant de résoudre des problèmes liés à la théorie des graphes sont disponibles dans la littérature. De ce fait, l’espace du simulateur de civilisation a également était modélisé sous forme de graphe.

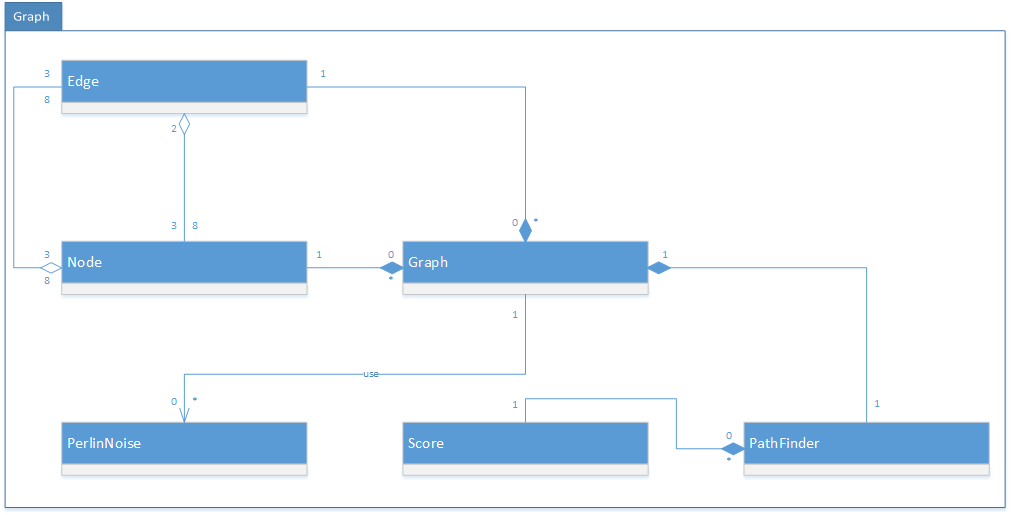


Figure - Package Graph du simulateur de civilisation

Plus précisément, le type de graphe utilisé dans le cadre de ce package est celui d’un graphe grille amélioré (Figure 14). Amélioré car un détail change par rapport aux grilles normales. Dans le graphe utilisé ici, chaque nœud possède 8 arrêtes, alors que dans un graphe grille seuls 4 arrêtes sont utilisées par nœud. Les arrêtes ont pour rôle de modéliser la connexité de deux sommets, c’est-à-dire la possibilité pour une unité de passer d’un sommet A à un sommet B en empruntant ce chemin. Ils ne sont donc pas dirigés car utilisables dans un sens ou dans un autre (de A vers B ou de B vers A).

Graphe chemin

Graphe grille

Un graphe grille est le produit cartésien de deux graphes chemins.

Graphe du package

Chaque nœud est lié à 8 arrêtes (exception faite des nœuds sur les extrémités).

Figure - Structures de graphe utilisées.

Dans le modèle proposé ici, on remarque que la classe Graph se compose d’une classe PathFinder. Celle-ci fournit au graphe une interface permettant de lancer un algorithme de recherche de plus court chemin entre deux de ses nœuds. Pour ce faire, la classe PathFinder implémente l’algorithme A\* (A star). Son fonctionnement sera détaillé dans une autre partie de ce rapport. La classe Score quant à elle, est un élément propre à cet algorithme et n’a pas de lien direct avec le graphe.

Un dernier détail n’a pas été expliqué dans ce package. Il s’agit du fait que la classe Graph utilise une classe PerlinNoise. Celle-ci fournit une interface afin de générer un bruit de Perlin. Ce bruit peut ensuite être utilisé par la classe Graph pour générer un relief aléatoire, mais réaliste, sur les nœuds qui la composent. En effet, chaque nœud utilise un système de coordonnées cartésiennes à trois dimensions pour être repéré dans l’espace. Cependant, cette fonctionnalité n’est pas utilisée par le SMA de ce projet, car tout se déroule dans un plan à deux dimensions. Comme ce package est assez indépendant du reste du projet, la fonctionnalité a tout de même était prise en compte en vue d’une potentielle réutilisabilité dans un autre cas.

#### Générateurs pseudo-aléatoires

Comme pour toute simulation stochastique à évènements discrets, différents types de générateurs sont nécessaires. Ce projet de SMA fait appel à plusieurs d’entre eux, rassemblés dans un unique package (Figure 15).

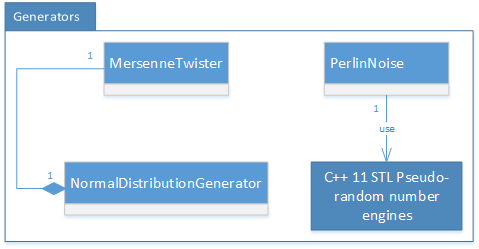


Figure - Package Generators du simulateur de civilisation

Tout d’abord, il nécessite un simple générateur de nombres pseudo-aléatoires. Pour remplir ce rôle une classe MersenneTwister a été développée. Elle implémente une interface qui permet de générer des nombres pseudo-aléatoires uniformément répartis sur 623 dimensions (pour des nombres de 32 bits) grâce à un algorithme basé sur un TGSFR (twisted generalised shift feedback register). Ce générateur porte le nom de Mersenne Twister et a été inventé par Makato Matsumoto et Takuji Nishimura en 1997. Dans la simulation il est utilisé par de nombreuses classes et composants, notamment par les classes comportementales des unités lorsqu’elles recherchent des ressources dans la mémoire de la civilisation. Cet algorithme sera détaillé plus loin dans le rapport.

Un second type de générateur utilisé par les unités permet de tirer des nombres suivant une loi normale paramétrable. Il est implémenté par la classe NormalDistributionGenerator et pour ce faire il utilise la technique mise au point par Box et Muller en 1958. Cette technique nécessite le tirage de deux nombres aléatoires pour obtenir des nombres distribués selon une loi normale. C’est pourquoi la classe NormalDistributionGenerator est composée d’un Mersenne Twister. Ce générateur de loi normale est utilisé par les unités au moment où elles cherchent un lieu pour construire de nouveaux bâtiments. Ce comportement sera mieux détaillé par la suite.

Enfin le dernier générateur utilisé au cours de la simulation permet de générer un environnement de manière aléatoire, sans que ce dernier soit totalement chaotique. Il s’agit d’un générateur de bruit de Perlin. Son fonctionnement et son utilisation sont détaillés dans la partie qui traite des problèmes algorithmiques du SMA. Son implémentation est réalisée à travers la classe PerlinNoise du package Generators et pour fonctionner, celle-ci utilise un Mersenne Twister intégré à la STL depuis l’arrivée du C++ 11.

### Problèmes algorithmiques

Le génie logiciel associé à une conception orientée objet permet de structurer correctement l’architecture d’un programme. Cependant, les traitements associés aux objets modélisés restent toujours à définir. Certains intègrent parmi leurs méthodes des traitements algorithmiques complexes qu’il est nécessaire de préciser. Dans ce sens, une première sous-partie a pour rôle d’expliquer le fonctionnement des IA présentes dans les entités actives. Une seconde sous-partie a pour rôle d’expliciter le fonctionnement de l’algorithme A\* utilisé par les unités afin de calculer le plus court chemin qui les sépare d’un objectif. Puis dans un dernier temps, une troisième sous-partie traite de la mise en œuvre d’algorithmes permettant de générer du relief dans un environnement à l’aide d’un bruit de Perlin. Elle explique également que la manière dont il possible d’utiliser ce bruit pour disposer des ressources aléatoirement sur une carte.

#### Stratégies des entités actives et des agents

Les entités actives se distinguent donc des entités passives par le fait qu’elles exécutent une séquence d’action à chaque pas de temps de la simulation. La partie précédente qui traite de génie logicielle a montré la nécessité de développer un package de classe comportementales pour ces unités. Néanmoins le comportement en lui-même de chaque entité n’a pas été expliqué. C’est le rôle de cette partie.

##### Comportement du monde

Le monde, au même titre que les civilisations ou les agents est une entité active du SMA. Il utilise donc une classe WorldBehaviour dans le but d’implémenter les actions qu’il doit exécuter. Le monde en réalité n’implémente pas de stratégie particulière, il sert surtout à transmettre le flot d’exécution du programme aux civilisations qu’il contient. Etant donné que le projet n’intègre pas de package Simulator à la manière du framework MAVIS, le monde est chargé d’organiser les évènements qu’il exécute de manière stochastique. Ainsi, son seul comportement est d’effectué un mélange du conteneur qu’il utilise pour référencer ses civilisations, puis il appelle la méthode run de chaque civilisation. Cette méthode transmet la main aux civilisations qui exécutent ensuite leur propre comportement.

##### Stratégie des civilisations

Dès lors que le monde fait appel à la méthode run d’une civilisation, celle-ci exécute son comportement défini dans la classe CivilizationBehaviour. Ce comportement a pour but de spécifier la gestion d’une équipe d’unités. Ainsi la civilisation a pour rôle d’affecter une tâche à chaque unité et de répartir l’ensemble des tâches de manière homogène sur toutes les unités. Il ne faut pas que toutes les unités se retrouvent à couper du bois par exemple si la civilisation manque d’or.

Pour mener à bien sa mission, la civilisation possède des listes (conteneurs) d’unités (Figure 16). Chaque liste correspond à une tâche que doivent effectuer les unités. L’objectif ensuite pour la civilisation est de veiller au bonne équilibre de ces listes au cours de la simulation. Ainsi, lorsque de nouvelles unités sont créées, elles sont dans un premier temps affectées à une liste qui référence toutes les unités dépourvues de travail (unités libres). La civilisation affecte ensuite les unités libres à l’une des listes de tâches qu’elle gère. Plus précisément ces listes sont aux nombres de trois et référencent les unités assignée à un travail de collecte. Il y a une liste pour les bucherons (collecte de bois), une liste pour les mineurs (collecte d’or), et une liste pour les cueilleurs (collecte de nourriture). Lorsqu’une civilisation affecte une unité à un travail, elle lui associe bien entendu la classe comportementale qui correspond à cette tâche.



Liste de bûcherons

Liste de mineurs

Liste de cueilleurs

Liste d’unités libres

Civilisation

Affectation d’une unité avec le comportement mineur

Figure - Gestion des listes d'unités d'une civilisation

La civilisation a également pour rôle de déclencher la construction d’un nouveau bâtiment. C’est elle qui gère les stocks de ressources. Aussi, lorsque les unités ont suffisamment amassé des ressources, la civilisation assigne une unité à une tâche de construction et consomme les ressources nécessaires (diminution des stocks de bois, d’or et de nourriture) à l’édification d’une nouvelle bâtisse. La construction est ensuite entièrement gérée par la classe comportementale de l’unité.

##### Intelligence artificielle des unités

La civilisation affecte donc ses unités à des tâches de récoltes ou de construction mais son rôle s’arrête là. C’est ensuite l’unité qui agit de manière autonome en fonction de la mission qu’elle doit remplir. A cet effet les unités disposent de trois classes comportementales dont le fonctionnement est expliqué ci-après. Une première classe peut cependant rapidement être exposée. Il s’agit de la classe NopBehaviour. L’abréviation Nop signifie no operation car la classe a pour rôle de ne rien faire faire à l’unité. Elle est surtout utilisée au moment de la création des unités, avant que la civilisation ne les affecte à une tâche plus complexe.

###### Collecte

La mission première des unités est donc d’explorer l’environnement afin de localiser et de collecter des ressources. Ce comportement est implémenté dans la classe GathererBehaviour. La stratégie adoptée dans ce comportement est à double fonction. Effectivement, elle permet à la fois aux unités d’explorer l’environnement si leur connaissance de celui est trop faible. Dans le même temps, lorsque les unités ont suffisamment acquis de savoir sur le monde qui les entoure, la collecte peut commencer de manière efficace.

Pour mettre en place cette stratégie, l’algorithme implémenté dans ce comportement utilise grandement l’aléatoire. Afin de comprendre pourquoi, il faut d’abord se pencher sur la manière dont les unités ont conscience de leur environnement. Cette conscience est le fruit d’un travail collaboratif qui permet de maintenir à jour une mémoire commune présente dans la civilisation. Trois cartes, ou plus exactement trois matrices sont disponibles dans cette mémoire. Il y a une matrice par type de ressources (or, bois, nourriture). Chaque cellule d’une matrice contient un booléen. Celui-ci indique la présence où l’absence d’une ressource. Chaque cellule est mise à jour par les unités de la civilisation lorsqu’elles explorent l’environnement. Pour ce faire, elles sont capables de marquer la présence d’une ressource dans la matrice correspondante, lorsqu’elles passent à proximité de celle-ci. Les unités ont bien entendu une portée limitée au-delà de laquelle elles ne sont pas capables d’identifier des ressources. Cette mutualisation des connaissances leur permet de profiter du travail accompli par d’autres unités lorsqu’elles se déplacent dans l’environnement.

Néanmoins, au commencement de la simulation, les unités ignorent tout de leur environnement. Il faut donc que le comportement de collecte intègre une tactique afin de gérer le cas où la mémoire commune des unités se trouve pauvre en informations. Ainsi l’algorithme de recherche de ressource procède en deux étapes. Au cours de la première, il va fouiller la carte de ressources qui intéresse l’unité en fonction de son rôle (cueilleur, mineur ou bucheron). Cette fouille s’effectue en tirant des points au hasard sur la carte de ressources. Si le point tiré fait référence à une ressource alors l’unité tente de trouver un chemin pour l’atteindre. Puis, dans le cas où la ressource est atteignable, l’unité commence à se déplacer dans sa direction. Dans le cas contraire, elle tire un nouveau point sur la carte et relance le traitement précédent. Au bout d’un certain nombre de tirages, si aucune ressources n’a pu être trouvée, l’unité stop sa recherche. La deuxième phase de l’algorithme se déclenche alors.

Au cours de cette seconde phase, l’unité va tenter de trouver de manière aléatoire un point de l’environnement qu’elle peut atteindre. Elle va donc tirer plusieurs points de manière aléatoire. Puis, pour chaque point, elle va tenter de trouver un chemin lui permettant de l’atteindre. Si le point est atteignable l’unité commence à se déplacer dans sa direction. Au cours de son déplacement elle va mémoriser ce qu’elle perçoit. De cette manière les cartes de ressources de sa civilisation vont être mises à jour. Si le point n’est pas joignable, alors l’unité en tire un nouveau et relance ce processus depuis le début. Bien entendu, après un certain nombre de tirage l’unité rend la main à sa civilisation et attend le prochain pas de temps pour reprendre son comportement de collecte de manière à ne pas bloquer tout le programme.

###### Construction

Lorsque les unités ont collecté suffisamment de ressource, l’une d’entre se voit affectée à une tâche de construction par la civilisation. Cette phase de construction est entièrement gérée par l’unité via la classe comportementale BuilderBehavior.

L’algorithme utilisé par cette classe fait appel à un générateur de loi normale. Ce générateur est utilisé par l’unité pour trouver un emplacement de construction suffisamment proche du premier bâtiment de la civilisation. De cette manière on simule l’expansion d’un village depuis son centre vers l’extérieur.

Pour obtenir ce comportement, l’algorithme commence par calculer une direction aléatoire. Il tire ensuite une paire de nombres aléatoires distribués selon une loi normale centrée. Puis grâce à des règles de trigonométries basiques, il calcule un couple de points de l’environnement selon la direction générée précédemment et dont l’écart au centre de la civilisation correspond à celui des nombres aléatoire par rapport au centre de la loi normale. L’écart type de la loi varie en fonction du nombre de bâtiments construits. Plus il y a de bâtiment et plus l’écart type croit (Figure 17). Cette croissance a pour effet d’augmenter l’éloignement des nouveaux bâtiments du centre de la civilisation.



Génération d’une direction aléatoire

Un angle θ est choisi aléatoirement entre 0 et π. Ainsi on obtient une direction aléatoire.

θ

Génération d’un écart selon une loi normale

Le long de la direction calculée précédemment, on tire deux points aléatoires distribués selon une loi normale.

p1

p2

Loi normale

Direction

Figure - Illustration de l'algorithme utilisé pour trouver un emplacement de construction

Ensuite, l’algorithme vérifie si l’emplacement obtenu grâce au calcul précédent n’est pas déjà occupé par une entité passive ou par un autre bâtiment. Puis si ce n’est pas le cas, l’unité calcul un chemin potentiel jusqu’à ce point. S’il est atteignable, elle s’y rend pour construire un bâtiment, si ce n’est pas le cas, elle relance le processus depuis le début. Ici encore, au bout d’un certain nombre de tentatives, l’unité rend la main à la civilisation de manière à ne pas bloquer l’exécution du programme tout entier.

Une fois que l’unité a atteint son emplacement de construction la création d’un bâtiment est lancée. Au cours de ce processus, l’unité reste immobile un certain moment. Puis le bâtiment apparait dans l’environnement. La création de 8 nouvelles unités accompagne celle d’un bâtiment. Ainsi la population de la civilisation augmente.

Dans les différents algorithmes expliqués précédemment, il a été omis de dire comment les unités faisaient pour trouver un chemin en direction d’un objectif. C’est le rôle de la partie suivante.

#### Algorithme de Pathfinding

Le pathfinding est un problème d’intelligence artificielle qui consiste à rechercher le plus court chemin possible reliant deux sommets d’un graphe. Dans ce projet il occupe une place important car c’est lui qui dicte aux unités leur manière de se déplacer. L’algorithme utilisé est pour ce SMA est le célèbre A\* développé par Peter Hart, Nils Nilsson et Bertram Raphael en 1968.

##### Introduction à l’algorithme A\*

La mise en place d’un algorithme permettant aux unités de se déplacer de manière crédible dans l’environnement a été un point essentiel du projet. Comme expliqué précédemment l’objectif était que les unités empruntent toujours le plus court chemin possible pour se rendre d’un point A à un point B (Figure 18). Ce constat a conduit au développement d’un package UML de classes dédiées à la recherche de plus courts chemins dans l’environnement. Ainsi, en parallèle du SIG qu’utilisent les entités pour se repérer, l’environnement est modélisé sous forme de graphe. Cette structure permet de mettre en œuvre des traitements algorithmiques complexes comme ceux qui permettent de rechercher des plus courts chemins.

A

B

B

A

Trajectoire irréaliste d’unité.

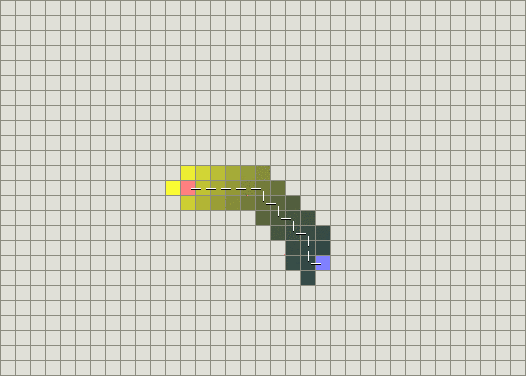
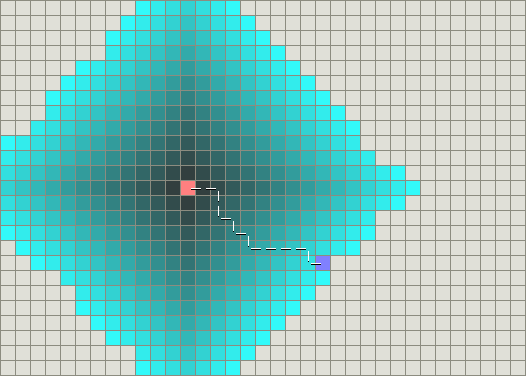
L’unité n’anticipe pas la présence d’un obstacle et effectue un détour.

Trajectoire réaliste d’unité.

L’unité voit l’obstacle et le contourne.

Figure - Trajectoires d'une unité.

La recherche de plus court chemin est problème récurrent dans la littérature qui traite de la théorie des graphes. L’un des plus connus est l’algorithme de Dijkstra mise au point par l’informaticien du même nom Edsger Dijkstra en 1959. C’est un algorithme qui parcourt les nœuds du graphe en explorant le nœud le plus proche du point de départ à chaque itération. Il garantit de trouver le chemin optimal entre deux sommets du graphe, mais sa complexité est polynomiale (Figure 19 – Dijkstra).



Dijkstra

Recherche best-first

Exécution d’un Dijkstra dans un cas simple. Les dalles en bleue sont celles explorées par l’algorithme. Plus leur teinte est claire et plus le coût pour les atteindre est élevé.

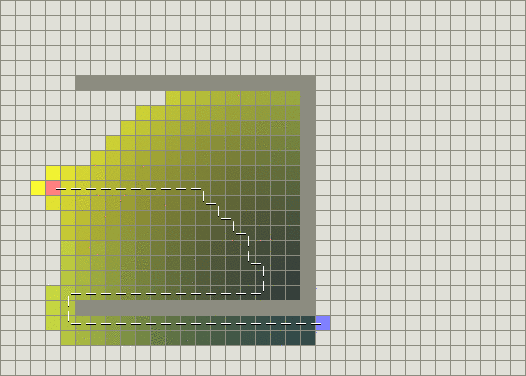
Exécution d’une recherche best-first dans un cas simple. Les dalles en jaune sont celles explorées par l’algorithme. Plus leur teinte est claire et plus l’estimation de leur distance à l’objectif est élevée.

Figure - Comparaison d'algorithmes de recherche de plus courts chemins

Face à ce problème de complexité, d’autres algorithmes ont été développés. L’un des plus connus est l’algorithme de recherche best-first (Figure 19 – Recherche best-first). Celui-ci parcourt les sommets du graphe en explorant le nœud qui semble le plus « prometteur » à un instant donné selon une heuristique d’estimation (algorithme glouton). Cette heuristique a pour but de prédire la distance qui sépare un sommet du graphe de l’objectif à atteindre. Ainsi contrairement à Dijkstra ce ne sont pas les sommets les plus proches du point de départ qui sont traités en priorité, mais les sommets les plus proches de l’objectif. Cependant, contrairement à Dijkstra, l’algorithme de recherche best-first ne garantit pas de trouver le chemin le plus court entre deux sommets du graphe, mais sa complexité est plus faible.

Du fait que ce projet de SMA propose une visualisation en temps réel de la simulation, l’algorithme effectuant le calcul des plus courts chemins se devait d’avoir une complexité limitée. Ainsi même si Dijkstra fournit à coup sûr un résultat optimal, sa trop grande complexité le rend inutilisable pour un programme qui propose une visualisation interactive.

L’algorithme de recherche best-first quant à lui, garantit une complexité plus faible, mais les résultats fournis laissent parfois à désirer (Figure 20). Ceux-ci ne collent pas avec la contrainte du cahier des charges qui stipule que les unités doivent se déplacer de manière réaliste.



Recherche best-first avec obstacle

Dans le cas où un obstacle est présent sur la route de l’unité, l’algorithme calcul un chemin qui la pousse à faire un long détour.

Figure - Cas mal géré par l'algorithme de recherche best-first

L’algorithme A\* combine les qualités des deux algorithmes précédents afin de fournir des résultats parmi les meilleurs possibles. Ainsi il est souvent utilisé dans des applications temps réel, privilégiant la vitesse de calcul à l’exactitude des résultats. C’est cet algorithme qui a été implémenté pour répondre aux besoins de ce SMA.

##### L’algorithme A\*

Le secret du A\* réside dans l’heuristique d’évaluation des nœuds qu’il utilise. Celle-ci combine les informations utilisées par Dijkstra et par la recherche best-first. Ainsi l’heuristique d’évaluation du A\* se compose de la somme de deux fonctions :

Une fonction donne le coût exact d’un sommet depuis l’origine du chemin calculé. Ce coût est en général calculé en effectuant la somme du poids des arrêtes à franchir pour l’atteindre. Si le poids associé à chaque arrête représente la distance euclidienne qui sépare deux nœuds, alors le coût évalué représente la distance qui sépare le sommet traité de l’origine du chemin. C’est l’approche qui a été utilisée dans la version du A\* de ce SMA. En effet, la fonction peut également être calculée en intégrant des coûts proportionnels à la difficulté de franchissement d’une arrête. Ainsi il possible de modéliser différents types de terrains plus ou moins difficiles à passer, comme des hautes herbes ou des forêts par exemple, mais cette approche n’est pas utilisée ici.

La fonction quant à elle est une heuristique admissible qui permet d’estimer le coût du trajet séparant le sommet en cours de traitement de l’objectif à atteindre. Une heuristique admissible est une fonction qui ne surestime le coût d’un chemin pour atteindre l’objectif. Dans le cas du SMA de ce projet, la fonction calcule la distance euclidienne du sommet traité à l’objectif. Cela revient à estimer la distance à vol d’oiseau (sans passer par les arrêtes) du sommet en cours de traitement à l’objectif. Ici encore, cette fonction peut être complexifiée en intégrant au calcul de h(n) la difficulté moyenne pour franchir une arrête.

Néanmoins, la difficulté de franchissement d’un de terrain n’a pas été prise en compte dans le SMA du projet. En effet les unités se déplacent à la même vitesse peu importe le type de terrain qu’elles franchissent, donc inutile d’intégrer cette spécification au A\*. La seule contrainte de déplacement des unités est qu’elles ne peuvent pas passer par certaines cellules du SMA occupées par des entités considérées comme infranchissables. Dans le modèle adopté pour ce projet, les seules entités considérées comme infranchissables sont les ressources de bois autrement dit les arbres.

Le A\* explore donc les sommets du graphe en fonction de leur score retourné par la fonction. Ainsi le comportement de l’algorithme est paramétrable suivant la qualité des résultats que l’on souhaite obtenir. En effet, si la fonction n’est pas prise en compte au moment du calcul de, l’algorithme parcourt en priorité les sommets les plus proches de l’origine et se transforme en Dijkstra. Vice versa, si la fonction est ignorée, le A\* explore en priorité les sommets dont la prédiction de distance à l’objectif est la plus faible. De ce fait, il se transforme en une simple recherche best-first. Ce paramétrage est intéressant car il permet d’ajuster la complexité de l’algorithme en fonction des contraintes du programme. Une des difficultés de paramétrage de l’algorithme est donc de trouver un bon compromis entre et. Il faut également veiller à utiliser les mêmes unités et les mêmes échelles pour le calcul de ces deux fonctions sinon l’une des deux peut prendre le pas sur l’autre et influencer le A\* de manière anormale aux yeux de l’utilisateur.

Le cœur de l’algorithme vient donc d’être dévoilé, mais son implémentation reste toujours à expliquer. C’est l’objectif de la partie suivante qui détail également les structures de données utilisées dans l’implémentation proposée.

##### Implémentation

Le A\* pris en dehors de toute implémentation réelle est assez simple à comprendre. Il se contente de gérer deux ensembles appelés ouvert et fermé. L’ensemble ouvert contient les sommets dont l’exploration a été planifiée. L’ensemble fermé quant à lui mémorise les sommets déjà traité par le A\*. Initialement l’ensemble fermé est vide, et l’ensemble ouvert contient le sommet de départ.

A chaque itération l’algorithme va alors extraire, parmi les sommets de l’ensemble ouvert, celui dont l’évaluation de l’heuristique a donné le plus petit résultat. Dans le jargon on parle de sommet ayant le plus petit score. L’algorithme va ensuite ajouter le sommet traité à l’ensemble fermé, puis il va planifier l’exploration de ses sommets voisins. Pour ce faire, il va parcourir l’ensemble des voisins et vérifier qu’ils n’ont pas déjà été traités, c’est-à-dire qu’ils ne sont pas déjà présents dans l’ensemble fermé. Si ce n’est pas le cas, l’algorithme va évaluer le voisin en cours de traitement à l’aide de la fonction. Si le voisin n’a encore jamais été scanné par l’algorithme ses informations sont mises à jours. On calcule et mémorise alors l’évaluation de ce sommet par la fonction, ainsi que le sommet qui a permis de l’atteindre (père). Dans le cas où l’exploration du voisin avait déjà été planifiée (déjà présent dans l’ensemble ouvert) mais que la fonction donne un meilleur résultat au cours de cette évaluation, les informations associées au voisin sont mises à jour.

Une précision concernant la gestion des obstacles est à apporter. Comme il a été expliqué précédemment les unités ne peuvent franchir les arbres présents dans le SMA car ceux-ci constituent un obstacle. Dans l’algorithme cela se traduit par le fait qu’au moment où l’on traite les voisins du sommet en cours, les nœuds considérés comme des obstacles sont marqués à l’aide d’un booléen. De cette façon les nœuds considérés comme des obstacles sont ignorés, et leur exploration ne se fera jamais. Par conséquent un nœud obstacle ne pourra jamais se retrouver dans un chemin calculé.

Algorithme 1 : A\*

Entrée :

* depart : sommet de départ du chemin à calculer.
* objectif : sommet à atteindre grâce au chemin calculé.

**Procedure** A\* (depart, objectif)

closedset := Ø;

openset := {depart};

pere := Ø;

objectifAtteint := faux;

g\_score[depart] := 0;

f\_score[depart] := g\_score[depart] + h(depart, objectif);

**Tant que** objectifAtteint = faux **et** que openset ≠ Ø **faire** **:**

Extraire le sommet u ayant le f score le plus faible parmi les sommets de l'openset;

**Si** u = objectif **alors** :

objectifAtteint := vrai;

**Sinon**

Ajouter u au closedset;

**Pour** **chaque** voisin v de u **faire :**

**Si** v ∉ closedset **alors :**

tentative\_g\_score := g\_score[u] + cout\_mouvement(u, v);

**Si** v ∉ openset ou que tentative\_g\_score < g\_score[v] **alors:**

pere[v] := u;

g\_score[v] := tentative\_g\_score;

f\_score[v] := g\_score[neighbor] + h(v, objectif)

**Si** v ∉ openset **alors :**

Ajouter v à l'openset;

**Fsi**;

**Fsi**;

**Fsi**;

**Fin**;

**Fsi**;

**Fin**;

**Si** objectifAtteint **alors:**

Lancer la reconstruction du chemin grâce au tableau pere en partant du sommet objectif;

**Fsi**;

**Fin** A\*;

Dans l’implémentation réelle en C++, les ensembles ouvert et fermé sont chacun matérialisés par un vecteur de la STL. Ce choix a été retenu pour plusieurs raisons. Tout d’abord l’ensemble ouvert doit être tenu à jour à la manière d’une file de priorité. C’est-à-dire que les éléments ayant le plus petit score doivent se situer au début de file de manière à pouvoir être extrait en premier. Ce constat oriente dans un premier le choix du conteneur vers la classe priority\_queue de la STL. Cependant cette classe n’est pas prévue pour qu’on puisse y rechercher un élément, ce qui pose problème lorsque l’on teste l’appartenance d’un sommet à l’ensemble ouvert. De ce fait, un vecteur a été choisi pour remplir le rôle d’ensemble ouvert. Il est ensuite mis sous forme de tas, puis un algorithme de tri par tas lui est appliqué afin de faire remonter le sommet ayant le plus petit score en tête. L’ensemble fermé est également un vecteur de la STL.

Le A\* permet donc aux unités du SMA de calculer leur chemin, la plupart du temps, de manière optimale pour se rendre à un endroit de l’environnement. De plus sa faible complexité permet à l’application tout entière de ne pas être trop ralentie, ce qui permet une visualisation en temps réelle de la simulation. Il reste un dernier point algorithmique à aborder concernant ce SMA, il s’agit de la génération aléatoire de l’environnement. La partie suivante traite de ce problème.

#### 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 aléatoires 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.

#### 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.

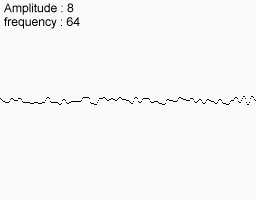
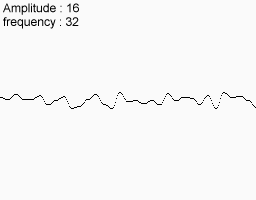
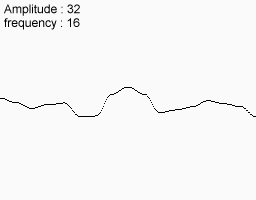
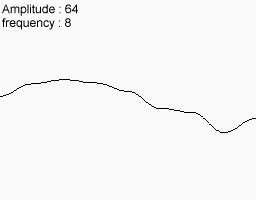
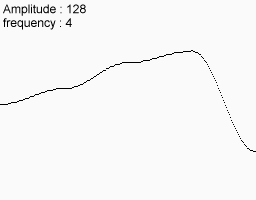


Figure 21 - Fonctions de bruit pour différentes octaves

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 22).

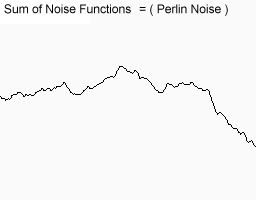


Figure 22 - Somme des octaves

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.

#### 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).

#### 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 23). 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.

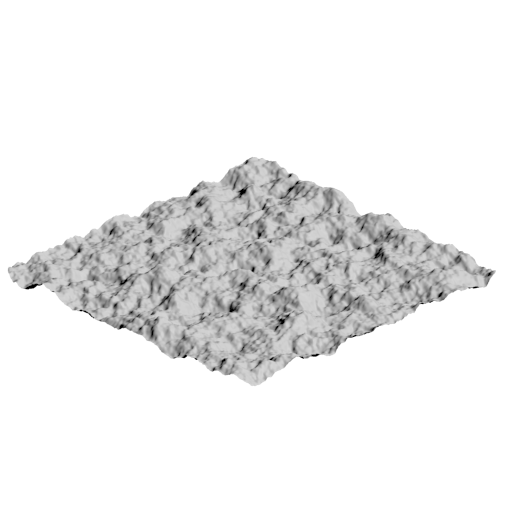
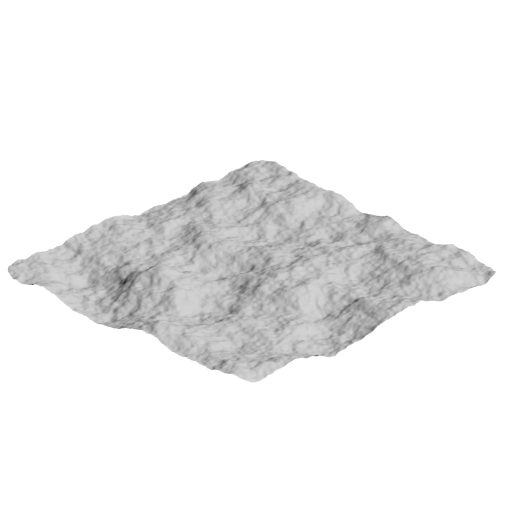


Figure 23 - Génération avec coefficient d'échelle de 0.25 (à gauche) et 0.50 (à droite)

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).

Tableau - 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 |

La génération aléatoire est donc un point essentiel du programme. Il permet tester le comportement des unités dans divers conditions d’abondance de ressources en ne faisant varier que quelques paramètres. C’est donc un outil essentiel pour cette simulation.

Maintenant que l’ensemble des points algorithmique du SMA ont fini d’être présentés, il est temps d’entamer la partie résultats qui aborde l’exécution du programme plus en détails.

# Résultats

De par toutes les problématiques qu’elle a pu soulever, la conception de ce SMA fut très enrichissante. Néanmoins le programme ne présente pas de réel intérêt scientifique. C’est pourquoi il n’a pas été choisi de faire de lui faire subir toute la batterie de tests habituellement nécessaires afin de valider la qualité des résultats d’une simulation. En conséquence cette partie résultat se contente de fournir différentes captures d’écran du programme en cours d’exécution ainsi qu’un jeu de tests rassemblant différentes données sur l’évolution d’une civilisation.

Voici donc à quoi ressemble l’environnement dans son ensemble. Les zones sombres sont celles que les unités n’ont pas encore explorées.



Figure - Capture d'écran du programme

Si l’on zoom sur un endroit de façon à avoir un aperçu plus précis des choses voici ce que l’on peut découvrir.



Unité

Nourriture

Or

Bois

Figure - Zoom sur une zone du monde

Après cette brève présentation de l’interface, il est temps de passer l’explication du test effectué. Pour cet essai, une sortie texte a été ajoutée au programme. De cette façon, tous les 100 pas de temps, ce dernier met à jour un fichier texte avec des informations sur le déroulement de la simulation. Ainsi il est possible de suivre l’évolution de divers variables comme la quantité de bois récolté ou la taille de la population par exemple. Pour ce test, une seule civilisation a été observée. Au niveau du paramétrage du programme, chaque unité de la civilisation possède une capacité de récolte limitée. Celle-ci a été fixée à 20, c’est-à-dire qu’à chaque pas de temps, elles sont capables de ramasser 20 unités de bois, d’or ou de nourritures. Le champ de vision des unités est limité à 10 cellules. C’est-à-dire que lorsqu’une unité se déplace elle ne voit que ce qui l’entoure dans un carré de 10 cases de côté sur la grille du SMA. En ce qui concerne l’algorithme de collecte et d’exploration du monde, le nombre de points aléatoires tirés sur la carte est limité à 100 par unité. Concernant, le coût de fabrication d’un bâtiment pour une civilisation, il est fixé à 5000 unités de bois, d’or et de nourriture. Enfin, en ce qui concerne la quantité de ressources disposées dans l’environnement, la densité d’or et de nourriture est fixée à 0,005 tandis que celle du bois est à 0,15. Le bois est donc la ressource majoritairement présente pour ce test. Voici donc les résultats de ce test (Figure 26).

Figure - Evolution des quantités de ressources

Après 10000 pas de temps exécutés par le monde, l’environnement ne contient plus de ressources. Comme prévu, le bois arrive en tête des ressources les plus abondantes dans l’environnement. Il est tellement abondant que la courbe qui modélise son évolution est lisse. Ce n’est pas le cas pour la nourriture et l’or, on peut voir un certains nombres de variations. Ces variations ne sont pas dues au fait que les données soient bruitées. En réalité, elles sont causées par l’utilisation qu’en fait la civilisation. En effet, lorsque celle-ci remarque qu’elle a suffisamment de ressources, elle les consomme pour créer de nouveaux bâtiments et de nouvelles unités. C’est ce qui cause ces variations et comme ces deux ressources sont moins abondantes, leurs fluctuations sont plus visibles sur les courbes.

L’évolution de la population d’unités est également intéressante à observer. On remarque que celle-ci met un certains à croitre jusqu’à 2000 pas de temps. Puis elle connait une soudaine hausse tant que des ressources sont disponibles. Cela est dû au fait que plus il y a d’unités, plus les stocks de ressources sont abondants et plus la civilisation peut s’agrandir. La première ressource qui vient à manquer est l’or. Autour de 6000 pas de temps, la population cesse de croitre et l’on remarque ce chiffre coïncide avec le moment où la courbe d’or commence à stagner.

Figure - Evolution de la population d'unités

Difficile de commenter ces résultats car le nombre d’essais est trop faible. On peut cependant dire qu’avec une IA assez basique, les unités arrivent tout de même à collecter énormément de ressources et qu’elles se développent rapidement. Pour complexifier un peu les choses on pourrait intégrer de nouvelles contraintes à la simulation, comme le fait que les unités consomment en permanences des ressources. On pourrait aussi songer à améliorer le programme en ajoutant toute une gamme de comportements martiaux aux unités. Ainsi, elles seraient capables de se défendre et d’attaquer face à une autre civilisation.

# Conclusion

Ce projet par biens des aspects fut très enrichissant. Tout d’abord, il nous a permis de découvrir un domaine de la simulation que nous ne connaissions pas : celui des SMA. Dans un premier nous avons donc dû faire des recherches bibliographiques afin de découvrir ce sujet. Cette démarche est proche de celle adoptée dans la recherche lorsque l’on conçoit une nouvelle expérience. En ce sens, ce fut donc intéressant de suivre cette approche.

Une fois que le sujet fut mieux maîtrisé nous avons dû nous pencher sur la phase de conception. Celle-ci a nécessité la rédaction d’un mini cahier des charges, mais surtout, elle a permis de mettre en œuvre toutes les compétences de génie logicielle acquises au cours de cette deuxième d’école d’ingénieur à l’ISIMA. Il a fallu apprendre à utiliser des design patterns mais aussi à les modifier afin de les ajuster à nos besoins.

Enfin le dernier point positif de ce projet est qu’il nous a permis de confronter à la conception d’intelligences artificielles, un domaine qui n’est pas simple à maitriser en informatique. De la même manière que pour le SMA il a fallu étudier ce qui existait déjà dans la littérature, comme pour l’implémentation du A\* par exemple. Mais il a aussi fallu développer nos propres algorithmes lorsque ce fut nécessaire, comme avec l’implémentation des IA de nos entités actives.

# Références bibliographiques

André M.C. Campos, David R.C. Hill, “An Agent Based Framework for Visual-Interactive Ecosystem Simulations“, SCS Transactions on Simulation, 15(4) :139-152, Décembre 1998.

Amit Patel, Amit’s A\* Pages, http://theory.stanford.edu/~amitp/GameProgramming/, Février 2014.

# Annexes

Le projet est accessible via un git à l’adresse suivante : https://github.com/Dramloc/sma.

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