



FACULTÉ
DES SCIENCES
DE MONTPELLIER

Master 1 Informatique
Parcours IMAGINA
UE HMIN201 TER

>>>

Rapport de TER sur MetaCiv

Arnaud Sanchez, Dimitri Maronnier et Thomas Rampin

Encadrants: Mr Ferber, François Suro et Tiberiu Stratulat

22 mai 2017

Faculté des Sciences
Campus Triolet
Place Eugène Bataillon
34095 Montpellier
France
Tél. +33 (0)4 67 14 36 75

<http://sciences.edu.umontpellier.fr>

Remerciements

Nous remercions particulièrement Monsieur Ferber, François Suro ainsi que Tiberiu Stratulat pour leur accompagnement tout au long de ce projet. Leur encadrement et les différentes réunions passées nous ont permis de mener à bien nos tâches. Leur disponibilité nous a permis de converger vers la solution plus vite lorsque nous bloquons sur un problème.

Nous tenons à remercier Monsieur Bourreau, et tous les intervenants de l'UE de conduite de projet, pour nous avoir apporter les outils et les connaissances de gestion de projet, qui nous ont permis de réaliser ce projet en ayant une organisation rigoureuse.

Nous remercions également les personnes, qui de près ou de loin, nous ont aidés à réussir dans notre projet.

Sommaire

Titre	1
Remerciements	2
Sommaire	3
1 Introduction	6
2 Glossaire	7
2.1 Partie programmation des sondes et éditeur de plan	7
2.1.1 Plan :	7
2.1.2 Cogniton :	7
2.1.3 Attribut :	7
2.1.4 Objet :	7
2.2 Partie moteur 3D	7
2.2.1 Vertex Shader :	7
2.2.2 Vertex fetching :	7
2.2.3 Vertex Attributes :	7
2.2.4 Tessellation :	7
2.2.5 Tessellation Control Shaders :	7
2.2.6 Tessellation Engine :	8
2.2.7 Tessellation Evaluation Shaders :	8
2.2.8 Geometry Shaders :	8
2.2.9 Rasterization :	8
2.2.10 Fragment Shaders :	8
2.2.11 Framebuffer :	8
2.2.12 Model-View Matrice :	8
2.2.13 Projection Matrice :	8
2.2.14 Perspective Matrice :	8
3 MetaCiv, le point de départ et nos objectifs	9
3.1 La mise en route	9
3.1.1 L'éditeur de plan	9
3.1.2 Les sondes et graphes par défaut	10
3.1.3 L'apparition du moteur 3D	11
3.1.4 Objectifs et organisation	11
3.1.4.1 Objectifs	11
3.1.4.2 Organisation	12
4 L'éditeur de plan	12
4.1 Design de l'éditeur de plan	12
4.1.1 Analyse de l'éditeur original	12
4.1.1.1 Fonctionnement d'un plan	13
4.1.1.2 Faiblesses de l'éditeur existant	13
4.1.2 Création du nouvel éditeur	15
4.1.2.1 Spécifications	15
4.1.2.2 Création de la maquette	15
4.2 Prototype de l'éditeur	17

4.2.1	Implémentation d'un bloc	17
4.2.1.1	Description d'un bloc	17
4.2.1.2	Effet Magnétique	18
4.2.1.3	Lier des blocs entre eux	20
4.2.1.4	Particularités des blocs logiques	21
4.2.2	Fenêtre éditeur	22
4.2.2.1	Panneau édition	22
4.2.2.2	Panneau de la liste des actions	23
4.2.2.3	Création d'un nouveau bloc	24
4.3	Intégration dans Metaciv	26
4.3.1	Ajout des actions aux blocs	26
4.3.2	Modifications sur le plan	28
4.3.2.1	Ajout d'action dans le plan	28
4.3.2.2	Suppresion d'actions dans le plan	30
4.3.2.3	Ajout des paramètres interactifs sur les blocs	30
4.3.2.4	Lecture d'un plan existant	31
4.4	Conclusion de la partie éditeur de plan	32
5	Les sondes et les graphes par défaut	32
5.1	Introduction aux graphes	33
5.2	Prise en main via l'interface utilisateur	33
5.2.1	Lancement de la fenêtre de graphes	33
5.2.1.1	Ajout et configuration de sondes	34
5.3	Programmation des sondes par défaut	37
5.3.1	La création des onglets	38
5.3.2	Onglet par défaut simple	38
5.3.3	Onglet par défaut complet	40
5.3.4	Onglet personnel	44
5.3.5	Sauvegarde de la configuration par défaut	44
5.4	Conclusion de la partie des sondes par défaut	45
6	La vue 3D	46
6.1	Introduction à la 3D	46
6.1.1	Intérêt de la vue 3D	46
6.1.2	Présentation de la librairie LWJGL	46
6.1.3	OpenGL 3.3/4.1	47
6.2	MetaCiv2D vers MetaCiv3D	47
6.2.1	Problématique	47
6.2.2	Redirection de la sortie TKDefaultViewer	48
6.2.3	Génération du terrain	49
6.2.4	Les tortues	57
6.2.5	Les facilities	59
6.2.6	Les debugs Messages	60
6.3	Structure du moteur graphique	61
6.3.1	Gestion des shaders	61
6.3.2	Modèles 3D	62
6.3.3	Gestion des rendus	66
6.3.4	La caméra	67
6.4	Rendu graphique	69
6.4.1	Rendu du terrain	69
6.4.2	Rendu de la mer	72
6.4.3	Physically Based Rendering	76
6.4.4	Post-Processing	79

6.4.4.1	Bloom effect	80
6.4.4.2	Depth of field	80
6.4.4.3	Contrast Balance	81
6.4.5	Amélioration possibles	81
6.4.6	Tutoriel	81
6.5	Contrôle de la vue 3D	81
6.6	Contrôle du terrain	81
6.7	Logique des fichiers pour Tortues/Facilities/Terrain	83
6.8	Conclusion de la partie 3D	83
7	Conclusion	83
	Bibliographie	84

1 Introduction

MetaCiv est une simulation de civilisations reposant sur un système multi agents. Un système multi agents permet de simuler en informatique les interactions propre aux sociétés animales et humaines (par exemple, les fourmis ou les termites). Ces agents sont des entités appartenant à un environnement et interagissant entre eux via différents modes de communications. Ils peuvent s'organiser sous forme de groupes (famille, clan, tribu, etc...) et de rôles (métier, statut, classe, etc..), ils sont autonomes grâce aux communications, aux perceptions et aux représentations. Tout ces différents mécanismes permettent d'ouvrir de nouvelles perspectives à ces individus, ils peuvent alors entrevoir des actions qu'ils n'auraient pas pu faire seuls.

Les systèmes multi agents font l'objet de recherches depuis longtemps, notamment en intelligence artificielle. Grâce à leur modélisation de sociétés, ils ont un large spectre d'applications, cela va des jeux vidéos à la création de foule pour le cinéma, en passant par le trading automatique dans la finance. Dans le cadre de ce travail d'études et de recherches l'application de ces systèmes est utilisée pour modéliser et simuler des phénomènes complexes.

Notre TER consistait à développer un nouvel éditeur de plan qui rendrait la création et la modification des plans beaucoup plus intuitives et simples pour l'utilisateur final. Nous devions aussi faciliter l'utilisation des sondes et des graphes, avec la création de configuration par défaut de celles-ci. Suite à une initiative personnelle de Dimitri, une nouvelle tâche fut rajoutée à notre projet, celle de développer un moteur 3D pour les simulations.

Ce TER nous a demandé de mettre en pratique plusieurs compétences acquises durant le premier semestre, notamment dans le domaine des Interfaces Homme Machine, du génie logiciel et bien sûr de la programmation orientée agent. Cela nous a demandé des compétences en Java, graphismes et en conduite de projet. Pour la rédaction de ce rapport nous avons utilisé \LaTeX .

Après avoir fait une description du MetaCiv existant, nous allons exposer nos objectifs. Le rapport sera divisé en trois parties principales. La première sur l'éditeur de plans, la deuxième sur l'amélioration des sondes et graphes et enfin la dernière partie sur la partie 3D de MetaCiv.

2 Glossaire

2.1 Partie programmation des sondes et éditeur de plan

2.1.1 Plan :

Un plan est une suite d'action à effectuer pour simuler un comportement. C'est une phase dans laquelle l'agent se trouve, et durant laquelle il exécute des actions en lien avec son état actuel.

2.1.2 Cogniton :

Le cogniton est, en quelque sorte, une "unité de pensée" qui influe sur les choix de l'agent. Comme l'agent est amené à faire des choix à plusieurs moments de la simulation, cette unité de pensée n'est pas figée, elle évolue au cours du temps.

2.1.3 Attribut :

Les attributs sont propres à une civilisation. Cela peut être le charisme, l'âge, la faim, etc... C'est ce qui caractérise la civilisation. Ces attributs évoluent au cours du temps.

2.1.4 Objet :

Les objets sont des objets courants qui peuplent l'environnement des agents. Cela peut être des baies, des poissons, de la viande... Les agents peuvent interagir avec eux en les prenant ou en les déposant.

2.2 Partie moteur 3D

2.2.1 Vertex Shader :

Le vertex shader est le premier stage programmable dans la pipeline d'OpenGL, il est le seul stage obligatoire dans la pipeline graphique.

2.2.2 Vertex fetching :

Avant que le vertex shader soit lancé, il y'a une étape appellé vertex fetching ou vertex pulling qui est une fonction fixée d'OpenGL. Elle a pour but de donner les entrées au vertex Shader.

2.2.3 Vertex Attributes :

Vertex Attributes est la méthode d'introduction des vertex data dans la pipeline OpenGL, pour déclarer un vertex attribute, il faut déclarer une variable dans le vertex shader en usant le qualificateur `in`.

2.2.4 Tessellation :

La tessellation est un procédé pour « casser » une primitive, connue sous le nom de patch dans OpenGL, en plusieurs petite primitives comme des triangles.

2.2.5 Tessellation Control Shaders :

La première des trois phases de la tessellation est la tessellation control shader. Ce shader prend en entrée la sortie du vertex shader, et est responsable de deux choses :

- déterminer le niveau de tessellation que l'on envoie au moteur de tessellation
- générer les données qui vont être au tessellation evaluation shader

2.2.6 Tessellation Engine :

Le moteur de tessellation est une fonction fixe d'OpenGL de la pipeline, qui prend une surface représenté par un patches et le casse en plusieurs primitives simple comme des points, lignes ou triangles.

2.2.7 Tessellation Evaluation Shaders :

Après le passage par le moteur de tessellation, il produit un nombres de vertices en sortie représentant les primitives créés. Ils sont passés au Tessellation Evaluation Shaders, pour un traitement quelconque, mais ce niveau de shaders est susceptible d'être appelé un grand nombre de fois, il faut donc faire attention au calcul effectué.

2.2.8 Geometry Shaders :

Le geometry shader est le shader intervenant après le vertex et l'étape de tessellation et avant la rasterization. Le geometry shader est exécuter une fois par primitive.

2.2.9 Rasterization :

La rastérisation est le procédé pour déterminer quels fragments peut être à l'intérieur d'une primitive.

2.2.10 Fragment Shaders :

Le fragment shader est le dernier stage programmable dans la pipeline d'OpenGL. Ce stage est responsable de la détermination de la couleur pour chaque fragment avant d'être envoyé au framebuffer.

2.2.11 Framebuffer :

Le framebuffer est la dernière étape de la pipeline graphique d'OpenGL. Il représente le contenu visible de l'écran et de nombreuses régions de mémoire qui sont utilisé pour stocker des valeur par pixel autre que la couleur.

2.2.12 Model-View Matrice :

La transformation la plus commune dans une application 3D, est de prendre le model dans le model space et de le passer dans le model-view space. Nous déplaçons le model dans l'espace du monde et ensuite dans l'espace de la vue.

2.2.13 Projection Matrice :

La projection matrice est appliquée à tous les vertices après la model-view transformation. Cette projection définit le volume de vue est établit les plans de clipping. Les plans de clipping, sont des équations de plan 3D utilisées par OpenGL pour déterminer si une primitive peut être vue.

2.2.14 Perspective Matrice :

Une fois que les vertices sont dans l'espace de vue, il nous faut passer dans le clip space, que nous faisons en appliquant notre projection matrice, qui peut être représentée par une perspective ou orthographique matrice. Avec la perspective matrice nous pouvons remarquer la déformation liée à la perspective.

3 MetaCiv, le point de départ et nos objectifs

MetaCiv est un framework de modélisation et de simulation de sociétés basé sur la programmation orientée agents et les systèmes multiagents. Il repose sur la plateforme MadKit [1] / TurtleKit [2], qui permet une programmation orientée agents en langage Java [4]. C'est un outil utilisé pour modéliser et simuler des sociétés humaines, il touche donc le domaine de la sociologie et permet des avancées dans celui de l'archéologie.

3.1 La mise en route

3.1.1 L'éditeur de plan

Lors de notre première réunion de TER, MetaCiv nous fut présenté brièvement car nos missions allaient être quasi exclusivement tournées vers du développement pur. L'interface fut l'objet principal de cette réunion. En effet, nous avons pris en main le framework pour connaître les tenants et aboutissants des futurs lignes de codes que nous allons ajouter et/ou modifier. Nous nous sommes principalement concentrés sur l'éditeur de plan (voir la figure 1) car il allait être l'objet de la tâche essentielle du TER.

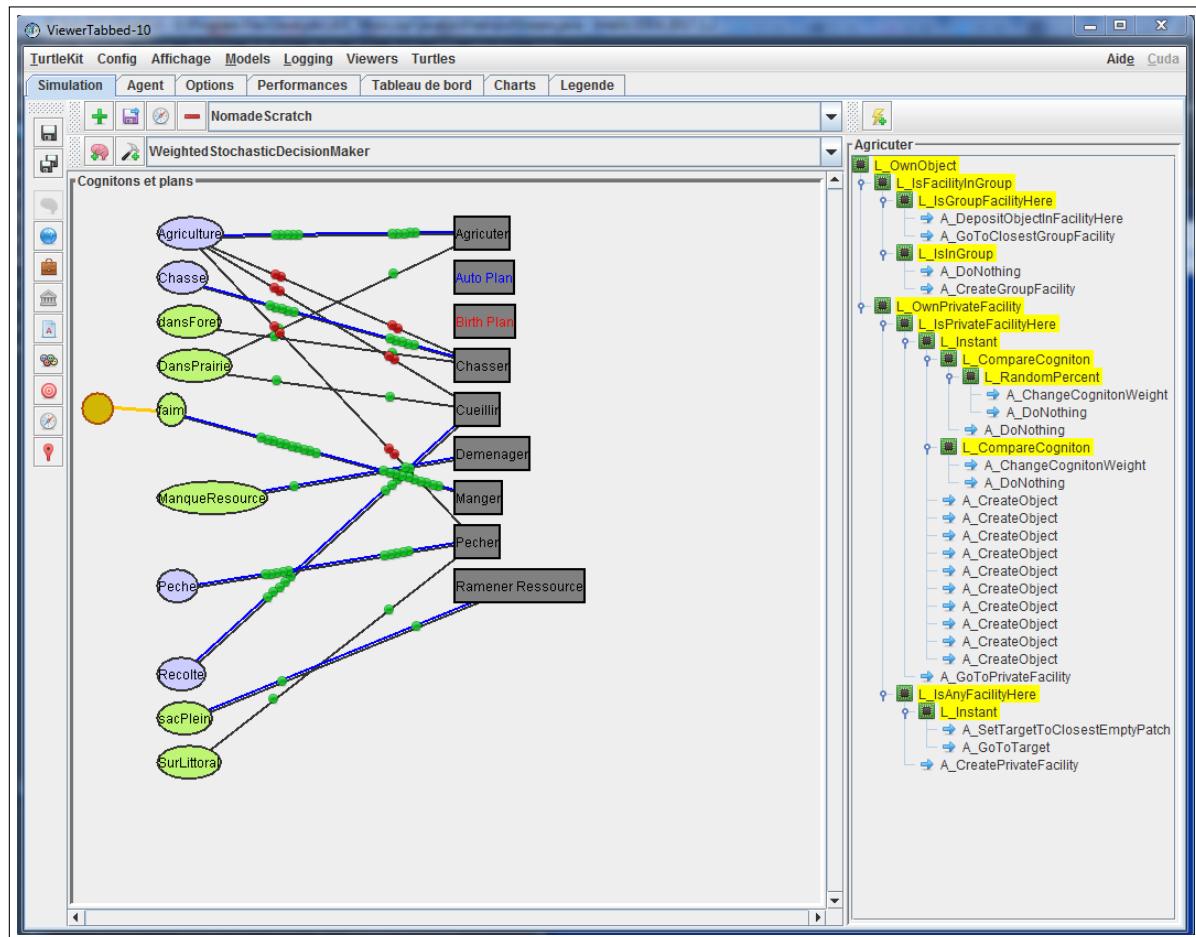


Figure 1. Vue d'ensemble de l'ancien éditeur de plan

La partie que nous allons principalement revoir se situe au niveau de l'arborescence sur la droite. Nous devons rendre la vision du plan plus abordable pour un utilisateur final. Nous allons expliquer les différentes notions (vocabulaire, idées, etc.) liées à l'éditeur de plans dans la section qui lui est consacrée (voir la section 4). L'idée qui a émergé dès la première réunion, était de reprendre le design de Scratch [3]. Un design simple et coloré, rendant la prise en main facile.

3.1.2 Les sondes et graphes par défaut

Nous avons aussi aborder la question des sondes qui sera le deuxième objectifs de ce TER. Nous avons donc assisté à une démo (voir la figure 2) de la part de François pour nous montrer ses attentes. Il nous a aussi expliqué le contexte de ses sondes, ainsi que le travail à accomplir. L'idée première était de définir des sondes qui traquaient des données de la simulation par défaut, sans configuration de la part de l'utilisateur. Deux voies nous étaient proposées, une dites "algorithmique" et une "statique". Nous nous sommes orientés naturellement vers la première voie.

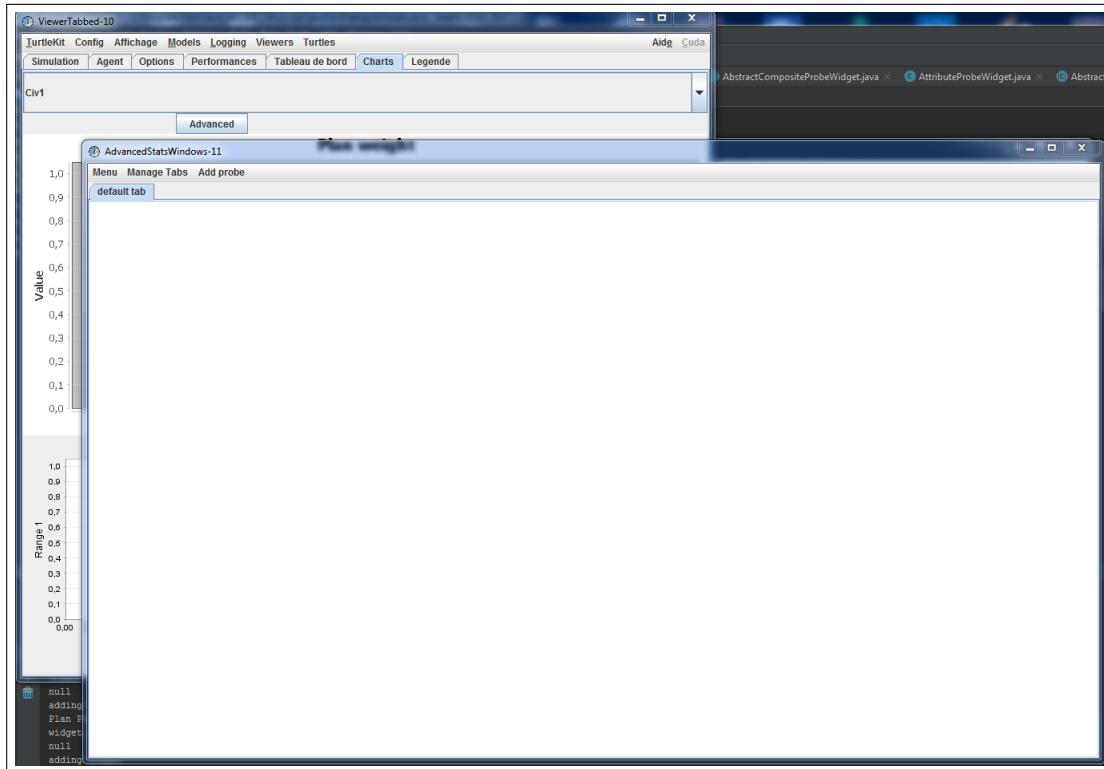


Figure 2. État initial de la fenêtre des sondes. Page vierge.

Ces graphes sondent des données très utiles pour l'utilisateur. En effet, cela lui permet de suivre sa ou ses civilisations de façon détaillée. Les graphes apportent une dimension visuelle aux données, ceci est plus intuitif que de simples chiffres (voir la section 5 pour le détail de cette partie du TER).

3.1.3 L'apparition du moteur 3D

Pourquoi nous parlons d'apparition ? Car ce moteur 3D n'était pas prévu dans les plans de Mr. Ferber. Il est né d'une initiative personnelle de Dimitri. En effet, lors de la deuxième réunion, Dimitri a fait une démonstration de son moteur, et cela a fait forte impression (voir la figure 3). Suite à cette démo, Mr. Ferber fut convaincu d'ajouter le moteur aux objectifs de départ du TER. La démo représentait simplement l'environnement des agents, avec fidélité notamment sur la présence de montagnes, de prairies, de la mer, etc... L'étape suivante était de représenter les agents dans cette environnement. La vue 3D ouvre une nouvelle perspective dans l'observation des agents. Cette partie du TER est devenue tout aussi importante que le nouvel éditeur de plans.

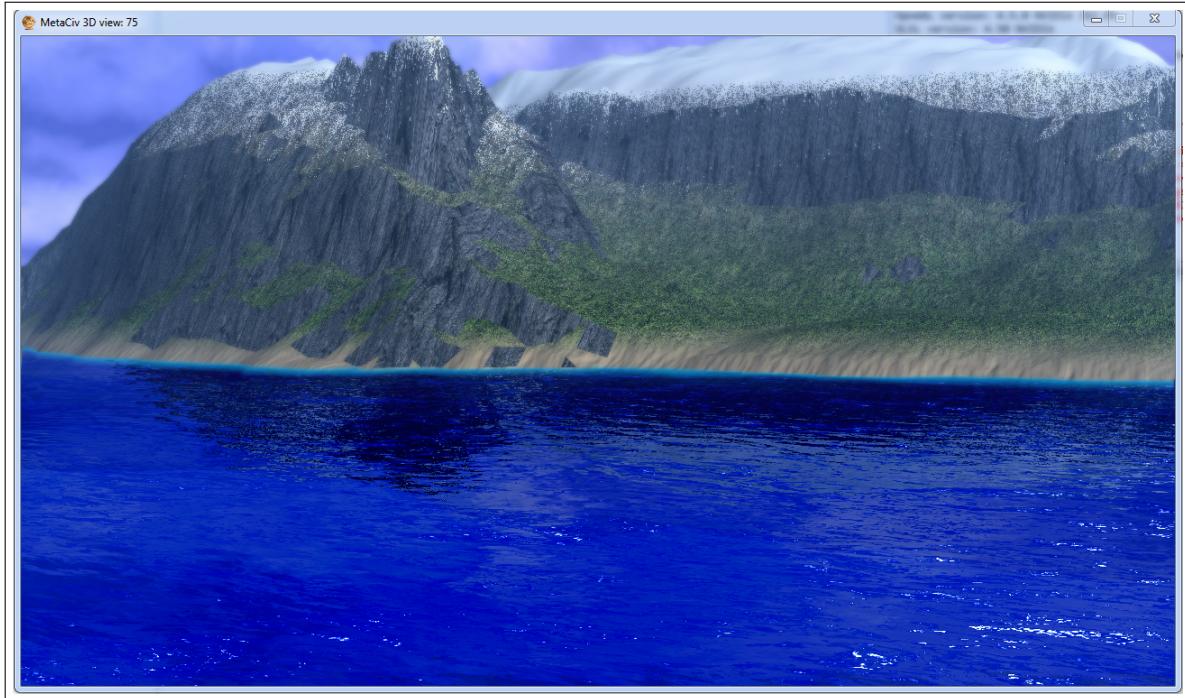


Figure 3. Aperçu du moteur 3D (Forme finale)

3.1.4 Objectifs et organisation

3.1.4.1 Objectifs

1. Faire un nouvel éditeur de plans inspiré de Scratch
 - (a) Coloré
 - (b) Graphique
2. Créer des sondes par défaut pour permettre à l'utilisateur d'avoir un retour de données de la simulation
 - (a) Sauvegarder à la création
 - (b) Un onglet simple, avec quelque graphes basiques
 - (c) Un onglet complet avec tous les graphes
 - (d) Un onglet vierge personnalisable
3. Implémenter un moteur 3D
 - (a) Représentation 3D de l'environnement
 - (b) Représentation des agents
 - (c) Amélioration et ajout de différents atouts pour le moteur 3D
4. Une civilisation qui modélise toutes nos implémentations

3.1.4.2 Organisation Nous nous sommes organisés autour de 3 tâches importantes, la 3D, l'éditeur de plan et les graphes par défaut.

- Aranud
 - Editeur de plan
 1. Conception de l'IHM (avec Thomas)
 2. Réalisation d'un prototype
 3. Intégration du prototype dans MetaCiv
 - Dimitri
 - Programmation du moteur 3D
 1. Représentation du monde
 2. Représentation des agents
 3. Interactions avec les agents
 - Thomas
 - Mise en place de graphes par défaut
 1. Création des onglets
 2. Sauvegarde
 3. Configuration par programmation
 - Rédaction des documents finaux

Nous avons utilisé Github pour versionner notre code. Nous faisions des réunions avec nos encadrants toutes les deux semaines pour faire un point sur l'avancement. Toutes les deux semaines nous avions l'obligation de montrer quelque chose, même si ce n'était pas fonctionnel. Les réunions duraient entre 1h et 1h30, et elles se déroulaient au LIRMM.

4 L'éditeur de plan

L'objectif principal de notre TER était le design et l'implémentation d'un éditeur permettant de modifier les plans utilisés dans Metaciv pour définir le comportement des agents dans la simulation. Cet éditeur ayant pour objectif de remplacer le mécanisme d'édition de plans déjà en place dans Metaciv.

L'éditeur devait répondre à plusieurs critères importants notamment au niveau du design et de l'ergonomie, le but final étant de faire un éditeur qui soit au goût du jour et avec une prise en main facile et rapide pour un utilisateur lambda.

4.1 Design de l'éditeur de plan

La première étape de création du nouvel éditeur de plan fut de comprendre le fonctionnement des plans dans Metaciv, de noter les spécifications requises ainsi que les suggestions données par les encadrants de notre TER pour le développement de cet éditeur. Nous avons par la suite réalisé un premier jet du design et enfin après les remarques sur celui, nous avons obtenu le design final de l'éditeur de plan.

4.1.1 Analyse de l'éditeur original

L'éditeur devait remplir plusieurs spécifications et critères importants. Ces critères s'appuyaient sur les manques et faiblesses révélées par l'utilisation de l'éditeur existant de Metaciv. Pour comprendre pourquoi l'éditeur existant était insuffisant, il faut comprendre comment fonctionne un plan dans Metaciv.

4.1.1.1 Fonctionnement d'un plan Un Plan est une sorte d'algorithme utilisé par les agents dans Metaciv. Ces plans permettent de définir des comportements pour ces agents grâce à une succession d'actions prédéfinies dans Metaciv. Ces actions permettent de réaliser des tests de conditions ainsi que d'interagir avec les agents via des paramètres modifiables différents d'actions à actions. Elles sont utilisées par exemple pour faire construire un bâtiment à un agent ou vérifier que un agent a les matériaux nécessaires pour la construction de ce bâtiment.

Ils y a 3 types distincts d'actions dans un plan :

1. Les actions simples, elles permettent d'interagir avec l'agent, de le modifier ou de lui faire faire une activité.
2. Les actions logiques qui sont divisées en deux catégories :
 - (a) Les actions de type « Si, Sinon », si l'agent remplit la condition de l'action logique, il va effectuer une action sinon il en effectuera une autre.
 - (b) Les actions de type « Répéter » qui sont des listes d'actions qui doivent être répétées ou bien exécutées en même temps.

Un plan est donc une suite et combinaison d'actions dans le but de faire réaliser quelque chose à un agent. Un exemple de plan pourrait être le plan Manger, pour faire manger un agent, ou le plan Pêcher.

On peut en déduire que les plans peuvent varier du très simple au très compliqué suivant leur objectif final. Ils ont cependant en commun de comporter un nombre important d'actions qui doivent toutes être paramétrées pour obtenir le comportement voulu.

4.1.1.2 Faiblesses de l'éditeur existant L'éditeur original de Metaciv se présentait sous la forme d'un arbre où chaque nœud de l'arbre représentait une action du plan.

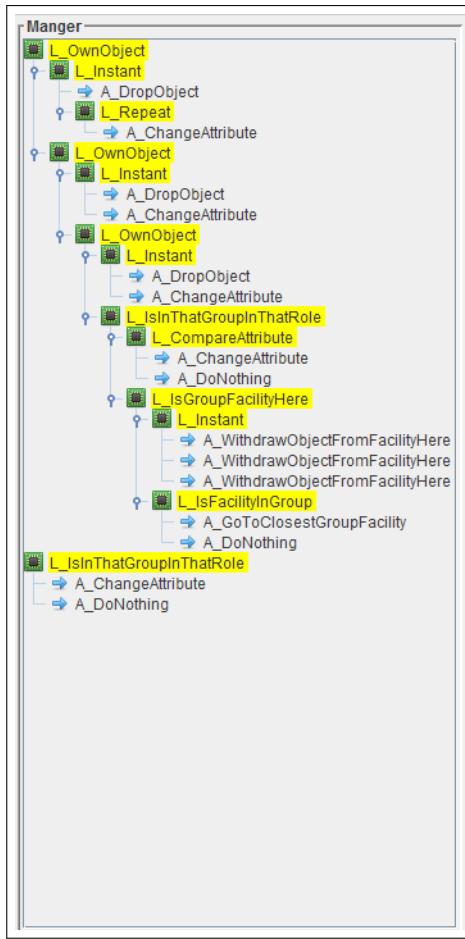


Figure 4. Plan "Manger" dans l'ancien éditeur

L’arbre étant vertical, les actions s’exécutent de haut en bas.

Les nœuds non feuilles de l’arbre symbolisait les actions logiques d’un plan ainsi que les actions « Répéter ». Pour les actions logiques, le premier nœud fils représente l’action à effectuer si la condition de l’action logique est remplie tandis que le deuxième nœud est pour le cas où la condition n’est pas remplie. Pour les actions « Répéter », les nœuds fils sont les actions que l’action doit exécuter.

Les nœuds feuilles sont les actions simples d’un plan.

Une des faiblesses de l’éditeur à ce niveau est que il n’y a pas de distinctions évidentes entre les actions logiques et les actions « Répéter », aussi, malgré le fait qu’il n’y a que deux actions possibles pour une action logique, il est possible d’ajouter autant de nœuds fils à une action logique ce qui fera que le plan ne fonctionnera pas, enfin il est aussi possible d’ajouter des nœuds fils à des actions simples alors que elles ne sont pas supposées en avoir.

Pour ajouter une action dans l’arbre, il fallait cliquer sur un bouton situer au dessus de l’arbre ouvrant une fenêtre de dialogue permettant de choisir dans une liste déroulante l’action à ajouté à l’arbre. Cette action est ensuite créée et ensuite ajouté comme premier dans l’arbre. Elle doit ensuite être paramétré via une autre boite de dialogue ainsi que déplacer via la souris à l’endroit voulu. Une autre façon d’ajouter une action est de faire un clic droit sur une action déjà dans l’arbre et d’utiliser l’option « insérer avant ou après ».

Autre défaut, il faut passer par un nombre important de fenêtre de dialogue pour ajouter ou modifier des actions. Il faut chercher dans une petite liste déroulante non filtrable pour trouver l’action à créer à chaque fois, il n’est possible de modifier une action que par une fenêtre d’édition. Au final, l’éditeur de plan original est utilisable mais peu pratique pour réaliser des plans complexes et requiert la connaissance des actions pour les différencier et réaliser un plan fonctionnel. De plus,

l'aspect visuel de l'éditeur est peu attractif et pas moderne.

4.1.2 Création du nouvel éditeur

Après avoir analyser l'ancien éditeur et trouver ses manquements et défauts, nous avons établis une liste de critères en coopération avec les encadrants du TER et réaliser le design final du nouvel éditeur.

4.1.2.1 Spécifications

La liste des critères à remplir est :

- Un aspect moderne pour l'éditeur, celui ci doit être plus proche des codes de design actuels, ce qui n'est pas compatible avec l'aspect d'arbre de l'éditeur original.
- Un moyen simple et efficace de différencier les différents types d'actions ainsi que leurs fonctionnements, par exemple un novice de Metaciv doit tout de suite comprendre que une action logique peut contenir deux actions seulement et que une action logique s'apparente à une condition « Si Sinon ».
- Les actions doivent être facilement configurables sans passer par des fenêtres de dialogue.
- Ils doit être possible de déplacer plusieurs actions en même temps pour aller plus vite.
- Les actions qui sont possibles d'ajouter doivent être facilement accessible et l'ajout d'une action doit être simple.
- Un moyen de rechercher une action voulu dans la liste des actions disponibles.
- Un système pour « Glisser / Déposer » les actions afin de les déplacer et les créer doit être mis en place.
- Possibilité de « Copier / Coller » des actions.

Pour répondre à ces critères, Mr Ferber à proposer de réaliser un éditeur semblable à celui du langage Scratch, l'éditeur du langage réunissant tous les besoins listés et est facilement utilisable car Scratch est un langage destiné à apprendre la programmation aux enfants et débutants.



Figure 5. Editeur Scratch

4.1.2.2 Crédit de la maquette Pour créer la maquette de l'éditeur nous nous sommes donc inspirés de Scratch. Nous avons chercher à produire une version simplifiée pour être plus facilement et rapidement implémenter. Nous avons aussi chercher à adapter ce concept aux actions d'un plan. Nous avons tout d'abord choisi de créer une fenêtre d'édition avec deux parties, une partie édition à proprement parler où seront positionner les actions du plans et où l'essentiel de la manipulation de l'éditeur se situera. La deuxième partie sera la liste des actions disponibles pour définir un plan.

Cette deuxième partie comportera des sections permettant de grouper des actions semblables pour ainsi permettre à l'utilisateur de trouver plus facilement les actions dont il a besoin. Par exemple une section pour les actions simples et une autre pour les actions logiques. L'utilisateur pourra alors glisser et déposer depuis la liste vers la partie éditeur pour ajouter une action au plan, comme dans Scratch. Cela résout notre critère de création d'action par une solution simple et intuitive. De plus une barre de recherche se situera au dessus de la liste des actions possibles, elle permettra de rechercher une action dans la liste, la liste se modifiera au fur et à mesure que l'utilisateur tape sa recherche montrant ainsi les résultats obtenus par celle ci.

La première partie est la plus importante, les actions d'un plan seront représentées par des blocs qu'il sera possible de déplacer avec la souris, cela permettra de changer rapidement les combinaisons de blocs. Le système de « Glisser / Déposer » nous pose un cependant un problème, comment savoir quels séries de blocs est celle qui doit être exécuté par un agent, pour résoudre ce problème nous avons choisi d'ajouter sur l'éditeur un bloc racine qui ne sera pas interactif, tous les blocs qui sont rattachés au bloc racine seront ceux qui seront pris en compte dans l'exécution du plan.

Les blocs incorporeront un effet magnétique qui permettra de rattacher deux blocs entre eux créant ainsi une succession d'actions à réaliser dans le plan. Les actions simples seront des blocs simples tandis que les blocs logiques auront deux emplacements reconnaissables qui seront utilisés pour placer les deux actions à effectuer si la condition de l'action logique est satisfaite ou pas. Il ne doit être possible de mettre que une seule action dans les emplacements d'un bloc logique mais la nature de l'action n'est pas importante cependant si c'est une autre action logique il faudra que la taille du bloc s'adapte au fait que cette action à elle même des actions à effectuer à l'issu de son test de condition. Les blocs « Répéter » seront semblables aux blocs logiques dans leur fonctionnement mais il n'auront que un emplacement pour blocs d'action et il pourra incorporer autant d'actions que voulu.

Pour différencier ces blocs, on utilisera un code couleur :

- Les blocs simples seront violet.
- Les blocs logiques seront jaunes.
- Les blocs « Répéter » seront verts.

On utilisera des couleurs utilisées fréquemment dans le design de site web issu du flat design.

Pour permettre le paramétrage rapide d'un bloc, nous avons décidé d'implémenter là aussi un système similaire à Scratch. Nos blocs contiendront des listes déroulantes, des champs à remplir pour modifier les paramètres de l'action. A la différence de Scratch, il ne sera pas possible d'utiliser des blocs comme paramètres de blocs logiques, les différents types d'actions logiques produisant déjà un effet similaire.

Le clic droit sur un bloc permettra à l'utilisateur de modifier les paramètres de celui avec la fenêtre d'édition de l'éditeur original ainsi que la possibilité de supprimer un bloc de l'éditeur ou copier / coller un bloc.

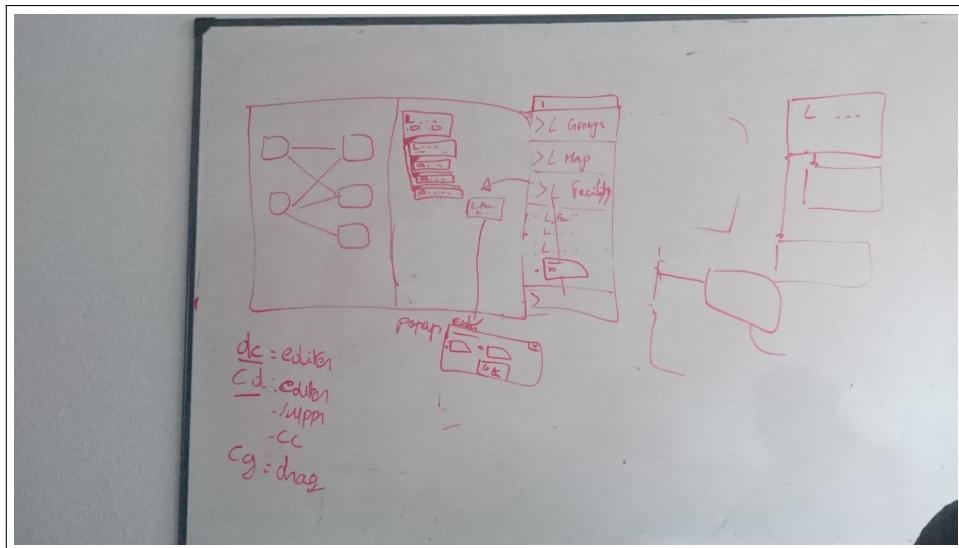


Figure 6. Photo du design issu d'un brainstorming

4.2 Prototype de l'éditeur

Afin de permettre aux encadrants du TER d'apporter leur critiques et remarques à l'éditeur nous avons tous d'abord décidé de créer une version prototype de l'éditeur afin de mettre en place au fur et à mesure les aspects fondamentaux de l'éditeur et ainsi une fois obtenue une version satisfaisante, intégré ce prototype dans Metaciv.

Le prototype contient au final une fenêtre contenant les deux parties de l'éditeur, la création des blocs par un glisser / déposer depuis la liste d'actions, remplie d'actions factices dans le prototype, le système de glisser / déposer permettant de bouger des blocs dans la partie édition, le système de magnétisme permettant de lier des blocs entre eux, les blocs correspondant aux actions simples, logiques et « Répéter », la présence du bloc racine, un menu clic droit sur un bloc permettant de le supprimer seulement ainsi que un exemple de paramètre de bloc modifiable, sous la forme d'une liste déroulante avec des valeurs factices.

Nous verrons plus ou moins en détail l'implémentation de chacune des ses fonctionnalités suivant leurs complexités et leurs importances.

4.2.1 Implémentation d'un bloc

4.2.1.1 Description d'un bloc Le bloc est l'élément central de l'éditeur de plan, il représente une action du plan qui sera effectué par un agent dans la simulation.

Comme il y a différents types d'actions, il y aura forcément différents types de blocs, un type de bloc pour chacun des types d'actions pour être exact.

Un bloc est composé d'un nom, celui de l'action, d'un contenu, d'un parent, et de un ou plusieurs fils.

Le contenu contient les différents paramètres de l'action, dans le prototype ils ne sont pas fonctionnels.

Le parent et les fils d'un bloc sont les éléments qui permettent de liés des blocs entre eux. Cela permet de créer un arbre de blocs où chaque opération sur un bloc peut alors affecter tous ses blocs fils ou son parent. Par exemple, quand on déplace un bloc, il demandera à ses fils de se déplacer aussi, de manière récursive.

Les blocs d'actions simples ont un fils nommé fils « Append », il représente le bloc directement en dessous.

Les blocs logiques de type « Si Sinon » ont 3 fils, un fils « Append », un fils « Si » qui représente le bloc action a effectué si la condition du bloc logique est respectée et un fils « Sinon » dans le cas

où la condition n'est pas respectée.

Les blocs logiques de type « Répéter » ont 2 fils, un fils « Append » et un fils « Répéter » qui est le premier bloc d'une chaîne de blocs que le bloc logique « Répéter » doit exécuter.

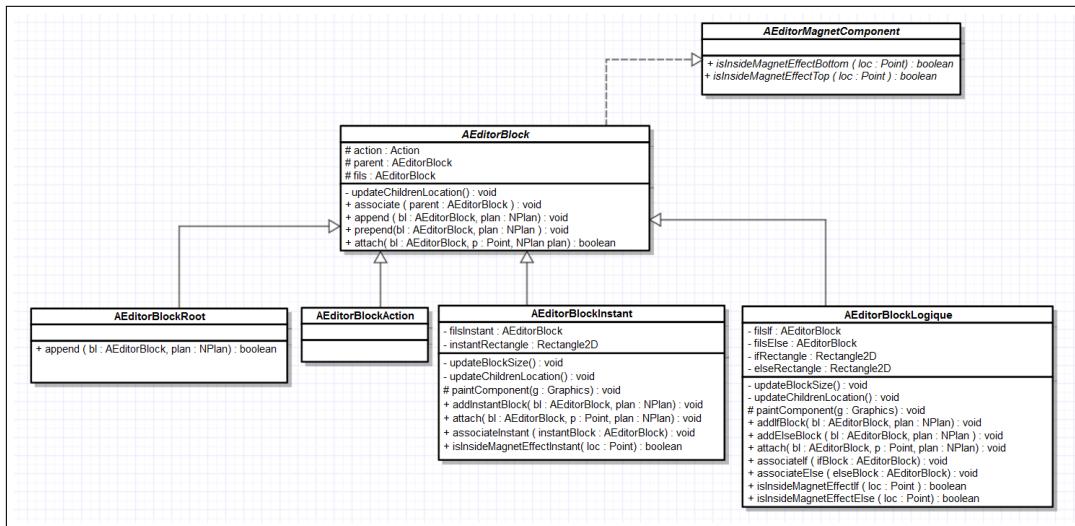


Figure 7. Diagramme UML simplifié des blocs

4.2.1.2 Effet Magnétique L'effet magnétique d'un bloc est ce qui permet de lier les blocs entre eux. Tous les types de blocs ont au minimum 2 zones d'effet magnétique, une zone au dessus du bloc et une en dessous. Du à la manière d'utiliser l'éditeur la zone au dessus du bloc n'est en pratique jamais utilisé. La zone en dessous, par contre, est utilisé quand un bloc est déplacé sous un autre pour les lier, le bloc qui a été déplacé devient alors fils du bloc sous lequel il a été déplacé.

Quand on déplace un bloc, l'éditeur va, pendant le déplacement, vérifier si la position de la souris se situe dans la zone magnétique d'un bloc en parcourant tous les blocs présents dans le panneau d'édition et en demandant à chacun de ces blocs si la position de la souris se situe dans leur zone. Si la souris est dans sa zone, le bloc va alors indiquer visuellement que oui en ajoutant une bordure verte dans la partie basse ou haute du bloc selon la zone magnétique où se situe le point.

Listing 1. Fonction de changement de bordure pour le magnétisme

```

public void changeBorder(Point p) {
    resetBorder();
    if(isInsideMagnetEffectBottom(p)) {
        if(canHaveChild()) {
            setBottomBorder(AEditorBlock.OK_MODE);
        }
    } else if(isInsideMagnetEffectTop(p)) {
        if(!hasParent() && !hasParent()) {
            setTopBorder(AEditorBlock.OK_MODE);
        }
    }
}
  
```

Dans le code ci dessus, on voit que le point `p`, qui est la position de la souris, est vérifié par le bloc et si il est dans sa zone magnétique basse, le bloc ajoute une bordure basse `OK_MODE` qui est le code pour une bordure verte.

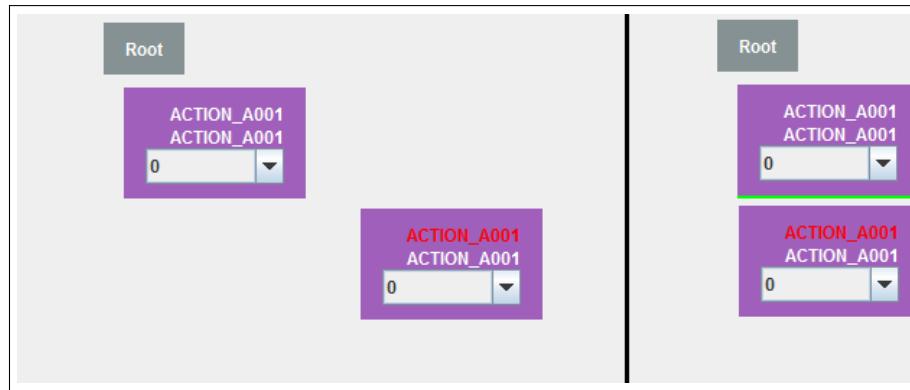


Figure 8. Comparaison entre un bloc déplacé hors d'une zone magnétique et dans une zone magnétique

Comme les blocs logiques ont un ou deux emplacements pour ajouter des blocs en plus de la possibilité d'en ajouter après eux, ils ont besoin de zones magnétiques supplémentaires pour permettre l'ajout de blocs dans ces emplacements.

Les blocs logiques de type « Si, Sinon » ont alors deux zones magnétiques supplémentaires, une pour la partie « Si » du bloc et une autre pour la partie « Sinon ». Ces zones magnétiques ont le même fonctionnement que les zones en dessous et au dessus d'un bloc à la différence que elles ne sont utilisées que si le bloc logique n'a pas déjà un bloc dans ses emplacements logiques, cela veut dire que si on veut changer le bloc effectué par l'action logique, il faut déplacer celui qui est déjà présent et placer le nouveau à la place.

Pour représenter la zone vide d'un emplacement logique, où se situe alors la zone magnétique, on utilise un rectangle avec des dimensions fixes qui sera peint par dessus le bloc, ce bloc aura la même couleur que la couleur de fond de l'éditeur pour donner l'illusion d'un trou. Quand on déplace la souris dans la zone magnétique d'un emplacement logique, ce rectangle deviendra vert pour montrer que l'on peut poser le bloc à l'endroit de la souris.

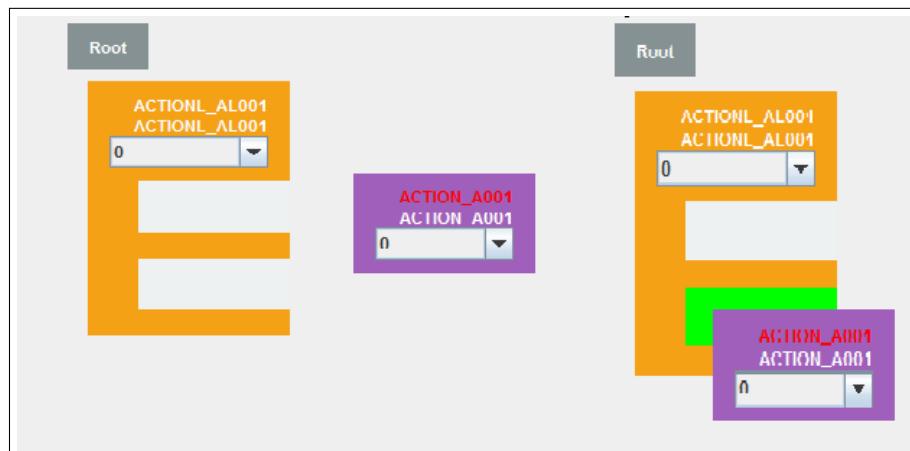


Figure 9. Comparaison entre un bloc logique normal et lorsque l'on déplace un bloc dans une de ses zones magnétiques

Ce principe est le même pour les blocs logiques de type « Répéter » mais eux disposent seulement d'un seul emplacement logique.

Listing 2. Fonction de peinture pour les blocs logiques

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
```

```

updateBlockSize(); // met à jour la taille du block avant de la repeindre
g.setColor(instantColor);
g.fillRect((int) instantBlock.getX(), (int) instantBlock.getY(), (int)
instantBlock.getWidth(), (int) instantBlock.getHeight());
}

```

Cette fonction permet de peindre un bloc logique de type « Répéter », la variable `instantColor` représente la couleur qui va être utilisé pour peindre le rectangle juste après.

4.2.1.3 Lier des blocs entre eux Quand on dépose un bloc dans une zone magnétique d'un autre bloc, les deux blocs vont alors se liés.

Pour lier deux blocs il faut faire appel à une fonction d'attachement nommée `attach()`. Cette fonction va vérifier à nouveau que le point où est déposé le bloc à attacher se situe dans une zone magnétique et vérifier que le bloc qui va devenir parent peut recevoir un fils. Si ces deux conditions sont satisfaites, on peut alors liés les deux blocs.

Dans le cas de l'ajout d'un fils « Append », comme il est possible d'insérer un bloc entre deux blocs déjà liés, il faut vérifier que le bloc parent n'a pas déjà un fils « Append », si il n'en a pas, on ajoute simplement le bloc avec la fonction `append()` sinon on doit détacher le fils « Append » courant du parent, ajouter le nouveau bloc avec la fonction `append()` puis ajouter avec `append()` à nouveau le fils que l'on à détacher au bloc parent au bloc que l'on vient d'ajouter.

La fonction `append()` ne fait que mettre à jour les variables représentant le parent et les fils des deux blocs que l'on cherche à lier ainsi que déplacer le nouveau bloc fils, ainsi que ses éventuels fils, sous le bloc parent.

Listing 3. Fonction d'attachement des blocs

```

public boolean attach(AEditorBlock block, Point p, NPlan plan) {
    if(isInsideMagnetEffectBottom(p) && canHaveChild()) {
        if(!hasChild()) { // si pas de fils, on append direct
            append(block, plan);
        } else { // sinon on recupere le fils actuel, on append le nouveau block puis on
            // append l'ancien fils au block qu'on a ajoute
            AEditorBlock currentChild = getChild();
            dissociateFromChild(currentChild);
            append(block, plan);
            block.getLastChild().append(currentChild, null); // plan a null puisque le
            // plan est deja modifie dans append du nouveau block
        }
        return true;
    } else if(isInsideMagnetEffectTop(p)) {
        if(!hasParent()) {
            prepend(block, plan);
            return true;
        }
    }
    return false;
}

```

Pour les blocs logiques, cette fonction est surchargée pour prendre en compte les ajouts de fils dans les emplacements logiques.

Listing 4. Fonction d'attachement des blocs

```

@Override
public boolean attach(AEditorBlock block, Point p, NPlan plan) {
    if(isInsideMagnetEffectInstant(p)) {
        if(filsInstant == null) {

```

```

        addInstantBlock(block, plan);
    }
    return true;
}
return super.attach(block, p, plan);
}

```

Ici le code de la fonction `attach()` d'un bloc logique « Répéter », on voit le rajout de la gestion d'un ajout d'un fils « Répéter ». Les mêmes modifications sont réalisées sur la fonction `attach()` des blocs logiques « Si Sinon » pour gérer les fils « Si » et « Sinon ».

On peut aussi voir que dans les cas d'ajout de fils « Répéter » ou « Si » ou « Sinon », la fonction `append()` est remplacée par une autre fonction qui suivra la même logique mais qui modifiera des variables différentes et placera le bloc ajouté et ses fils à des endroits différents. Dans le code précédent, `append()` est remplacé par `addInstantBlock()`.

4.2.1.4 Particularités des blocs logiques Quand on ajoute un bloc dans un emplacement logique d'un bloc logique, le bloc que l'on ajoute est potentiellement relié lui-même à d'autres blocs, il faut donc que le bloc logique s'agrandisse pour s'adapter à la taille totale de la chaîne de bloc que l'on vient d'ajouter. Cette opération est réalisée par la fonction `updateBlockSize()` avant que le programme affiche un bloc logique, dans la fonction `paintComponent()` vu précédemment.

Cette fonction calcule la taille que doit faire le bloc logique, elle prend en compte la taille du contenu du bloc, la taille de tous les blocs dans les emplacements logiques ainsi que la taille des séparateurs utilisés pour structurer visuellement le bloc logique, comme par exemple séparer la partie « Si » et « Sinon » d'un bloc, et ceux entre chaque bloc.

Pour calculer la taille totale des blocs dans les emplacements logiques, la fonction récupère de façon récursive la taille de ces blocs. La fonction met alors à jour les rectangles qui permettent de créer les trous dans les blocs logiques.

Listing 5. Fonction d'attachement des blocs

```

if(filsIf != null) {
    AEditorBlock child = filsIf;
    double nHeight = spaceBetweenBlockPlusBorder; // space before block
    ++nbif;
    while(child != null) {
        nHeight += child.getPreferredSize().getHeight() + spaceBetweenBlockPlusBorder; // height of block + space after it
        ++nbif;
        child = child.getChild();
    }
    ifBlock.setRect(ifBlock.getX(), ifBlock.getY(), ifBlock.getWidth(), nHeight);
} else {
    ifBlock.setRect(ifBlock.getX(), ifBlock.getY(), ifBlock.getWidth(),
        DEFAULT_INNER_SIZE);
}

```

Dans cette partie de la fonction `updateBlockSize()` d'un bloc logique « Si,Sinon », on voit que si le bloc a un fils « Si », on ajoute la taille des blocs fils « Append » récursivement tant qu'il y a des blocs qui se suivent, on note aussi que l'on prend en compte le petit espace qu'il y a entre chaque bloc pour pas qu'ils ne soient collés. La fonction agrandit par la suite le rectangle qui fait le trou du compartiment « Si » du bloc. Dans le cas où il n'y aurait pas de fils « Si », ce rectangle est redimensionné à sa taille initiale.

Après avoir modifié la taille du bloc, il faut aussi mettre à jour la position des fils du bloc pour prendre en compte la nouvelle taille du bloc. Pour ce faire, au lieu de recalculer manuellement la nouvelle position des blocs fils, on va les détacher puis les rattacher, les blocs seront alors positionnés.

à la bonne position. Enfin il faut aussi actualiser le parent du bloc logique au cas où le bloc parent soit lui même un bloc logique, on a juste à demander au parent de s'afficher à nouveau, il effectuera alors la procédure que l'on vient de décrire.

4.2.2 Fenêtre éditeur

4.2.2.1 Panneau édition Le panneau édition est la partie gauche de la fenêtre éditeur, c'est dans ce panneau que les blocs sont affichés et manipulés.

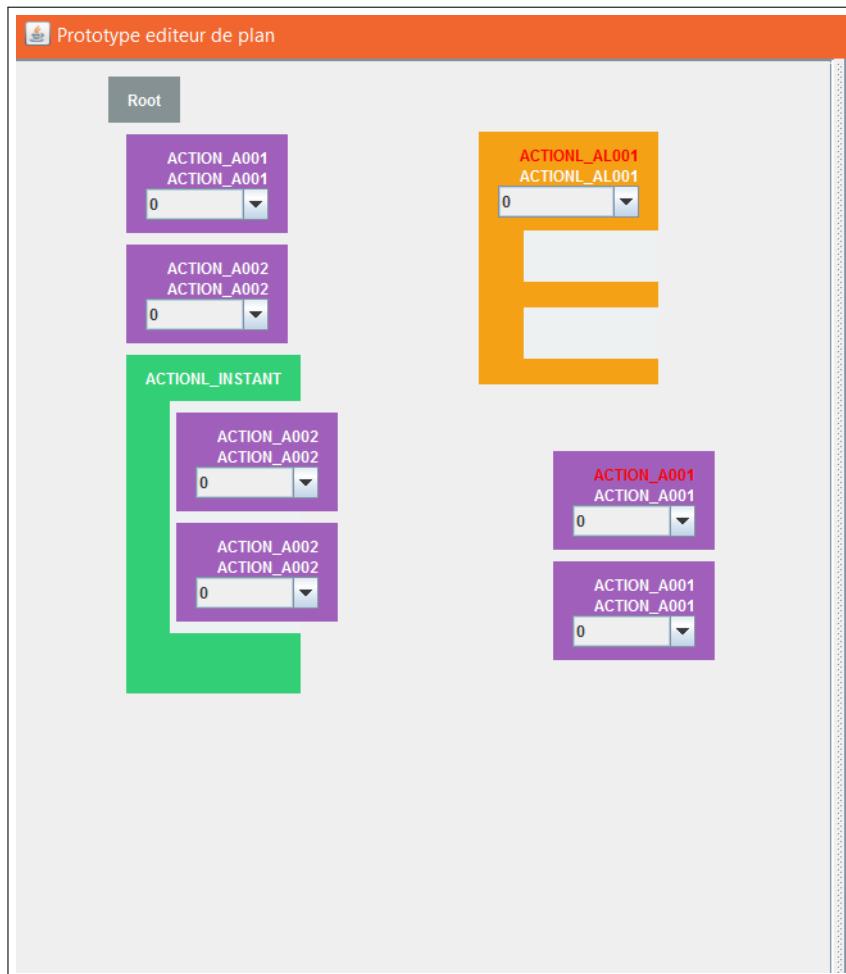


Figure 10. Partie édition du prototype

Ce panneau contient de base un bloc nommé « Root », c'est la bloc racine qui représente le début d'un plan. Il est ajouté à une position fixe à chaque lancement de l'éditeur de plan. C'est un type de bloc spécial qui ne permet que l'ajout de fils « Append » et qui ne représente aucune action.

Comme il est possible que des blocs se chevauchent, il faut que le panneau gère ce problème d'affichage. Le panneau hérite donc de la classe `JLayeredPane` qui permet d'affecter des indexs aux blocs pour qu'ils s'affichent correctement.

Pour pouvoir bouger les blocs dans le panneau d'édition, la classe du panneau d'édition contient une classe interne qui est un `listener` qui sera ajouté à chaque bloc dans le panneau édition. On crée une classe interne car cela nous permet d'effectuer facilement des actions sur le panneau depuis cette classe.

Cette classe interne gère les clics de souris sur les blocs. Il y a trois actions différentes qui sont gérées :

- Quand on clique gauche sur un bloc.

- Quand on fait glisser un bloc en maintenant le clic de souris.
- Quand on dépose le bloc en relâchant le clic.

Quand on clique gauche sur un bloc, le bloc va être immédiatement détacher de son fils « Append » et de son parent, si c'est un bloc logique il va quand même garder ces fils logiques. Quand on fait glisser la souris, le bloc est alors déplacé pour qu'il reste sous la souris et aussi on va aussi regarder si l'endroit où est le curseur de souris se situe dans une zone magnétique d'un bloc présent dans le panneau d'édition, si c'est le cas le bloc concerné affichera sa bordure verte pour indiquer que si l'on relâche la souris le bloc en train d'être déplacé sera lié à ce bloc. Quand on relâche la souris, on parcourt tous les blocs présents dans le panneau édition, on leur enlève leurs bordures vertes et on essaye d'attacher le bloc déposé à chacun des blocs, si l'opération est réussie, on arrête la boucle sinon si on a pas réussi à lier le bloc déposé à un autre bloc, on le dépose à la position de la souris.

4.2.2.2 Panneau de la liste des actions Le panneau de droite sur l'éditeur est le panneau qui contient la liste des actions qu'il est possible d'ajouter au plan.

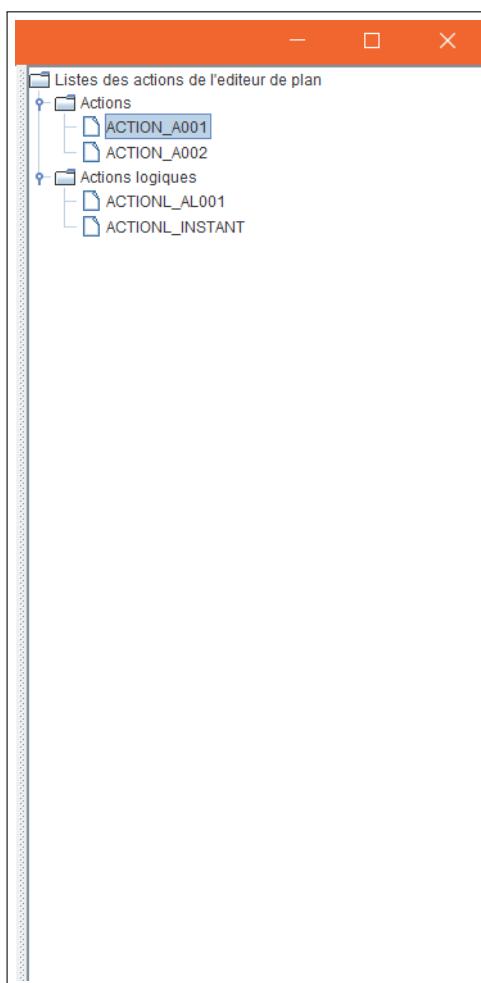


Figure 11. Partie liste des actions du prototype

Ce panneau contient un arbre Jtree qui est découpé en deux catégories :

1. La première catégorie contient les actions simples.
2. La deuxième, les actions logiques.

Un nœud dans l'arbre ne contient que le nom de l'action qu'il représente. Ce nom sera utilisé par la suite pour créer une action qui permettra elle-même de créer un bloc.

4.2.2.3 Crédit d'un nouveau bloc Pour créer un nouveau bloc, il faut que l'utilisateur glisse et dépose une action, dans la liste des actions, sur le panneau édition.

Pour permettre cette action, il a fallu implémenter un `TransferHandler` entre le panneau d'édition et le panneau de la liste des actions. Un `TransferHandler` permet de faire passer des informations entre deux éléments de l'interface graphique. Dans notre cas, on fera passer le nom d'une action, tirée du panneau des actions, au panneau d'édition pour que ce dernier puisse créer l'action correspondante au nom obtenu et un nouveau bloc à partir de cette action.

Dans le prototype les actions ne sont pas utilisées, on fait alors juste passer des noms factices mais le principe est le même.

Comme le transfert de données ne concerne en réalité que le panneau d'édition et l'arbre d'actions lui-même et non pas le panneau d'actions, il a fallut créer une classe étendant `Jtree` pour représenter l'arbre d'actions.

Dans cette nouvelle classe ainsi que dans la classe du panneau d'édition, on a ajouté une classe interne héritant de `TransferHandler` qui vont chacune réaliser une action différente.

Pour l'arbre d'actions, cette classe va simplement définir que seule la copie d'un nom d'action est transférable et créer l'objet à transférer contenant le nom de l'action.

Listing 6. Fonction permettant le transfert des blocs

```
private class ActionsJTreeTransferHandler extends TransferHandler {

    @Override
    public int getSourceActions(JComponent c) {
        return TransferHandler.COPY; // copy seulement
    }

    @Override
    protected Transferable createTransferable(JComponent c) {
        JTree tree = (JTree) c;
        DefaultMutableTreeNode node =
            (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
        if (node.isLeaf()) { // seulement les feuilles sont transférables
            return new StringSelection(node.getUserObject().toString());
        } else {
            return null;
        }
    }
}
```

Pour créer cet objet, on cherche le nœud correspondant à une action sélectionnée dans l'arbre puis on retourne un nouvel objet transférable contenant le nom de l'action.

Dans le panneau d'édition, la classe `internet` va effectuer deux actions, elle va d'abord vérifier pendant toute la durée du transfert que l'endroit où est la souris permet la création d'un nouveau bloc grâce à la fonction `validateDropLocation()`.

Listing 7. Fonction qui valide ou non l'import

```
@Override
public boolean canImport(TransferSupport support) {
    //check for string flavor
    if (!support.isDataFlavorSupported(DataFlavor.stringFlavor)) {
        return false;
    }
    //fetch the drop location
    DropLocation loc = support.getDropLocation();
    //check if there is a component at the drop location
```

```
    return editorPanel.validateDropLocation(loc);
}
```

Pour valider la position de la souris, la fonction va vérifier que la souris n'est au dessus de aucun bloc déjà présent de plus elle va vérifier si la souris se trouve dans une zone magnétique pour pouvoir afficher les bordures d'aide sur les blocs.

La deuxième action de la classe est de créer un nouveau bloc à partir des données transférées avec la fonction `importData()`.

Listing 8. Fonction de crétaion du bloc si la position est ok

```
@Override
public boolean importData(TransferSupport support) {
    if(!canImport(support)) {
        return false;
    }
    //fetch transferable data
    Transferable t = support.getTransferable();
    try {
        String data = (String) t.getTransferData(DataFlavor.stringFlavor);
        // cree un block avec les donnees transferees
        DropLocation loc = support.getDropLocation();
        ProtoBlock block;
        if(data.equals("ACTIONL_AL001")) {
            block = new ProtoBlockLogique(data);
        } else if (data.equals("ACTIONL_INSTANT")) {
            block = new ProtoBlockInstant(data);
        } else {
            block = new ProtoBlockAction(data);
        }
        //on cree un drag listener pour bouger les block apres les avoir inserer
        DragEListener dragL = new DragEListener();
        block.addMouseListener(dragL);
        block.addMouseMotionListener(dragL);
        //on ajoute le menu popup
        block.setComponentPopupMenu(new ProtoBlockPopupMenu(block));
        editorPanel.addBlock(block, loc.getDropPoint());
        for(Component c : editorPanel.getComponents()) { // reset les borders d'aide au
            drop des blocks
            ProtoBlock b = (ProtoBlock) c;
            b.resetBorder();
        }
        editorPanel.revalidate(); //important sinon le panel s'actualise pas
    } catch (UnsupportedFlavorException e) {
        e.printStackTrace();
        return false;
    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

Dans cette fonction, on verifie que l'on peut créer un bloc à la position de la souris puis on recupere les données transférées avec `getTransferData()` puis dans le prototype suivant le nom de l'action transférée on crée un type de bloc différent. On ajoute ensuite le listener qui permet de deplacer un bloc dans le panneau édition, et enfin on essaye de relier le bloc nouvellement créer à un

autre bloc déjà présent, enfin on enlève les bordures d'aide sur les blocs.

4.3 Intégration dans Metaciv

L'intégration de l'éditeur dans Metaciv consiste à ajouter les actions au prototype ainsi que les modifications dans le plan en bougeant les blocs.

4.3.1 Ajout des actions aux blocs

La première étape pour ajouter les actions aux blocs est de modifier l'arbre d'actions pour qu'ils contiennent la vraie liste des actions disponibles et non pas une liste factice. Il suffit de modifier l'initialisation de l'arbre d'actions pour qu'ils contiennent les noms des actions.

Listing 9. Ajout des actions au bloc

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("Listes des actions de  
l'éditeur de plan");  
DefaultMutableTreeNode actions = new DefaultMutableTreeNode("Actions");  
DefaultMutableTreeNode actionsL = new DefaultMutableTreeNode("Actions logiques");  
for (int i = 0; i < Configuration.actions.size(); i++){  
    if(!Configuration.actions.get(i).isDeprecated()){  
        String actionPerformed = Configuration.actions.get(i).getName().split("\\\\.") [  
            Configuration.actions.get(i).getName().split("\\\\.").length - 1];  
        DefaultMutableTreeNode action = new DefaultMutableTreeNode(actionPerformed); //on  
            cree un nouveau noeud contenant le nom de l'action  
        if(actionName.startsWith("L_")){ // si action logique, on ajoute dans l'arbre  
            d'actions logiques  
            actionsL.add(action);  
        } else if(actionName.startsWith("A_")) { // si action normale, on ajoute dans  
            l'arbre d'actions normales  
            actions.add(action);  
        }  
    }  
}  
root.add(actions);  
root.add(actionsL);
```

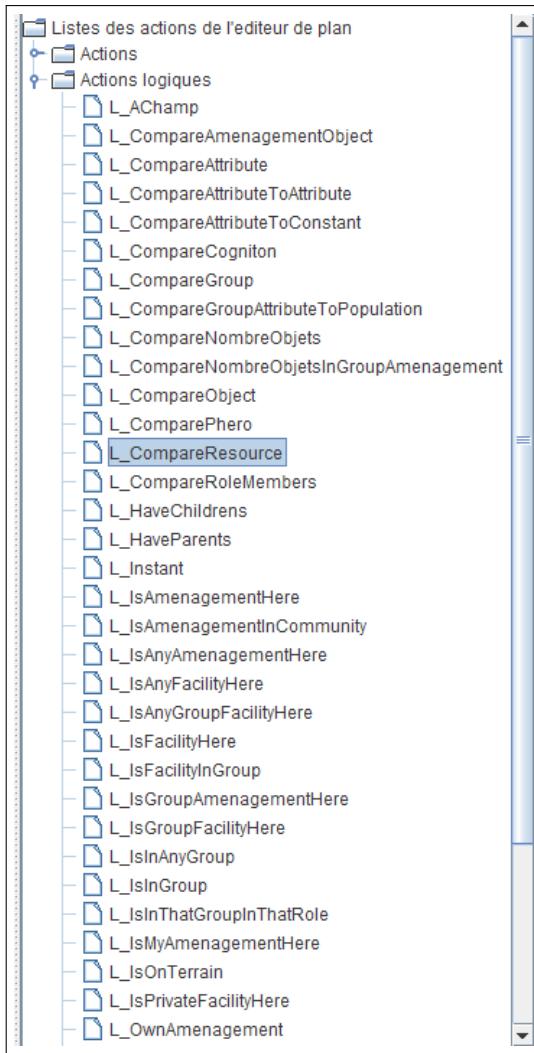


Figure 12. Liste des actions

Le code ci dessus initialise l'arbre des actions avec le nom de toutes les actions qui ne sont pas dépréciées. On construit un nœud pour chaque action, le nom du nœud étant le nom de l'action. On range par la suite les actions dans les bonnes catégories suivant leur initiale.

On modifie par la suite le `TransferHandler` du panneau d'édition pour qu'il crée un objet `Action` à partir d'un nom d'action.

Listing 10. Création de Action par une factory

```
String[] s = new String[1];
s[0] = data; // on recupere le nom de l'action et on la met dans ce tableau utilise
pour creer l'action
Action action =
    Action.actionFactory(s, Configuration.getSchemaCognitifEnCourEdition());
```

L'action est créée par une Factory utilisée à l'origine par l'ancien éditeur de plan pour créer des actions.

Pour créer un bloc on regarde le nombre d'actions qu'une action peut contenir.

Listing 11. Reagard du nombre d'actions qu'une action peut contenir

```
if(action.getNumberActionSlot() == 2) { // action logique
    block = new AEeditorBlockLogique(action);
} else if(action.getNumberActionSlot() == 0) { // action simple
```

```

block = new AEeditorBlockAction(action);
} else { // actions de type repeat ou instant
    block = new AEeditorBlockInstant(action);
}

```

Une action simple ne peut pas contenir d'autres actions tandis qu'un action logique « Si, Sinon » peut en contenir deux, enfin, une action logique « Répéter » n'a pas de limite. Il suffit de vérifier le nombre d'actions possibles et de créer le bloc correspondant à l'action à représenter.

4.3.2 Modifications sur le plan

Il y a plusieurs opérations qui modifie le plan et ces opérations peuvent soit ajouter des actions aux plans, soit en enlever, soit les déplacer. Pour simplifier les choses, les déplacements d'actions dans le plan consisteront à des suppressions puis des ajouts au nouvel endroit dans le plan. Il suffit alors de gérer l'ajout d'action dans le plan et la suppression.

4.3.2.1 Ajout d'action dans le plan L'ajout d'une action dans le plan se fait dans les fonctions qui permettent de relier et positionner les blocs entre eux.

Listing 12. Fonction d'ajout d'une action dans le plan

```

if(plan != null && action != null) { // si ce block contient une action et que l'on
    ajoute au plan
    for(Action a : plan.getActions()) {
        if(a.equals(action)) { // si l'action de ce block est directement accessible
            depuis le plan
            plan.addActionAfter(bl.getAction(), a); // on ajoute l'action de notre
                nouveau block directement apres
            bl.addChildrenActionsPlan(plan);
            return;
        }
    }
    Action act = getParentAction(this.action);
    if(act != null) { // si on on a trouve cette action
        act.addActionAfter(bl.getAction(), action); // on ajoute l'action de notre
            nouveau block apres l'action de ce block dans l'action parente qu'on vient
            de chercher
        bl.addChildrenActions(act);
    }
}

```

Dans la fonction append(), qui ajoute un bloc en dessous d'un autre, on va ajouter le code ci dessus. La fonction vérifie que le plan passé en paramètre n'est pas égal à null, on peut utiliser cette fonction en passant un plan null pour ainsi ne pas modifier le plan inutilement ce qui est utile quand on veut repositionner les fils d'un bloc logique que l'on vient de redimensionner. Après avoir vérifier qu'on veut modifier le plan, il faut vérifier si l'action du bloc parent est directement accessible depuis le plan. Si c'est le cas, on ajoute l'action du bloc fils (le bloc bl dans le code au dessus 12) après l'action du bloc parent dans le plan. On ajoute ensuite les actions des fils du bloc fils au plan avec la fonction addChildrenActionsPlan().

Listing 13. Fonction d'ajout de l'action du fils

```

protected void addChildrenActionsPlan(NPlan plan) {
    if(hasChild()) {
        plan.addActionAfter(fils.getAction(), action);
        fils.addChildrenActionsPlan(plan);
    }
}

```

}

Cette fonction ajoute récursivement l'action du fils « Append » d'un bloc au plan, l'action du fils s'insérant juste après celle du père dans le plan.

Si l'action du bloc parent n'est pas accessible directement depuis le plan il faut alors chercher dans le plan l'action qui contient l'action du bloc parent. On cherche cette action avec la fonction `getParentAction()` qui remonte récursivement les parents de chaque bloc en passant en paramètre l'action du bloc qui exécute la fonction, la fonction étant surchargé par les blocs logiques pour renvoyer leur action si l'action en paramètre correspond à l'action d'un de leur fils logiques.

Listing 14. Fonction d'un bloc normal

```
public Action getParentAction(Action thisAction) {
    if(parent != null) {
        return parent.getParentAction(action);
    }
    return null;
}
```

Au dessus la fonction d'un bloc normal.

Listing 15. Fonction d'un bloc logique "Répéter"

```
@Override
public Action getParentAction(Action thisAction) {
    if(filsInstant != null && filsInstant.getAction().equals(thisAction)) {
        return this.action;
    } else {
        return parent.getParentAction(this.action);
    }
}
```

Au dessus la fonction d'un bloc logique « Répéter », on voit que si l'action reçu en paramètre est identique à celle du fils logique, l'action de ce bloc logique est l'action parente recherchée.

Si on a trouvé l'action parente du bloc parent, on a plus qu'a suivre le même procédé que pour ajouter une action directement au plan. La seule différence étant que la fonction `addChildrenActionsPlan()` est remplacé par `addChildrenActions()`, la deuxième fonction prenant en paramètre une action au lieu d'un plan dans la première.

Pour les blocs logiques, il faut modifier de la même façon les fonctions d'ajout de fils dans les emplacements logiques.

Pour les blocs « Répéter » il suffit d'ajouter l'action du bloc fils « Répéter » ainsi que les actions de ses propres fils comme sous action à l'action du bloc logique.

Listing 16. Code de l'ajout d'actions dans une action logique "Répéter"

```
if(plan != null && action != null) { // si ce block contient une action et que l'on
    ajoute au plan
    action.addSousAction(bl.getAction());
    bl.addChildrenActions(action); // ajoute les actions fils du block ajoute a cet
    action
}
```

Pour les blocs « Si, Sinon » il ne peut y avoir que deux sous actions, dans le cas où on ajoute un fils « Si » il faut vérifier si il a déjà un bloc « Sinon » attaché au bloc logique, si c'est le cas il faut insérer la nouvelle action avant celle du bloc « Sinon ».

Si ce n'est pas le cas et aussi quand on ajoute un bloc « Sinon », on réalise exactement le même code que pour les blocs « Répéter ».

4.3.2.2 Suppresion d'actions dans le plan La suppression des actions du plan suit le même principe que l'ajout en plus simple car il n'est pas nécessaire de faire des cas spéciaux pour les blocs logiques.

Listing 17. Fonction de suppression des actions

```
public void removeActionFromPlan(NPlan plan) {
    for(Action a : plan.getActions()) {
        if(a.equals(action)) { // si l'action de ce block est directement accessible
            depuis le plan
            plan.removeAction(action);
            AEditorBlock f = fils;
            while(f != null) { // tant que ya des fils
                plan.removeAction(f.getAction());
                f = f.getChild();
            }
            break;
        } else { // sinon on cherche l'action parent de l'action du block auquel on
            append
            Action act = a.getParentAction(action); // on cherche l'action parent de
            l'action de ce block
            if(act != null) { // si on on a trouve cette action
                act.removeAction(action);
                AEditorBlock f = fils;
                while(f != null) { // tant que ya des fils
                    act.removeAction(f.getAction());
                    f = f.getChild();
                }
                break;
            }
        }
    }
}
```

Là encore, on a deux cas différents, si l'action à supprimer est directement accessible depuis le plan ou pas. Si elle n'est pas accessible depuis le plan, on cherche l'action parent de l'action à supprimer.

Une fois trouvée, on supprime l'action ainsi que toutes les actions des fils « Append » du bloc dont on a supprimé l'action. Les actions fils devront alors être rattachés au plan manuellement.

L'opération de suppression a lieu quand on déplace un bloc si celui ci était relié à un autre bloc ou quand on supprime un bloc du panneau édition.

4.3.2.3 Ajout des paramètres interactifs sur les blocs Au lieu de passer par une fenêtre de dialogue pour éditer les paramètres d'une action d'un bloc, les paramètres sont directement visibles sur les blocs. Les modifications de ces paramètres s'appliquent instantanément.

Pour ce faire, nous avons créé une classe étendant JPanel qui fera office de panel contenu d'un bloc. On a réutilisé la manière dont se créer une fenêtre de dialogue d'édition dans l'ancien éditeur de plan mais au lieu d'afficher les paramètres dans une fenêtre on les ajoute dans le panel contenu.

Les paramètres sont représentés par des ComboBox contenant les valeurs possibles du paramètre. Il y a aussi pour certaines actions la possibilité d'effectuer celle ci un certain nombre de fois, par exemple pour jeter un objet par terre. Deux boutons radios sont créés pour permettre de choisir entre une valeur choisie dans une ComboBox ou bien une constante Metaciv dans une autre ComboBox pour choisir le nombre de fois à exécuter l'action.

Pour l'implémentation spécifique de la création des ComboBox et boutons radios, il faudra se référer au rapport de TER du groupe ayant réalisé l'ancien éditeur de plan.

Pour que les modifications des valeurs dans les paramètres mettent à jour les actions nous avons créer deux types de listeners qui effectuent la même chose mais qui sont de nature différentes. Un de

ces listeners sera pour les modifications sur les ComboBox tandis que l'autre sera pour le changement de radios boutons sélectionné.

Ces deux listeners vont effectuer la modification d'une action, qui est reprise telle quelle de l'ancien éditeur lorsque l'on fermait la fenêtre de dialogue d'édition.

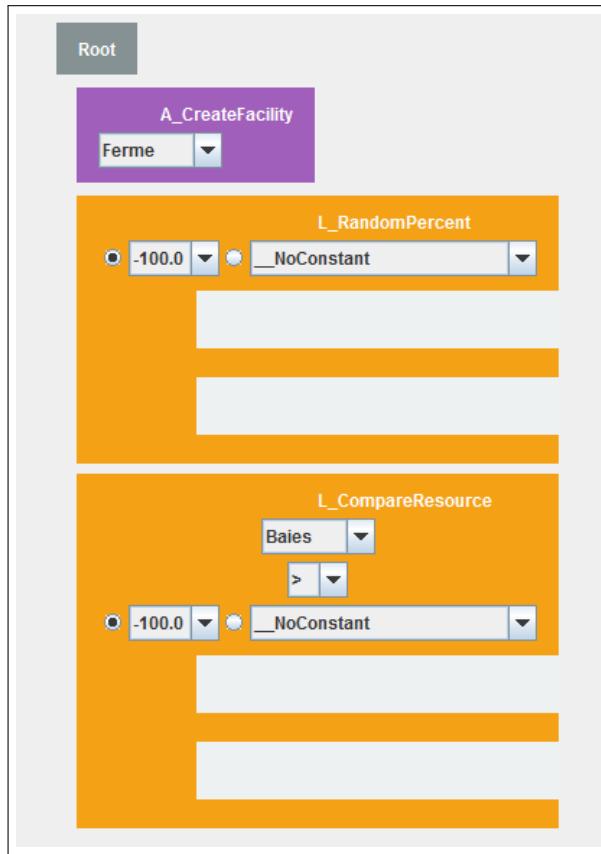


Figure 13. Blocs après ajout des paramètres interactifs

4.3.2.4 Lecture d'un plan existant Au démarrage de l'éditeur il faut afficher les éléments déjà présents dans le plan.

Jusqu'à présent, pour ajouter un bloc il nous fallait évaluer la position de la souris, hors ici on déjà la hiérarchie dans laquelle doivent se trouver les blocs. Il suffit alors de reprendre les fonctions `attach()` utilisée pour attacher des blocs entre eux et en créer une version qui utilisent un mode d'attachement au lieu d'un point pour représenter la position de la souris. La fonction réalise les mêmes modifications mais elle va utiliser le mode d'ajout passé en paramètre pour déterminer où insérer le bloc à lié.

Il y a 5 modes différents d'ajout :

1. « APPEND » pour ajouter après le bloc.
2. « PREPEND » pour ajouter avant.
3. « IF » pour ajouter un fils « Si »
4. « ELSE » pour ajouter un fils « Sinon »
5. « INSTANT » pour ajouter un fils « Répéter »

On lit le plan lorsque que l'on initialise le panneau d'édition.

Listing 18. Fonction qui ajoute une action dans le plan

```
private AEditorBlock addAction(AEditorBlock parent, Action action, int mode) {
    AEditorBlock block;
```

```

if(action.getNumberActionSlot() == 2) { // action If_else
    block = new AEeditorBlockLogique(action);
    addListenersToBlock(block);
    addBlock(parent, block, mode); // on ajoute le block cree au block parent
    addAction(block, action.getListeActions().get(0), AEeditorBlockLogique.IF); // on
        ajoute au block cree un block representant l'action If
    addAction(block, action.getListeActions().get(1), AEeditorBlockLogique.ELSE); // on
        ajoute au block cree un block representant l'action Else
} else if (action.getNumberActionSlot() < 0){ // action Instant
    block = new AEeditorBlockInstant(action);
    addListenersToBlock(block); // on ajoute le block cree au block parent
    addBlock(parent, block, mode);
    AEeditorBlock parentAdd = block;
    for(int i = 0; i < action.getListeActions().size(); ++i) {
        if (i == 0) {
            parentAdd = addAction(parentAdd,action.getListeActions().get(i),
                AEeditorBlockInstant.INSTANT); // on ajoute la premiere action au block
                Instant lui meme
        } else
            parentAdd = addAction(parentAdd,action.getListeActions().get(i),
                AEeditorBlockInstant.APPEND); // les autres actions seront ajoutees aux
                blocks de la derniere action ajoutee
    }
} else { // action normale
    block = new AEeditorBlockAction(action);
    addListenersToBlock(block);
    addBlock(parent, block, mode); // on ajoute le block cree au block parent
}
return block;
}

```

La fonction ci dessus est utilisée pour ajouter une action dans le plan. On voit que l'on utilise ici aussi le nombre de sous actions qu'une action peut posséder pour déterminer quel type de bloc créer. On créer alors un bloc, on lui ajoute les listeners pour le déplacer puis on le lie à son parent.

Dans le cas des blocs « Si, Sinon », on voit que l'on ajoute les actions « Si » et « Sinon » au bloc logique nouvellement créer en réutilisant la fonction `addAction()`.

Pour les actions « Répéter », on créer une boucle qui va fabriquer tous les blocs de toutes les sous actions de l'action « Répéter ». Pour la première itération, il faut attacher le bloc de la première sous action au bloc logique « Répéter », pour les autres actions on attache le bloc que l'on va créer au bloc que l'on a créer juste avant. On obtient le bloc créer juste avant en stockant le bloc retourné par `addAction()` qui correspond au bloc créer par cette fonction.

Enfin, pour les actions simples comme elles ne peuvent pas avoir de sous actions, on n'a pas besoin de rajouter une boucle pour ajouter les sous actions.

4.4 Conclusion de la partie éditeur de plan

Pour conclure, il reste quelques spécifications qui n'ont pas pu être implémentées par manque de temps et qui ont été jugées pas prioritaire comme par exemple la barre de recherche d'actions. L'éditeur de plan pourrait aussi être plus polissée à certains endroits comme par exemple l'alignement des paramètres dans un bloc. Des améliorations potentielles sont apparues une fois le prototype réalisé comme l'ajout d'une fonction de zoom par palié au fait qu'il est difficile d'avoir une vue d'ensemble à partir d'un certain nombre de blocs dans le plan.

5 Les sondes et les graphes par défaut

5.1 Introduction aux graphes

L'un des points importants de MetaCiv est de pouvoir avoir un maximum d'informations sur la simulation en cours. C'est pour cela que François a mis en place un système de sondes et de graphes pour pouvoir avoir ces informations. L'objectif était alors de trouver une configuration par défaut, et que celle-ci s'adapte à toutes les civilisations. L'adaptation des sondes est importante car les fichiers qui modélisent les civilisations peuvent chacun avoir une façon différente d'écrire le mot "baies", par exemple. C'était l'un des problèmes à résoudre dès le début de la tâche. Nous avons aussi discuté autour de la notion de graphes "vitaux", pour savoir lesquels étaient nécessaires par défaut, mais à une échelle simplifiée.

5.2 Prise en main via l'interface utilisateur

Pour prendre en main les sondes et leur système de configuration, nous avons dû faire quelques tests via l'interface utilisateur. Cela nous a appris beaucoup, notamment sur comment nous allions devoir cocher les cases dans nos différents panneaux (voir figure 17 et 18).

5.2.1 Lancement de la fenêtre de graphes

Pour ouvrir la fenêtre contenant tous les graphes il faut se rendre dans l'onglet Charts, puis cliquer sur le bouton Advanced (voir la zone en surbrillance sur la figure 14)

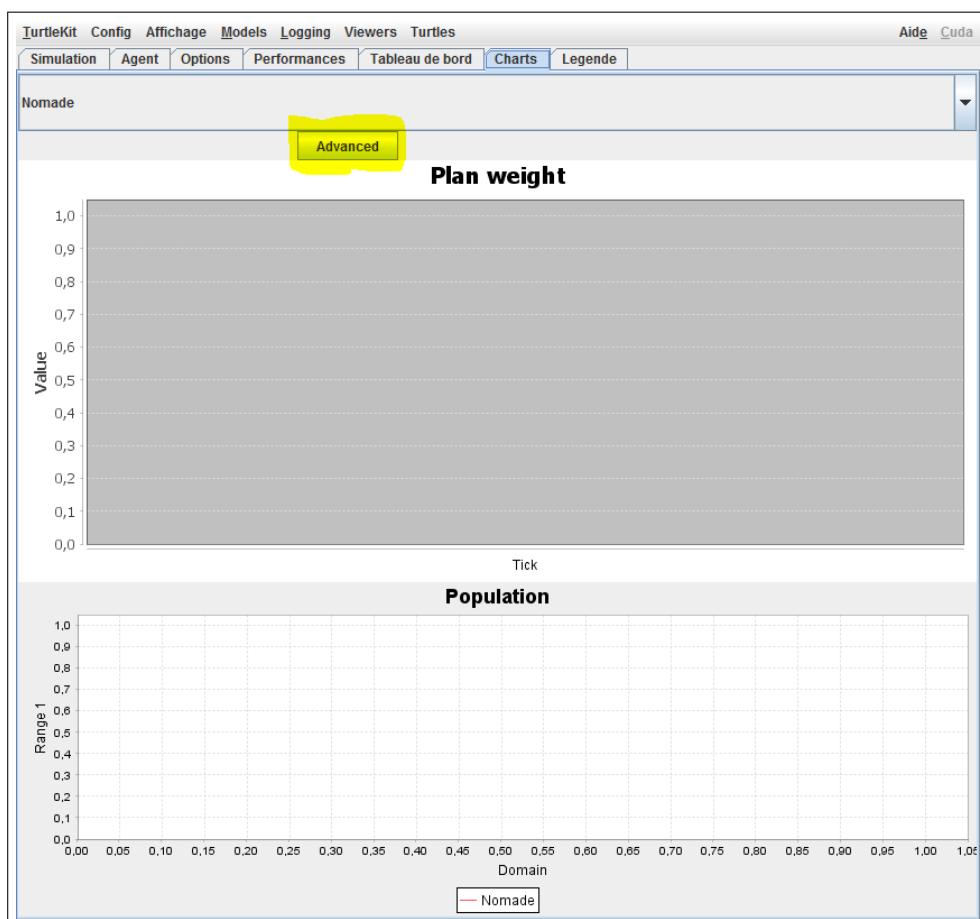


Figure 14. Onglet Charts avec les deux graphes généraux

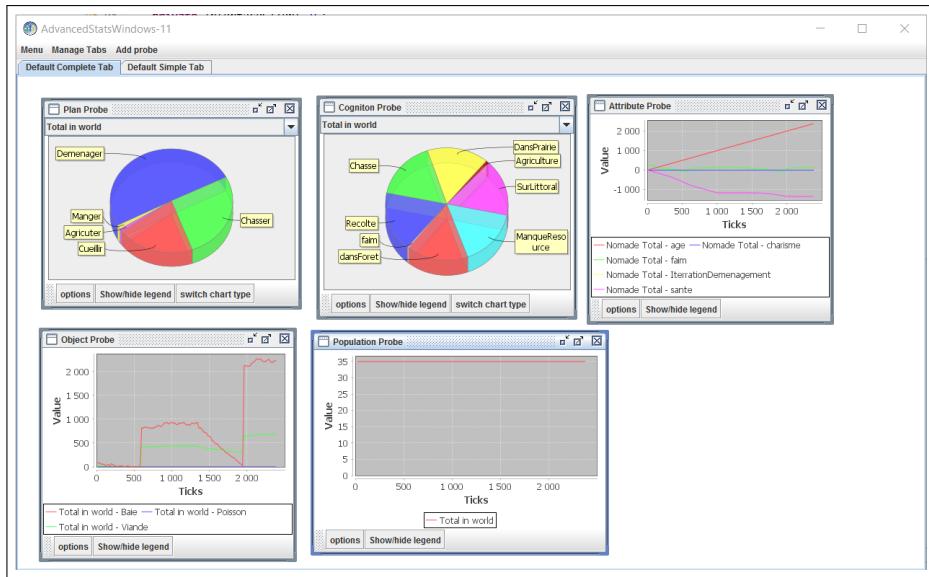


Figure 15. Fenêtre contenant tous les types de graphes possibles

5.2.1.1 Ajout et configuration de sondes Dans la figure 15, nous pouvons remarquer un menu dans la barre en haut qui se nomme **Add probe**. C'est via celui-ci que nous avons ajouté les différents graphes sur cette figure. Lorsque nous voulons ajouter un graphe, une fenêtre de configuration s'ouvre (voir figure 16).

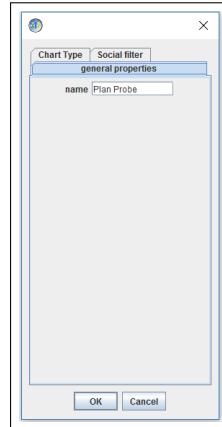


Figure 16. Fenêtre de configuration d'un graphe, ici pour les Plan

Suivant les graphes, nous avons des similitudes de configuration. Par exemple, toutes les sondes pour les graphes, à l'exception d'une, peuvent suivre ou traquer tous ceux qui a dans la simulation (voir la figure 17). Pour l'exception, il faut lui préciser la civilisation à traquer (voir la figure 21). Par la suite, il y a aussi des similitudes au niveau des types de graphes. Certains sont configurables comme pour les graphes de plan et de cogniton, où nous pouvons choisir entre un graphe en camembert et un autre en bâtonnets (voir la figure 18).

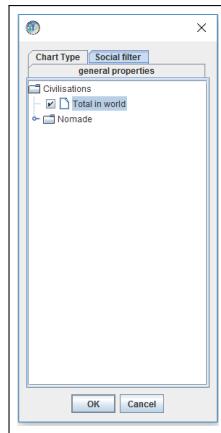


Figure 17. Onglet Social filter avec une arborescence représentant les différents groupes à traquer



Figure 18. Onglet Chart Type avec l'option "bâtonnets" ou l'option "camembert"

La première particularité fut avec la configuration des sondes Attribut.

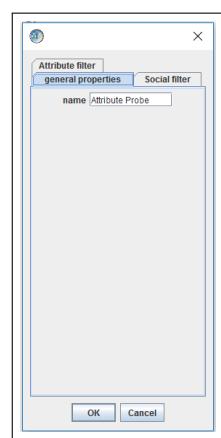


Figure 19. Fenêtre de configuration d'une sonde pour un graphe d'attributs

Celles ci ont une particularité, c'est qu'elles ne peuvent fonctionner que si elles ont une civilisation cochée dans l'arbre des groupes (voir la figure 21). Dans l'onglet Attribute Filter (voir figure 20), nous pouvons sélectionner tous les attributs de la civilisation que nous souhaitons traquer.



Figure 20. Onglet Attribute Filter toutes les cases cochées

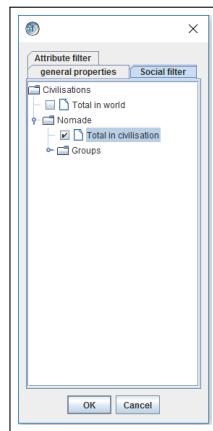


Figure 21. Onglet Social Filter avec une civilisation cochée pour que le graphe soit fonctionnel

Concernant la configuration des sondes pour les objets, il faut se rendre dans l'onglet Object Filter, puis sélectionner tous les objets voulus (voir figure 22). Ces objets sont définis dans le modèle de la simulation, cela va avoir une importance lors de la partie programmation.

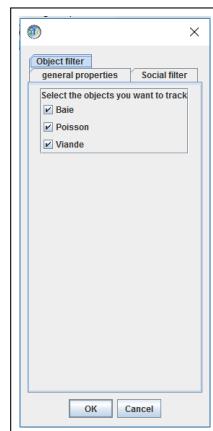


Figure 22. Fenêtre de configuration pour un graphe d'objet. Onglet Object Filter avec tous les objets définis dans le modèle cochés

Pour configurer une sonde qui traque la population, rien de plus simple. Il suffit simplement de définir la population à traquer et le tour est joué (voir figure 23).

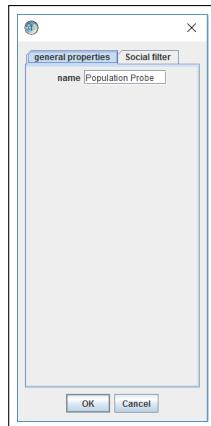


Figure 23. Fenêtre de configuration pour un graphe de population

Une fois toutes les configurations effectuées, on peut s'apercevoir d'un point important pour la partie programmation, les sondes se superposent (voir figure 24). En effet, par défaut elles ont une taille et un emplacement qui sont les mêmes pour toutes les sondes. Donc pour pallier à ce problème lors de la programmation des graphes, il faudra définir l'emplacement et la taille de ces graphes.

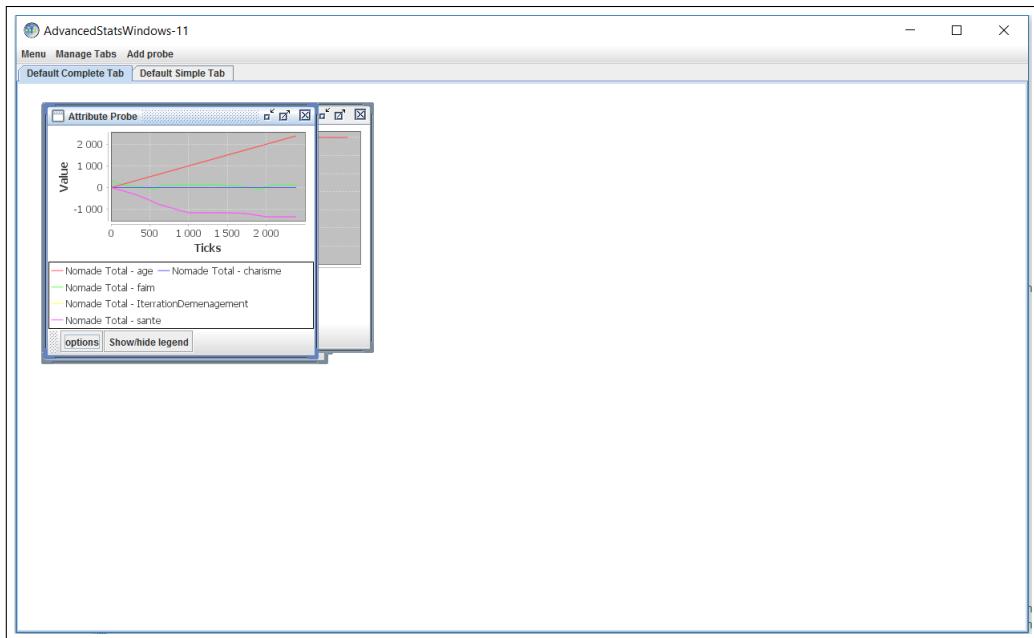


Figure 24. Fenêtre des sondes, avec un problème de placement par défaut

Une fois toutes les configurations effectués nous obtenons le résultat de la figure 15. Nous remarquons que les données sont plus significatives sous forme de graphe. Cela permet à l'utilisateur d'avoir un retour précis de sa simulation. Maintenant que nous connaissons la partie utilisateur, nous allons rentrer dans la partie programmation.

5.3 Programmation des sondes par défaut

Nous avons vu lors de nos tests utilisateur qu'il y avait quelques exceptions qu'il fallait prendre en compte pour le bon fonctionnement des sondes par défaut une fois la simulation lancée. Nous avons aussi remarquer des similitudes entre certaines configurations de sondes. Concernant l'implémentation, il y a déjà une base sur laquelle nous pouvons nous appuyer pour faire notre configuration. Il y a eu une phase de compréhension de toute cette base pour pouvoir faire notre tâche de façon à

utiliser un maximum les objets Java que nous avons sous la main.

5.3.1 La création des onglets

La première étape était de cibler la classe gérant l'ouverture de la fenêtre des graphes avancés, qui est la classe `AdvancedStatsWindows`. Une fois que nous l'avons trouvé, il fallait vérifier que le système de chargement des sondes, si elle existe déjà, fonctionnait. Car oui, la base sur laquelle nous sommes partis contient un système de sauvegarde et de chargement. Il fallait voir si avec une configuration de graphe déjà sauvegarder le système arrivait à la récupérer. Nous avons pu voir que cela fonctionnait, donc pour nous, notre configuration par défaut venait si il n'y avait pas de chargement depuis des fichiers sauvegardés.

Pour la création des graphes nous avons dû commencer par modifier la fonction `loadDefaultDesktop()` qui est en charge de la création des onglets par défaut lorsqu'il n'y a pas de fichiers sauvegardés.

Listing 19. Fonction `loadDefaultDesktop()`

```
private void loadDefaultDesktop() {
    createDefaultSimpleTab();
    createDefaultCompleteTab();
    createEmptyTab("Personal Tab");
}
```

5.3.2 Onglet par défaut simple

Dans le listing 19, nous appelons deux fonctions qui vont créer un onglet par défaut, simple (voir figure 25), puis un complexe (voir figure 26). Le dernier appelle à la fonction `createEmptyTab()` créer un onglet vierge où l'utilisateur peut mettre tous les graphes qu'il souhaite. Dans le premier onglet, nous avons décidé de mettre simplement 3 graphes, un traquant la population, un autre les plans et enfin un dernier traquant les objets. Nous avons choisi ces trois graphes en particulier car nous pensons qu'ils sont à avoir sous la main pour avoir les informations les plus élémentaires concernant la simulation.

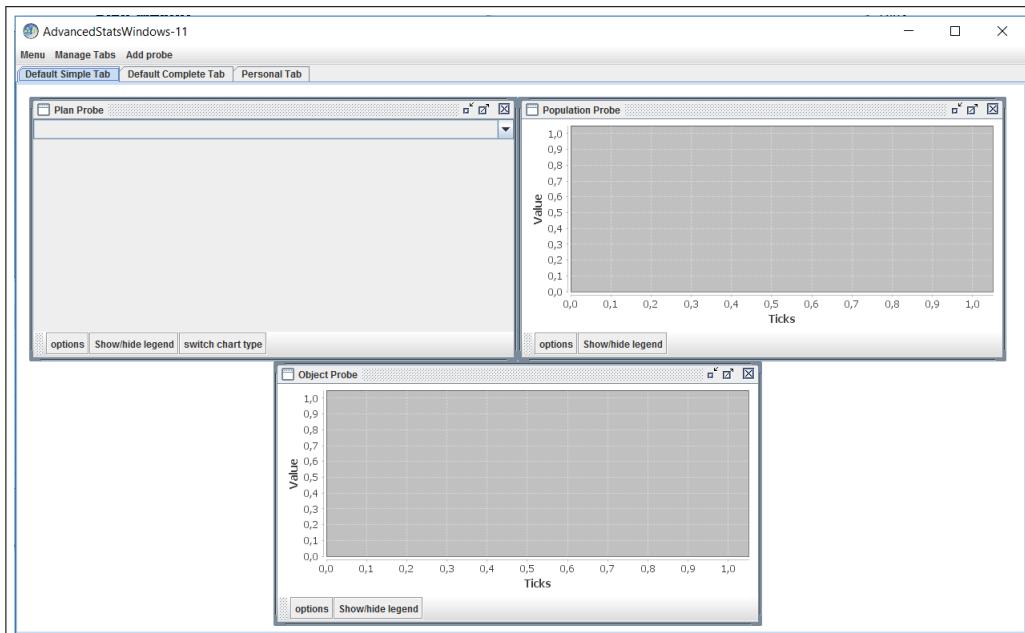


Figure 25. Fenêtre des sondes, onglet par défaut simple en version final

Dans la figure 25 le placement et la taille ont été configurés pour avoir un affichage claire et éviter ainsi le chevauchement qui nous faisait défaut au début.

La création de cet onglet est gérée par la fonction `createDefaultSimpleTab()`.

Listing 20. Fonction `loadDefaultDesktop()`

```
/*
 * Creation d'un panel simple avec les Probs classiques
 *
 * @return
 */
public JDesktopPane createDefaultSimpleTab() {
    int widthProb = width / 2 - 30;
    int heightProb = height / 2 - 30;

    //creation des differentes sondes
    ObjectProbeWidget obj = new ObjectProbeWidget();
    PopulationProbeWidget pop = new PopulationProbeWidget();
    PlanProbeWidget pla = new PlanProbeWidget();
    //configuration par defaut
    pla.getSocFilter().checkTotalInWorld();
    pla.getChartTypeSelector().getPie().setSelected(true);
    pla.applyOptions();
    pop.getSocFilter().checkTotalInWorld();
    pop.applyOptions();
    obj.getSocFilter().checkTotalInWorld();
    obj.getObjFilter().checkAllObjects();
    obj.applyOptions();
    //positionnement des graphes
    obj.setSize(widthProb, heightProb);
    pop.setSize(widthProb, heightProb);
    pla.setSize(widthProb, heightProb);

    pla.setLocation(15, 15);
    int pos = widthProb + 15;
    pop.setLocation(pos, 15);
    pos = heightProb + 15;
    obj.setLocation(15 + widthProb / 2, pos);

    JDesktopPane result = new JDesktopPane();
    addWidgetToTab(obj, result);
    addWidgetToTab(pop, result);
    addWidgetToTab(pla, result);
    tabs.addTab("Default Simple Tab", result);
    return result;
}
```

Dans le listing 20, nous commençons par calculer la taille de notre graphe en fonction de la largeur et de la hauteur de l'écran. Nous créons par la suite les différentes sondes et nous les configurons. La fonction `checkTotalInWorld()` (voir listing 21) va aller cocher dans l'arbre la racine pour pouvoir ainsi traquer tous les agents dans la simulation. La fonction `getPie().setSelected(true)` (voir listing 22) va quant à elle sélectionner le type de graphe en camembert et le cocher. La fonction `checkAllObjects()` (voir listing 23) va cocher tous les objets présent dans le modèle pour pouvoir les traquer. Et enfin la fonction `applyOptions()` permet d'appliquer les options sur l'objet Java, pour que celui prenne en compte les changements qui ont été opérés.

Listing 21. Fonction `checkTotalInWorld()` dans la classe `WidgetPanelSocialFilter`

```
public void checkTotalInWorld(){
    this.firstNode.setSelected(true);
}
```

Dans le listing 21, this.firstNode est le premier nœud de l'arbre que nous stockons lors dans une donnée membre lors de la création de l'arbre à la lecture du modèle de la civilisation. Ce nœud représente la case Total in world dans la fenêtre de configuration. Nous avons plus qu'à le setSelected(true) pour qu'il soit coché par défaut. Nous n'avons pas trouver de moyen d'accès à cette variable sans passer par une assignation à une donnée membre de la classe. Nous avons mis en place une autre technique pour pouvoir stocker les cases représentant les différentes civilisations, nous l'expliquerons un peu plus loin.

Listing 22. Fonction getPie() dans la classe WidgetPanelChartType

```
public JCheckBox getPie(){
    for (JCheckBox jc : this.check){
        if (jc.getText().equals(CTYP_Pie)){
            return jc;
        }
    }
    return null;
}
```

Concernant le listing 22 nous sommes dans la classe WidgetPanelChartType. Cette classe a comme donnée membre un tableau contenant tous les types disponibles de graphes possibles. Cela permet à l'utilisateur de pouvoir rajouter des types si il le souhaite. Ces types sont définis ar une String unique et nous allons nous en servir comme ID pour sélectionner notre type à cocher par défaut. Pour cela, nous parcourons tout nos types de graphes dans ce tableau, et nous retournons celui qui a l'ID voulu. Dans le listing 20, nous n'avons plus qu'à le setSelected(true) pour qu'il soit coché.

Listing 23. Fonction checkAllObjects() dans la classe WidgetPanelObjects

```
public void checkAllObjects(){
    for (JCheckBox jc : check){
        jc.setSelected(true);
    }
}
```

Enfin dans le listing 23 nous sommes dans la classe WidgetPanelObjects qui correspond à l'onglet dans la figure 22. Dans cette fonction nous parcourons un tableau contenant tous les objets définis dans le modèle. Ce tableau est rempli lors de la création de l'onglet, il est rempli avec les valeurs définies dans le fichier du modèle.

Une fois l'onglet par défaut simple achevé, nous devions nous pencher sur l'onglet par défaut complet.

5.3.3 Onglet par défaut complet

Cet onglet devait comporter tous les graphes possibles, et traquant toutes les données possibles. Pour cela il nous fallait rajouter deux graphes supplémentaires, un graphe portant sur les cognitons et un autre sur les attributs. La configuration du graphe des attributs nous a posé problème car nous devions modifiée notre méthode pour sélectionner, non plus Total in world, mais Total in Civilisations. Car la sonde ne peut marcher que si nous lui passons les civilisations à traquer.

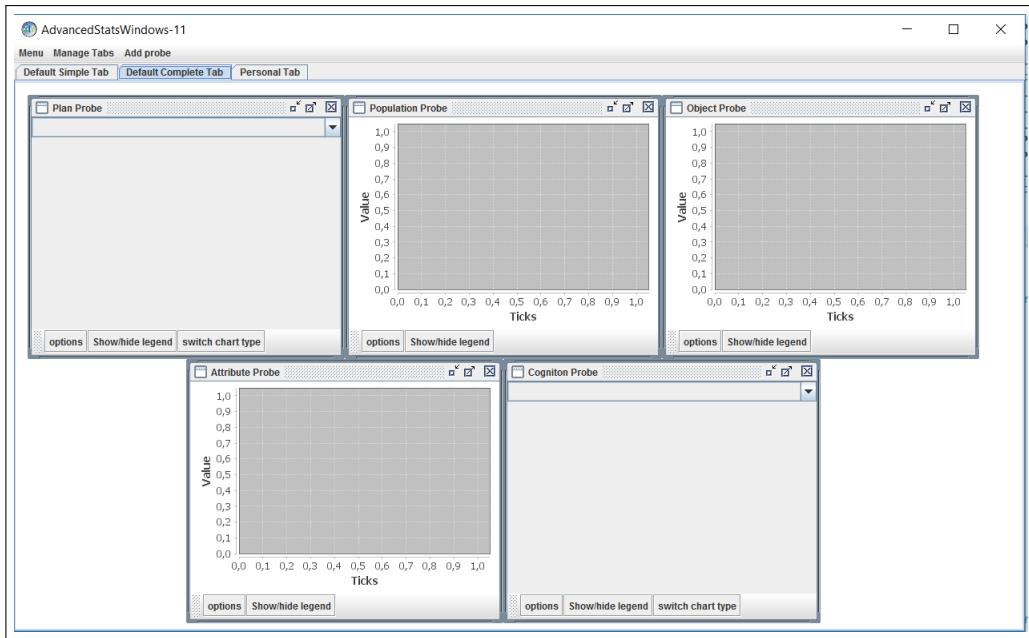


Figure 26. Fenêtre des sondes, onglet par défaut complet en version final

Dans la figure 26 nous avons placé les graphes sans ordre précis, mais toujours de façon claire.

Listing 24. Fonction `createDefaultCompleteTab()` dans la classe `AdvancedStatsWindows`

```
/**
 * Creation d'un panel plus complet avec toutes les probs possibles
 *
 * @return
 */
public JDesktopPane createDefaultCompleteTab() {
    int widthProb = width / 3 - 30;
    int heightProb = height / 2 - 30;

    //creation des sondes
    ObjectProbeWidget obj = new ObjectProbeWidget();
    PopulationProbeWidget pop = new PopulationProbeWidget();
    PlanProbeWidget pla = new PlanProbeWidget();
    AttributeProbeWidget att = new AttributeProbeWidget();
    CognitonProbeWidget cog = new CognitonProbeWidget();
    //configuration
    pla.getSocFilter().checkTotalInWorld();
    pla.getChartTypeSelector().getPie().setSelected(true);
    pla.applyOptions();
    pop.getSocFilter().checkTotalInWorld();
    pop.applyOptions();
    obj.getSocFilter().checkTotalInWorld();
    obj.getObjFilter().checkAllObjects();
    obj.applyOptions();
    att.getSocFilter().checkTotalCiv();
    att.getAttrFilter().checkAllAttributes();
    att.applyOptions();
    cog.getSocFilter().checkTotalInWorld();
    cog.getChartTypeSelector().getPie().setSelected(true);
    cog.applyOptions();
    //placement des graphes
    obj.setSize(widthProb, heightProb);
```

```

pop.setSize(widthProb, heightProb);
pla.setSize(widthProb, heightProb);
att.setSize(widthProb, heightProb);
cog.setSize(widthProb, heightProb);

pla.setLocation(15, 15);
int pos = widthProb + 15;
pop.setLocation(pos, 15);
pos += widthProb;
obj.setLocation(pos, 15);
pos = heightProb + 15;
att.setLocation(widthProb / 2 + 15, pos);
cog.setLocation(widthProb + widthProb / 2 + 15, pos);

JDesktopPane result = new JDesktopPane();
addWidgetToTab(obj, result);
addWidgetToTab(pop, result);
addWidgetToTab(pla, result);
addWidgetToTab(att, result);
addWidgetToTab(cog, result);
tabs.add("Default Complete Tab", result);
return result;
}

```

Dans le listing 24 nous avons suivi la même méthode que pour l'onglet simple, sauf que nous avons dû configurer la sonde du graphe d'attributs convenablement. Pour cela, nous avons fait une fonction `checkTotalCiv()`, qui va checker toutes les civilisations du modèle.

Listing 25. Fonction `checkTotalCiv()` dans la classe `WidgetPanelSocialFilter`

```

public void checkTotalCiv(){
    for(CheckNode c : civNodes){
        c.setSelected(true);
    }
}

```

Dans le listing 25, nous parcourons le tableau de civilisations que nous remplissons lors de la création de l'onglet `WidgetPanelSocialFilter` avec les civilisations. Puis nous cochons toutes ces civilisations pour pouvoir traquer leurs attributs.

Listing 26. Fonction `checkAllAttributes()` dans la classe `WidgetPanelAttributes`

```

public void checkAllAttributes(){
    for (Civilisation c : Configuration.civilisations){
        for (JCheckBox jc : check.get(c.getNom())){
            jc.setSelected(true);
        }
    }
}

```

Dans le listing 26, nous parcourons le tableau des civilisations présentes dans la configuration. Puis, pur chacune d'entre elles nous sélectionnons tout leurs attributs.

Listing 27. Extrait de la classe `WidgetPanelSocialFilter` avec le constructeur et les données membres permettant un suivi pour les civilisations et le monde entier

```

public class WidgetPanelSocialFilter extends JPanel {
    JTree tree;
    CheckNode rootNode;

```

```
CheckNode firstNode;
//tab des nodes de chaque civ
ArrayList<CheckNode> civNodes = new ArrayList<>();
public static final String totalInWorld = "Total in world";
public static final String totalInCiv = "Total in civilisation";
public static final String total = "Total";
public static final String totalForGroupType = "Total for group type";
public static final String separateCount = "Separate count for instances";

public WidgetPanelSocialFilter() {

    this.setLayout(new BorderLayout());
    rootNode = new CheckNode("Civilisations", true, false, CheckNode.LABEL_MODE);
    firstNode = new CheckNode(totalInWorld);
    rootNode.add(firstNode);
    for (Civilisation c : Configuration.civilisations) {
        CheckNode civNode = new CheckNode(c.getNom(), true, false,
            CheckNode.LABEL_MODE);
        CheckNode civCn = new CheckNode(totalInCiv);
        civNode.add(civCn);
        civNodes.add(civCn);
        CheckNode grpTitle = new CheckNode("Groups", true, false,
            CheckNode.LABEL_MODE);
        civNode.add(grpTitle);
        for (GroupModel g : c.getCognitiveScheme().getGroups()) {
            CheckNode grpNode = new CheckNode(g.getName(), true, false,
                CheckNode.LABEL_MODE);
            grpNode.add(new CheckNode(totalForGroupType));
            grpNode.add(new CheckNode(separateCount));
            CheckNode rleTitle = new CheckNode("Roles", true, false,
                CheckNode.LABEL_MODE);
            grpNode.add(rleTitle);
            for (String r : g.getCulturons().keySet()) {
                CheckNode rleNode = new CheckNode(r, true, false,
                    CheckNode.LABEL_MODE);
                rleNode.add(new CheckNode(totalForGroupType));
                rleNode.add(new CheckNode(separateCount));
                rleTitle.add(rleNode);
            }
            grpTitle.add(grpNode);
        }
        rootNode.add(civNode);
    }
    tree = new JTree(rootNode);

    tree.setCellRenderer(new TreeRenderer());
    tree.getSelectionModel().setSelectionMode(
        TreeSelectionModel.SINGLE_TREE_SELECTION
    );
    //tree.putClientProperty("JTree.lineStyle", "Angled");
    tree.addMouseListener(new NodeSelectionListener(tree));
    JScrollPane sp = new JScrollPane(tree);
    add(sp, BorderLayout.CENTER);

}
...
}
```

5.3.4 Onglet personnel

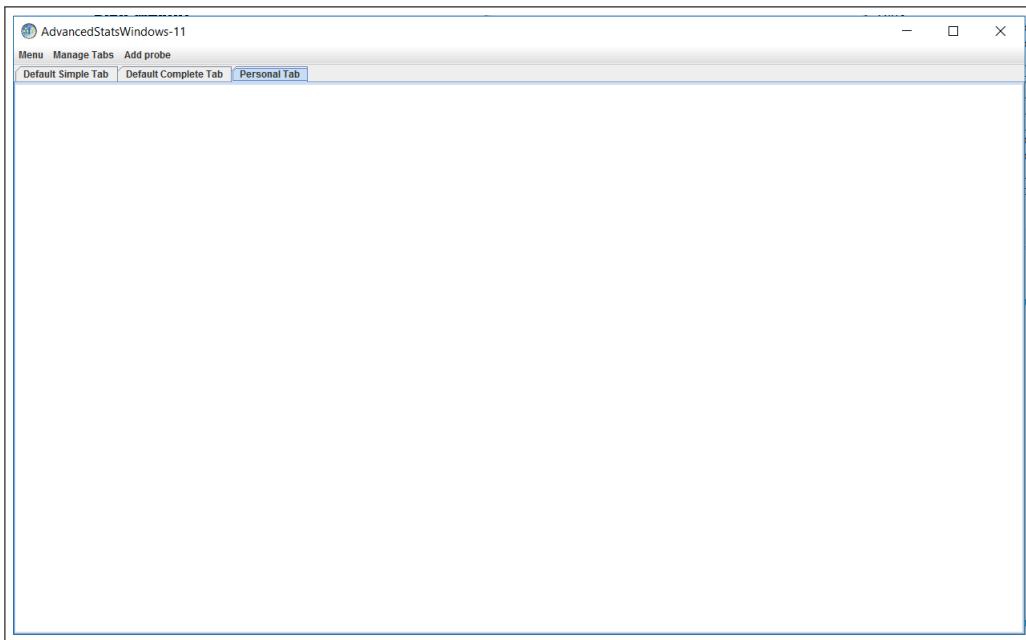


Figure 27. Fenêtre des sondes, onglet par défaut complet en version final

Listing 28. Fonction `createEmptyTab(String title)` dans la classe `AdvancedStatsWindows`

```
public JDesktopPane createEmptyTab(String title) {
    JDesktopPane result = new JDesktopPane();
    tabs.add(title, result);
    return result;
}
```

La fonction précédente crée un onglet vierge avec un titre donné en paramètre. Nous l'utilisons pour faire notre onglet personnel. Cette onglet sera la zone d'expérimentation de l'utilisateur.

5.3.5 Sauvegarde de la configuration par défaut

Une des tâche que nous devions faire était la sauvegarde de cette configuration par défaut. Il fallait pour cela trouver le moment propice à la sauvegarde. Nous avons décidé de mettre la sauvegarde juste après la création de la configuration par défaut. Nous avons essayé d'autres endroits, mais cela sauvegardait plus souvent qu'il ne le fallait. Par exemple, nous avons placé la sauvegarde dès que la fenêtre s'ouvrait, mais cela était inutile, car on sauvegardait une configuration qui n'avait pas été touchée.

Listing 29. Fonction `loadDefaultDesktop()` dans la classe `AdvancedStatsWindows` avec la fonction de sauvegarde

```
private void loadDefaultDesktop() {
    createDefaultSimpleTab();
    createDefaultCompleteTab();
    createEmptyTab("Personal Tab");
    /**
     * sauvegarde des tabs par defaut pour eviter la creation des objets a chaque
     * fois
     */
    System.out.println("save all tabs");
```

```
        for (int i = 0; i < tabs.getTabCount(); i++) {
            saveTab(tabs.getTitleAt(i), (JDesktopPane) tabs.getComponentAt(i));
        }
    }
```

Dans le listing 29, nous parcourons tout nos onglet et nous les sauvegardons dans un fichier avec la fonction `saveTab()`. Nous retrouvons ces fichiers dans un dossier `ADVStats` (voir figure 28). Ce dossier doit porter cette nomenclature pour que `MetaCiv` retrouve les fichiers de graphes. Sinon, il va céer une nouvelle configuration par défaut.

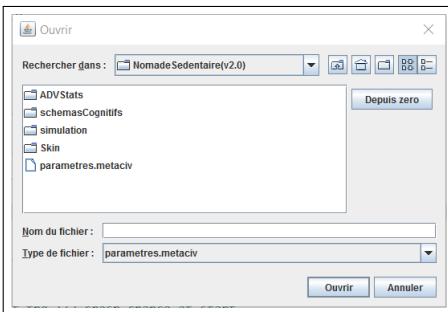


Figure 28. Dossier ADVStats contenant tous les fichiers concernant les graphes

5.4 Conclusion de la partie des sondes par défaut

Pour conclure, nous pensons que ces graphes sont très important lors d'une simulation. C'est pour cela qu'il fallait faire une configuration par défaut, pour que l'utilisateur puisse voir comment on configure une sonde pour un graphe. Lors de cette partie, il était essentiel de bien définir les graphes prioritaires. Le plus dur fut de bien comprendre l'architecture de ce qui avait déjà été programmé. Une fois la structure comprise, nous avons pu mener notre tâche à son terme avec un retour positif de nos encadrants.

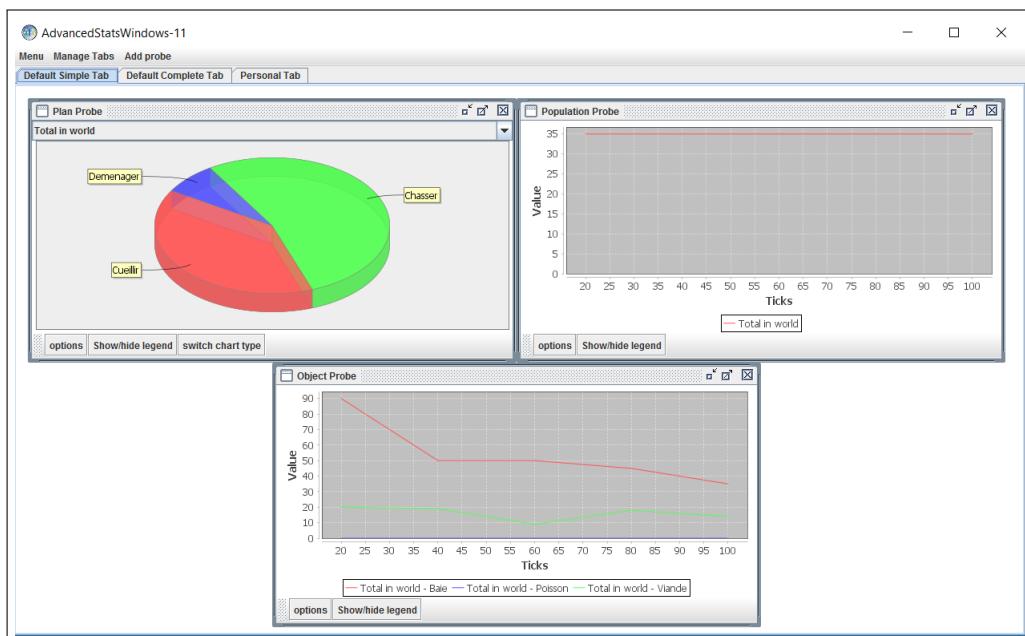


Figure 29. Onglet par défaut simple en action après le lancement de la simulation

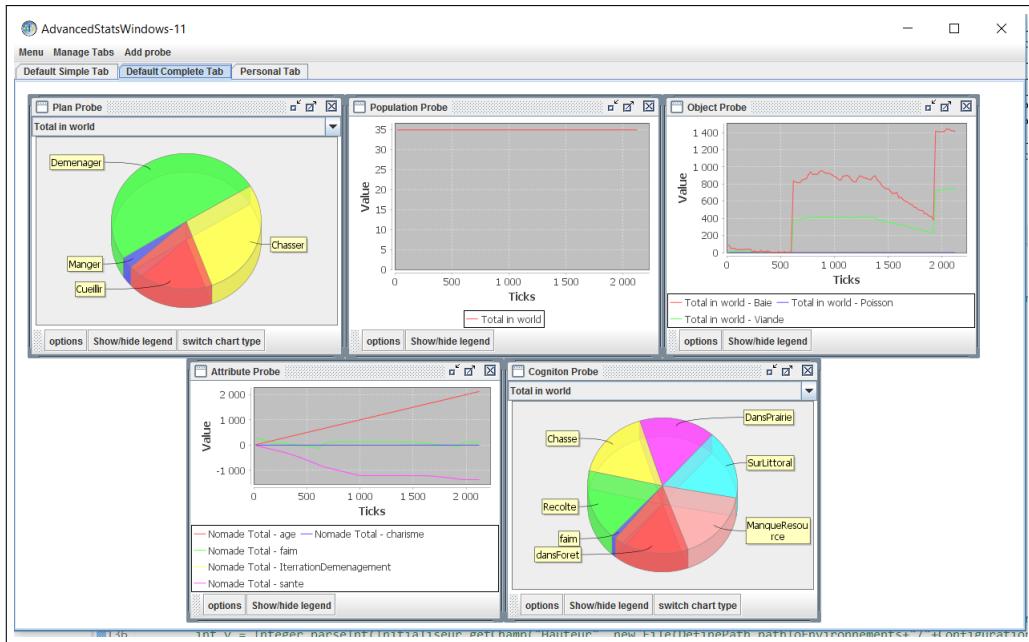


Figure 30. Onglet par défaut complet en action après le lancement de la simulation

6 La vue 3D

6.1 Introduction à la 3D

6.1.1 Intérêt de la vue 3D

Pourquoi et dans quels but nous avons voulu implémenter une vue 3D, pour la simulation de civilisation avec MetaCiv ? De notre temps, la représentation 3D est devenue classique, que ce soit dans le monde de la simulation ou vidéo-ludique. L'implémentation de la vue 3D sous MetaCiv permet ainsi à celui-ci d'entrer dans l'ère « moderne ». Comme MetaCiv est une application où l'on suit une ou plusieurs civilisation en temps réel, il fallait que le Framerate ne soit pas trop faible pour avoir une lisibilité convenable, en ce sens nous devions offrir un certain panel de configuration de la qualité graphique de la vue 3D, permettant ainsi à la vue 3D de tourner sur un large panel de configurations.

Un autre avantage évident de la vue 3D, est d'offrir une lisibilité des actions plus claire que dans la vue classique. Et offrant aussi une richesse au niveau du terrain plus intéressante.

Bien sûr la première raison pour nous d'avoir voulu implémenter une vue 3D, est d'utiliser la puissance graphique des équipements actuels pour pouvoir réaliser des simulations visuellement plus riche, permettant ainsi aux différents utilisateurs de MetaCiv de faire éventuellement des présentations de leur civilisation avec une rigueur des environnements plus importante que ce soit au niveau même du terrain pour visualiser une chaîne de montagne ou au niveau des bâtiments des civilisations.

Un des maître mot pour cette partie du TER était d'offrir la possibilité à l'utilisateur de changer les différents modèles de sa civilisation, tout en proposant des modèles par défaut très simple et fortement représentatif.

6.1.2 Présentation de la librairie LWJGL

LWJGL est une librairie Java qui permet un accès multi plate-forme aux API natives populaires utiles pour un développement d'applications graphiques tels qu'OpenGL, ou bien OpenAL pour l'audio et OpenCL pour la programmation parallèle.

LWJGL est une librairie distribuée sous une licence open source, elle est donc libre de droit.

Pour utiliser OpenGL pour une application développée sous Java, deux choix s'offrent à nous : JOGL ou LWJGL, nous avons opté pour la seconde option car LWJGL offre une gestion des événements plus intuitive et rapide que JOGL, avec un accès rapide aux différents périphériques de l'ordinateur comme la gestion de la souris ou du clavier. Ainsi que sa structure globale qui est plus proche de ce que l'on peut trouver en c++ avec glew, et étant plus familiarisé avec cette structure cela nous a conforté dans notre choix.

6.1.3 OpenGL 3.3/4.1

OpenGL est une librairie comportant un ensemble de fonctions de calcul d'image 2D ou 3D dont la première version est parue en 1992, développée par SGI (Silicon Graphics). Disponible sur de nombreuses plate-formes, ici nous développerons sous Java à l'aide de la librairie LWJGL. Pourquoi choisir OpenGL à une autre librairie graphique ? Car OpenGL est libre de droit et est compatible avec tous les systèmes d'exploitation. Le choix de la version d'OpenGL 3.3 a été fait pour tirer au mieux des innovations matérielles des cartes graphiques. Nous utilisons la version 4.1, seulement sur les équipements supportant cette version, pour pouvoir utiliser la technologie de tessellation.

6.2 MetaCiv2D vers MetaCiv3D

6.2.1 Problématique

La problématique principale du passage de la 2D vers la 3D réside dans le terrain, comment passer d'un BufferedImage vers un maillage 3D :



Figure 31. Terrain rendu dans un BufferedImage

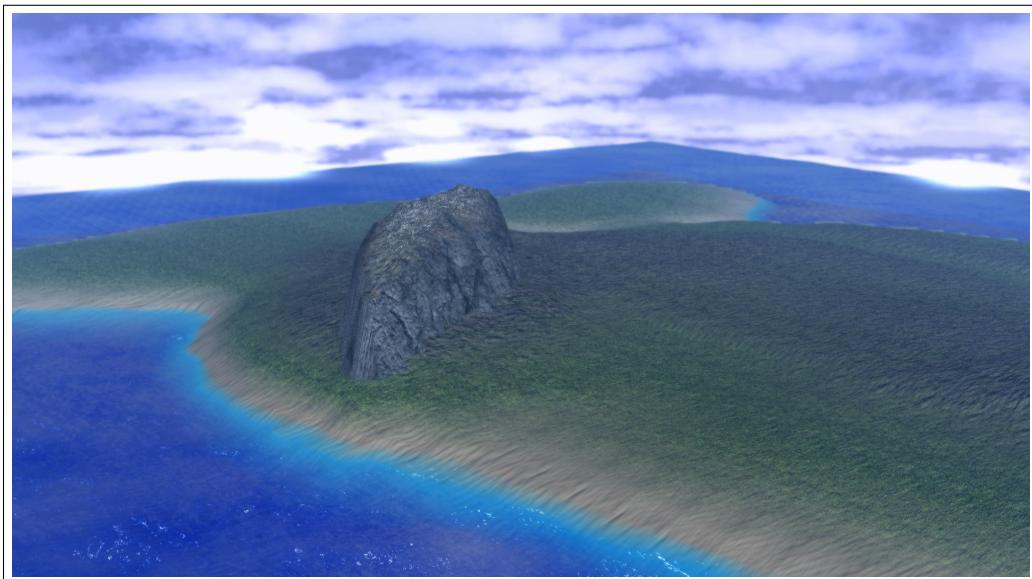


Figure 32. Terrain rendu en 3D à l'aide d'un maillage

Pour ce faire nous auront besoin d'une méthode permettant de trouver une hauteur pour chaque patch.

Une autre difficulté rencontrée est au niveau du texturing du terrain, pour trouver la bonne texture à appliquer et pour avoir aussi une transition lisse des textures sur le terrain.

Nous avons rencontré des difficultés pour trouver la position dans l'espace des tortues à chaque frame pour ainsi éviter de voir les tortues se téléporter.

6.2.2 Redirection de la sortie TKDefaultViewer

Pour récupérer les différentes valeurs nécessaires à la vue 3D, comme la position des tortues, les facilities et les debug String, ainsi que le BufferedImage que l'on transformera en texture qui sera utilisé par la suite pour la génération du terrain et le placement des textures ainsi que des routes. Nous allons rediriger la sortie du TKDefaultViewer dans la classe WorldViewer. Cette même classe où l'on créera notre thread pour la vue 3D, rendant par le fait la vue 3D et la vue 2D malheureusement fortement dépendantes, car si l'on ferme la vue 2D lors de la simulation la vue 3D ne sera plus mise à jour. Ce problème peut se corriger en parcourant la grille des patchs directement dans la boucle principale du thread dédié à la vue 3D, mais en faisant ceci nous parcourons l'ensemble des patchs une seconde fois.

Listing 30. Création du tread pour la vue 3D dans WorldViewer (package civilisation.world)

```
protected void activate() {
    super.activate();
    if(!bufferedView_activate && CivLauncher.choix3D==1){
        bufferedView = new BufferedImage(this.getWidth()*this.getCellSize(),
            this.getHeight()*this.getCellSize(), BufferedImage.TYPE_INT_ARGB);
        render = new renderMain(bufferedView, this.getGrid());
        Thread t = new Thread(render);
        g2d = bufferedView.createGraphics();
        t.start();
        bufferedView_activate = true;

        initialCellSize = cellSize;
    }
}
```

6.2.3 Génération du terrain

La partie principale du passage de la 2D vers la 3D, était de trouver un moyen de rendre le terrain sous forme de maillage. Plusieurs axes de réflexion se sont offert à nous pour générer le maillage. Le positionnement sur les axes x et z des différents vertices du maillage ne pose pas réellement de soucis car il nous suffit de récupérer la position des patchs sur les axe y et x (*NB : les axes (x,y) en 2D correspond aux axes (x,z) dans OpenGL*). Le véritable problème réside dans la hauteur du patch pour trouver la position sur l'axe y d'OpenGL. Pour ce faire, il fallait un moyen de contrôler l'élévation du terrain. Nous allons rappeler comment est définie le terrain rendue dans le BufferedImage standard de MetaCiv, le terrain est défini comme un ensemble de patchs. Un patch est défini par une couleur une position sur les axes (x,y), ainsi qu'une collection de tortues que nous aborderons plus tard.

Lors de l'utilisation de MetaCiv, l'utilisateur doit définir un ensemble de type de terrains.



Figure 33. Différents types de terrains

Un type de terrain est défini selon un ensemble de paramètres que l'utilisateur peut définir facilement à l'aide d'une fenêtre "Editer le terrain"

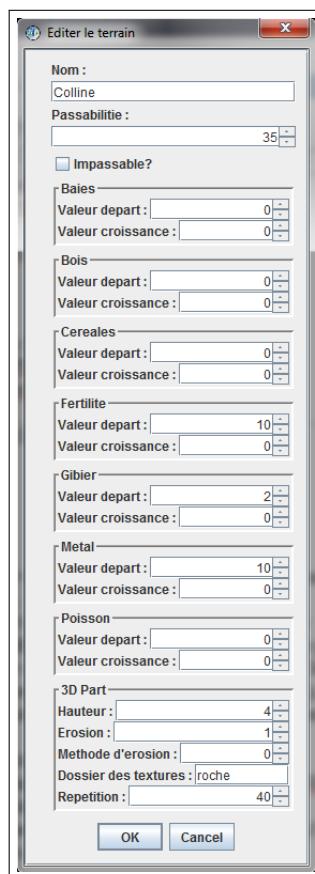


Figure 34. Fenêtre "Editer le terrain"

Comme vous pouvez le voir sur la figure (figure 34) précédente nous avons rajouté une section pour contrôler les différents paramètres nécessaires à la 3D.

Par la suite, en matchant la couleur du patch avec la couleur du type de terrain, on aura toutes les valeurs nécessaires pour construire notre maillage.

Nous allons maintenant vous montrer comment on convertit le `BufferedView` en maillage 3D. Pour les positions x et z comme dit plus haut il n'y aura pas de soucis, et pour la position y on aura besoin d'une fonction permettant de trouver la hauteur à l'aide de la couleur récupérée sur le `BufferedView` et en allant chercher parmi les types de terrain la bonne hauteur.

Pour comprendre le listing (voir listing 31) qui suit nous aurons besoin de définir les variables suivantes :

Listing 31. Constante de largeur et de hauteur

```
VERTEX_COUNT_H = image.getHeight()/World.getAccuracy();
VERTEX_COUNT_W = image.getWidth()/World.getAccuracy();
```

Où `image` est le `BufferedView` rendu dans la vue 2D de `MetaCiv`, et `World.getAccuracy()` est une variable de précision du maillage par rapport au `BufferedView`, augmentant de ce fait le nombre de polygones dans la scène. Ceci peut servir de variable d'optimisation de la vue3D. Cette variable peut être paramétrée par l'utilisateur.

Listing 32. Définition des tailles finales du maillage 3D

```
SIZE_X = World.getY()*World.getSize3D();
SIZE_Z = World.getX()*World.getSize3D();
```

`SIZE_X` et `SIZE_Z` sont les tailles finales du maillage 3D, `World.getY()` et `World.getX()` est la taille du terrain un 2D définie par l'utilisateur. `World.getSize3D()` est un multiplicateur défini par l'utilisateur pour contrôler la taille du maillage du terrain. Il peut aussi être défini par l'utilisateur.

Listing 33. Définition des tableaux de stockage des vertices, normales et coordonnées

```
float[] vertices = new float[VERTEX_COUNT_H*VERTEX_COUNT_W * 3];
float[] normals = new float[VERTEX_COUNT_H*VERTEX_COUNT_W * 3];
float[] textureCoords = new float[VERTEX_COUNT_H*VERTEX_COUNT_W *3];
```

Les tableaux précédents (listing 33) permettent de stocker la position des vertices dans l'espace, leurs normales ainsi que leurs coordonnées de textures.

Listing 34. Crédit du maillage pour le terrain (`renderEngine.terrain.Terrain.java`)

```
for(int i=0;i<VERTEX_COUNT_H;i++){
    jj = 0;
    for(int j=0;j<VERTEX_COUNT_W;j++){
        float h;
        h = getHeight(jj,ii,image,textures);

        vertices[vertexPointer*3] = (float)j/((float)VERTEX_COUNT_H - 1) * SIZE_X;
        vertices[vertexPointer*3+1] = h;
        if(!World.getHeightMap().equals(""))
            vertices[vertexPointer*3+1] +=
                getHeight(jj%image2.getHeight(),ii%image2.getWidth());
        vertices[vertexPointer*3+2] = (float)i/((float)VERTEX_COUNT_W - 1) * SIZE_Z;

        pos[vertexPointer] = new Vector3f(vertices[vertexPointer*3],
                                         vertices[vertexPointer*3+1], vertices[vertexPointer*3+2]);
        ver.add(new Vector3f(vertices[vertexPointer*3], vertices[vertexPointer*3+1],
                             vertices[vertexPointer*3+2]));

        normals[vertexPointer*3] = 0;
        normals[vertexPointer*3+1] = 0;
        normals[vertexPointer*3+2] = 0;
        tangents[vertexPointer*3] = 0;
        tangents[vertexPointer*3+1] = 0;
        tangents[vertexPointer*3+2] = 0;

        textureCoords[vertexPointer*3] = (float)j/((float)VERTEX_COUNT_W - 1);
```

```

textureCoords[vertexPointer*3+1] = (float)i/((float)VERTEX_COUNT_H - 1);
textureCoords[vertexPointer*3+2] = 0;
Uv[vertexPointer] = new Vector3f(textureCoords[vertexPointer*3],
    textureCoords[vertexPointer*3+1], textureCoords[vertexPointer*3+2]);
vertexPointer++;

jj+=World.getAccuracy();
}

ii +=World.getAccuracy();
}

```

On voit bien avec le listing précédent (listing 34 que les coordonnées x et z sont simplement trouvées à l'aide des formules suivantes :

Listing 35. Formules de calcul de x et y

```

x = j/(VERTEX_COUNT_H - 1) * SIZE_X
z = j/(VERTEX_COUNT_W - 1) * SIZE_Z

```

Pour la coordonnée y on utilise une fonction qui permet de trouver la hauteur, sachant qu'on utilise aussi une heightMap (map d'élévation) pour rajouter un certain coté naturel au terrain.

De même pour les coordonnées de textures, sauf qu'on multiplie par la taille du terrain. Pour les normales, elles sont calculées lors de la construction du tableau d'indices.

Les données du terrain d'un point de vue OpenGL, sont stockées de manière classique, c'est-à-dire que le maillage est représenté de manière indexée avec un tableau d'indices correspondant à l'emplacement dans le tableau de vertices, évitant ainsi aux vertices d'avoir à se répéter, ceci optimise ainsi l'espace mémoire, donc un triplet d'indices représente une face.

Listing 36. Crédation du tableau d'indices ainsi que les normales

```

for(int z=0; z<VERTEX_COUNT_H-1; z++){
    for(int x=0; x<VERTEX_COUNT_W-1; x++){
        int topLeft = (z*VERTEX_COUNT_W)+x;
        int topRight = topLeft + 1;
        int bottomLeft = ((z+1)*VERTEX_COUNT_W)+x;

        int bottomRight = bottomLeft + 1;
        indices[pointer++] = topLeft;
        indices[pointer++] = bottomLeft;
        indices[pointer++] = topRight;

        Vector3f a = ver.get(topLeft);
        Vector3f b = ver.get(bottomLeft);
        Vector3f c = ver.get(topRight);
        Vector3f vp1 = new Vector3f(b.x-a.x, b.y-a.y, b.z-a.z);
        Vector3f vp2 = new Vector3f(c.x-a.x, c.y-a.y, c.z-a.z);
        Vector3f v1 = new Vector3f(vp1.getX(), vp1.getY(), vp1.getZ());
        Vector3f v2 = new Vector3f(vp2.getX(), vp2.getY(), vp2.getZ());
        Vector3f normal = Helper.Vectoriel(v1, v2);

        if(normal.length()>0)
            normal.normalise();
        faces[k] = new face();
        faces[k].normal = normal;
        faces[k].pos[0] =topLeft;
        faces[k].pos[1] =bottomLeft;
        faces[k].pos[2] =topRight;
    }
}

```

```

k++;

indices[pointer++] = topRight;
indices[pointer++] = bottomLeft;
indices[pointer++] = bottomRight;

a = ver.get(topRight);
b = ver.get(bottomLeft);
c = ver.get(bottomRight);
vp1 = new Vector3f(b.x-a.x,b.y-a.y,b.z-a.z);
vp2 = new Vector3f(c.x-a.x,c.y-a.y,c.z-a.z);
v1 = new Vector3f(vp1.getX(),vp1.getY(),vp1.getZ());
v2 = new Vector3f(vp2.getX(),vp2.getY(),vp2.getZ());
normal = Helper.Vectoriel(v1,v2);

if(normal.length()>0)
    normal.normalise();
faces[k] = new face();
faces[k].normal = normal;
faces[k].pos[0] =topRight;
faces[k].pos[1] =bottomLeft;
faces[k].pos[2] =bottomRight;
normals[topLeft*3] += faces[k-1].normal.getX();
normals[topLeft*3+1] += faces[k-1].normal.getY();
normals[topLeft*3+2] += faces[k-1].normal.getZ();
diviseur[topLeft] ++;

normals[bottomLeft*3] += faces[k-1].normal.getX();
normals[bottomLeft*3+1] += faces[k-1].normal.getY();
normals[bottomLeft*3+2] += faces[k-1].normal.getZ();
diviseur[bottomLeft] ++;

normals[topRight*3] += faces[k-1].normal.getX();
normals[topRight*3+1] += faces[k-1].normal.getY();
normals[topRight*3+2] += faces[k-1].normal.getZ();
diviseur[topRight] ++;

normals[topRight*3] += faces[k].normal.getX();
normals[topRight*3+1] += faces[k].normal.getY();
normals[topRight*3+2] += faces[k].normal.getZ();
diviseur[topRight] ++;

normals[bottomLeft*3] += faces[k].normal.getX();
normals[bottomLeft*3+1] += faces[k].normal.getY();
normals[bottomLeft*3+2] += faces[k].normal.getZ();
diviseur[bottomLeft] ++;

normals[bottomRight*3] += faces[k].normal.getX();
normals[bottomRight*3+1] += faces[k].normal.getY();
normals[bottomRight*3+2] += faces[k].normal.getZ();
diviseur[bottomRight] ++;

k++;
}
}

```

Pour générer le tableau d'indices, il nous faut la position dans le tableau de vertices des quatre coins du polygones courant, on peut ainsi les rentrer dans le tableau d'indices, en prenant garde à l'ordre pour éviter les problèmes de back facing (face retournée).

Pour les normales, on va d'abord calculer les normales aux faces de chaque triangle, et ensuite pour chaque vertice du triangle on ajoute la normale pour pouvoir faire la moyenne des normales des faces avoisinantes. Par la suite on divisera toutes les normales par leur diviseur respectif.

Nous allons maintenant montrer comment est trouvé la hauteur pour un vertice donné. En commençant par traiter le cas le plus simple trouver la hauteur dans la `heightMap`. Pour ce faire, nous allons simplement récupérer la valeur `rgb` de la `heightMap`, en prenant garde de ne pas être hors des bornes de celle-ci. On la supposera carrée. Ensuite, pour éviter que la `heightMap` « creuse » uniquement le terrain on va par la suite la bornée à $[-256^3/2, 256^3/2]$ pour avoir aussi une élévation du terrain. Car sinon nous allons avoir des valeurs comprises entre -255 et 0. Nous divisons par la suite pour normaliser et ramener les valeurs dans l'intervalle $[-1, 1]$

Listing 37. Fonction pour trouver la hauteur dans une `heightMap`

```
public static float getHeight(int x, int z){
    if(x<0 || x >= image.getHeight() || z<0 || z >= image.getHeight()){
        return 0;
    }
    float height = image.getRGB(x, z);
    height += MAX_PIXEL_COLOUR /2f;
    height /= MAX_PIXEL_COLOUR /2f;
    height *= World.getIntensityHeight();

    return height;
}
```

Nous allons maintenant aborder la partie plus compliquée, trouver la hauteur à l'aide du `BufferedView`. Nous commençons par récupérer la valeur du pixel en `rgb`, il nous faut ensuite convertir cette valeur qui est en `int` en trois valeurs (`red,green,blue`). Pour réaliser cette opération nous allons effectuer un décalage bit à bit de 16 pour le rouge de 8 pour le vert et aucun pour le bleu. Ensuite, on effectue un `&` binaire de 255 permettant ainsi de récupérer la valeur des différents champs. Il nous reste ensuite qu'à trouver à quel type de terrain correspond ses champs. On a donc le type de terrain et la hauteur qui lui est associée.

Listing 38. Opérations binaires sur les pixels

```
int rgb = image.getRGB(x, z);
float red = (rgb >> 16) & 0x000000FF;
float green = (rgb >> 8 ) & 0x000000FF;
float blue = (rgb) & 0x000000FF;
float height = 0.0f;
```

Pour éviter un aspect trop abrupte des transitions entre les différents types de terrains comme sur l'exemple qui suit (Figure 35) :

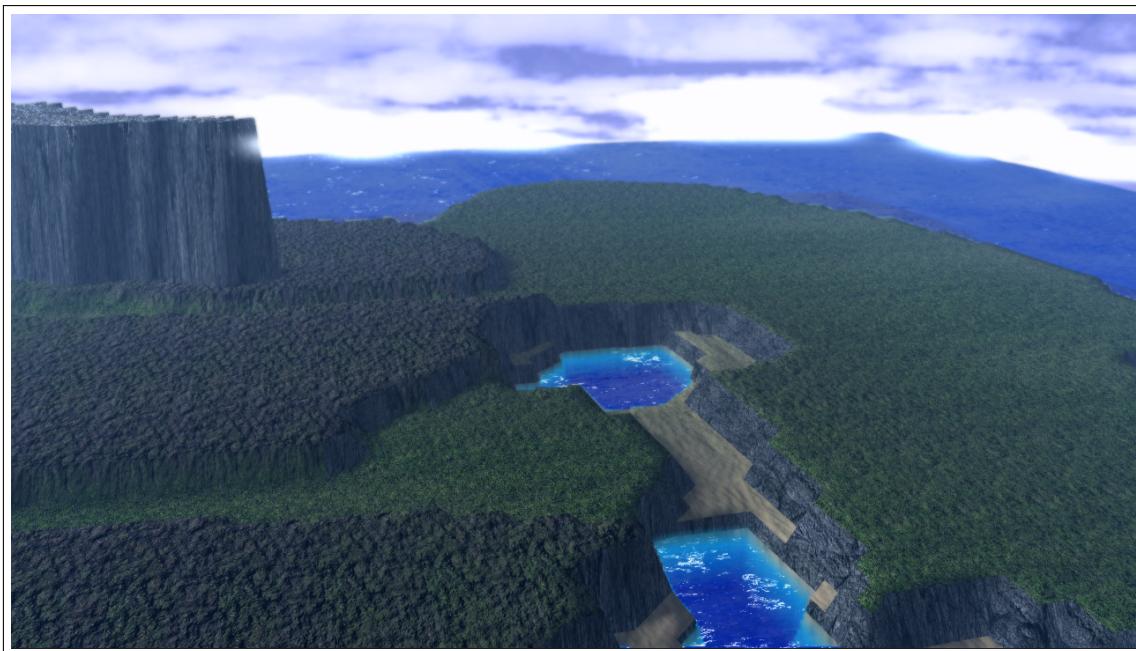


Figure 35. Exemple sans érosion

Il nous fallait un moyen de contrôler ce que l'on appelle, nous, l'érosion, c'est-à-dire un flou dans le maillage. Pour obtenir un résultat comme suit :



Figure 36. Même exemple avec érosion

Pour obtenir ce résultat, nous avons laissé à l'utilisateur le choix de la force d'érosion ainsi que la méthode à utiliser.

La première méthode, est une méthode qui consiste simplement à parcourir, selon une valeur d'érosion, et faire une moyenne des 8 voisins et ceux à une distance de deux en haut, en bas, à droite et à gauche pour donner un certain côté naturelle (voir le tableau ci dessous 1).

$(X - 2, Z)Erosion = 4$	$(X - 1, Z - 1)Erosion = 3$	$(X, Z - 1)Erosion = 2$	$(X + 1, Z - 1)Erosion = 3$	$(X + 2, Z - 2)Erosion = 3$
$(X - 2, Z)Erosion = 4$	$(X - 1, Z)Erosion = 2$	$(X, Z)Erosion = 1$	$(X + 1, Z)Erosion = 2$	$(X + 2, Z)Erosion = 4$
$(X - 1, Z + 1)Erosion = 3$	$(X, Z + 1)Erosion = 2$	$(X + 1, Z + 1)Erosion = 3$	$(X + 2, Z + 2)Erosion = 3$	$(X, Z + 2)Erosion = 4$

Table 1. Représentation du voisinage simple. La cellule bleue, représente le point courant.

La seconde méthode, quant à elle est plus précise mais plus gourmande en ressources pour être calculée. Il s'agit de la méthode classique de flou prenant tous les voisins (voir le tableau ci-dessous 2).

$(X - 2, Z - 2)Erosion = 3$	$(X - 1, Z - 2)Erosion = 3$	$(X, Z - 2)Erosion = 3$	$(X + 1, Z - 1)Erosion = 3$	$(X + 2, Z - 2)Erosion = 3$
$(X - 2, Z - 1)Erosion = 3$	$(X - 1, Z - 1)Erosion = 2$	$(X, Z - 1)Erosion = 2$	$(X + 1, Z - 1)Erosion = 2$	$(X + 2, Z - 2)Erosion = 3$
$(X - 2, Z)Erosion = 3$	$(X - 1, Z)Erosion = 2$	$(X, Z)Erosion = 1$	$(X + 1, Z)Erosion = 2$	$(X + 2, Z)Erosion = 3$
$(X - 2, Z + 1)Erosion = 3$	$(X - 1, Z + 1)Erosion = 2$	$(X, Z + 1)Erosion = 2$	$(X + 1, Z + 1)Erosion = 2$	$(X + 2, Z + 1)Erosion = 3$
$(X - 2, Z + 2)Erosion = 3$	$(X - 1, Z + 2)Erosion = 3$	$(X, Z + 2)Erosion = 3$	$(X + 1, Z + 2)Erosion = 3$	$(X + 2, Z + 2)Erosion = 3$

Table 2. Représentation du voisinage avancé. La cellule bleue, représente le point courant.

Listing 39. Fonction d'érosion

```
public static float getHeight(int x, int z, BufferedImage image,
    ArrayList<TerrainTexture> textures){

    int N=getErosion(x, z, image, textures);

    int dividande = 0;
    int merge = 3;
    float height = 0;
    float heightL, heightR, heightD, heightU;
    int algo = getBlur(x, z, image, textures);
    if(algo == 0){
        for(int i=0; i<N; i++){

            if(i % 2 == 0){
                heightL = getHeight2(Math.max(x-i*merge, 0), z, image, textures);
                heightR = getHeight2(Math.min(x+i*merge, image.getWidth()), z,
                    image, textures);
                heightD = getHeight2(x, Math.max(z-i*merge, 0), image, textures);
                heightU = getHeight2(x, Math.min(z+i*merge, image.getHeight()),
                    image, textures);
            } else {
                heightL = getHeight2(Math.max(x-(i-1)*merge, 0), Math.max(z-(i-1)*merge,
                    0), image, textures);
                heightR = getHeight2(Math.min(x+(i-1)*merge, image.getWidth()),
                    Math.min(z+(i-1)*merge, image.getHeight()), image, textures);
                heightD = getHeight2(Math.min(x+(i-1)*merge, image.getWidth()),
                    Math.max(z-(i-1)*merge, 0), image, textures);
                heightU = getHeight2(Math.max(x-(i-1)*merge, 0), Math.min(z+(i-1)*merge,
                    image.getHeight()), image, textures);
            }
            dividande += 4;
            height += heightL + heightR + heightD + heightU;
        }
        height/= dividande;
    }
    else{
        int i = 0;
        while(i < N){
```

```

int l = -1*(i+1);
int m = 0;
int h = -1*(i+1);
while(h < -1*(i+1)*-1){
    height += getHeight2(Math.min(Math.max(x+l*merge, 0), image.getWidth()),
        Math.min(Math.max(z+h*merge, 0), image.getHeight()), image, textures);
    l++;
    m++;
    if(m>=(2*(i+1)+1) ){
        h++;
        m=0;
        l=-1*(i+1);
    }
    dividande++;
}
i++;
}
height /= dividande;
}
return height;
}

```

Pour la génération des textures, nous avons créé une méthode permettant de combiner plusieurs textures dans une texture Atlas (Texture comportent plusieurs textures), car effectivement pour chaque type de terrain l'utilisateur peut avoir la texture de diffuse(couleur) ainsi que la normale map. En utilisant la fonction getRGB de la classe BufferedImage pour créer un buffer contenant tout les pixels pour chaque canal. Ensuite on crée un ByteBuffer pour la texture Atlas, on récupère ensuite, pour chaque pixel la valeur rouge, vert, bleu et de transparence en effectuant une opération binaire. On génère la texture à l'aide de la fonction OpenGL : glGenTexture(). On prend ensuite bien soin de paramétriser le wrapping des texture en GL_REPEAT. Et pour les filtres nous utiliserons GL_LINEAR pour un rendu plus propre.

Listing 40. Définition des paramètres de la texture

```

//Setup wrap mode
GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_WRAP_S, GL11.GL_REPEAT);
GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_WRAP_T, GL11.GL_REPEAT);

//Setup texture scaling filtering
GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_MIN_FILTER,
    GL11.GL_LINEAR);
GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_MAG_FILTER,
    GL11.GL_LINEAR);

```

On stocke finalement les texel de la texture Atlas à l'aide de la fonction OpenGL : glTexImage2D().

Listing 41. Appel à la fonction glTexImage2D pour stocker les texel

```

GL11.glTexImage2D(GL11.GL_TEXTURE_2D, 0, GL11.GL_RGBA8,
    imageDiff.getWidth()*2+imageNrm.getWidth()*2, imageDiff.getHeight(), 0,
    GL11.GL_RGBA, GL11.GL_UNSIGNED_BYTE, buffer);

```

Voilà qui conclut pour la génération du maillage du terrain.

6.2.4 Les tortues

Un des intérêt de la vue 3D, est de représenter les tortues en 3D. Nous avons donc créé une classe Java Turtle3D caractérisée par sa position dans l'espace, un objet 3D, une couleur d'action, un entier identifiant, ainsi qu'une couleur identifiant. Cela nous permettra de sélectionner les tortues à l'aide de la souris. Et pour finir, la classe Turtle3D possède une variable d'interpolation pour faire une interpolation entre deux positions, pour éviter de voir les tortues se téléporter lors d'un déplacement.

Pour assurer la possibilité de choisir les propres modèles de l'utilisateur, nous avons créé une méthode static de mise en place des modèles 3D.

Listing 42. Fonction setUp pour les modèles 3D des tortues (classe renderEngine.entities.Turtles3D.java)

```
public static void setUp(Loader loader){
    for (int i = 0; i < Configuration.civilisations.size(); i++){
        File f = new File(Configuration.pathToRessources + "/Skin/Civilisations/" +
            Configuration.civilisations.get(i).getNom() + "/" +
            Configuration.civilisations.get(i).getNom() + ".OBJ");

        if(f.isFile()){
            objects3ds.add(new Object3D(f.getAbsolutePath(), f.getParent(), loader, true,
                true, new Vector3f(0, 0, 0), 0, 0, 0, 1.0f));
            BoundingBox box = objects3ds.get(objects3ds.size() - 1).getModels().getBox();
            float distX = box.getMax().x - box.getMin().x;
            float distY = box.getMax().y - box.getMin().y;
            float distZ = box.getMax().z - box.getMin().z;
            float distMax = Math.max(Math.max(distX, distY), distZ);
            if(distMax>World.getSize3D()){
                objects3ds.get(objects3ds.size() - 1).setScale(World.getSize3D() /
                    distMax);
            }
        }else{
            objects3ds.add(new Object3D("turtle", "", loader, true, new Vector3f(150, 8,
                150), 0, 0, 0, 0.03f));
        }
    }
}
```

On voit dans le listing 42 on parcourt l'ensemble des civilisations, puis on regarde dans le dossier Skin (dossier où l'utilisateur stocke les différents modèles 3D et textures à utiliser pour sa simulation) et Civilisations (dossier où l'utilisateur stocke ces tortues). Si l'on trouve un fichier OBJ alors on le charge, sinon on utilise le modèle par défaut du logiciel. On remarque aussi que l'on force le modèle à tenir dans le patch, en construisant une boîte englobante de l'objet 3D et on test s'il tient dans un patch, sinon on effectue une mise à l'échelle homothétique.

Pour trouver la position de la tortue dans l'espace 3D, nous récupérons sa position dans l'espace 2D, mais comme nous redirigeons la sortie de la vue 2D, il nous faut prendre en compte la taille des cellules de la vue 2D, car selon le zoom il change. Il faut alors diviser la position par la taille des cellules actuelles, elle-même divisée par la taille initiale des cellules.

Listing 43. Position transformée pour être dans l'espace du tableau de vertices

```
int cX = (int) ((x/(cellSize/ (float)WorldViewer.initialCellSize)) /
    World.getAccuracy());
int cY = (int) ((y/(cellSize/ (float)WorldViewer.initialCellSize)) /
    World.getAccuracy());
```

On divise aussi par la précision du terrain, on obtient ainsi deux variables que l'on peut utiliser

pour trouver la position dans le tableau de vertices, on a ainsi la position dans l'espace 3D de la tortue. Maintenant pour rendre les déplacements des tortues plus propre nous avons mis en place un système d'interpolation entre les positions. Mais comme MetaCiv repose sur la plateforme MadKit / TurtleKit, il possède une vitesse de simulation. Il nous fallait donc récupérer cette vitesse définie dans le scheduler de MadKit, une fois récupérée on peut augmenter l'interpolation en fonction de cette vitesse.

Listing 44. Augmentation de la variable d'interpolation

```
turtles.get(id).increaseInterpolation( float) (1.0 / (double)
    CivLauncher.sch.getDelay() / (float) delta));
```

Dans le listing 44, on remarque la présence de la variable delta. Il s'agit d'un delta calculé en fonction du framerate de l'application évitant ainsi des problèmes de téléportation sur des machines avec un framerate différent.

Passons maintenant à la sélection des tortues à l'aide de la souris. En effet, nous trouvions intéressant voir indispensable la possibilité de sélectionner les tortues pour que la caméra suive la tortue ainsi sélectionnée, et aussi pouvoir voir les informations relatives à la tortue/agent sélectionné. Pour ce faire, nous arions pu faire un lanceur de rayon puis calculer l'intersection avec la boite englobante, mais cette technique est un peu gourmande si l'on veut de la précision. Elle peut être intéressante pour des applications où l'on veut récupérer la position de l'intersection mais dans notre cas cela ne nous intéresse pas. Nous nous sommes plutôt penchés sur la méthode de détection avec les couleurs, qui consiste à faire une passe supplémentaire pour les colorID des tortues dans un autre FrameBufferObject, et par la suite on peut utiliser la fonction readPixel d'OpenGL qui renvoie un ByteBuffer correspondant à la couleur du pixel à la position (x,y) spécifié.

Listing 45. Fonction qui récupère la couleur du pixel dans le framebuffer

```
public ByteBuffer ReadPixel(int x,int y)
{
    GL30 glBindFramebuffer(GL30.GL_READ_FRAMEBUFFER, this.frameBuffer);
    GL11.glReadBuffer(GL30.GL_COLOR_ATTACHMENT0);

    ByteBuffer pixels = BufferUtils.createByteBuffer(3);
    GL11.glReadPixels(x, y, 1, 1, GL11.GL_RGB, GL11.GL_UNSIGNED_BYTE, pixels);

    return pixels;
}
```

Il nous suffit par la suite de chercher à quelle tortue correspond cette couleur pour savoir si une tortue est sélectionnée et laquelle.

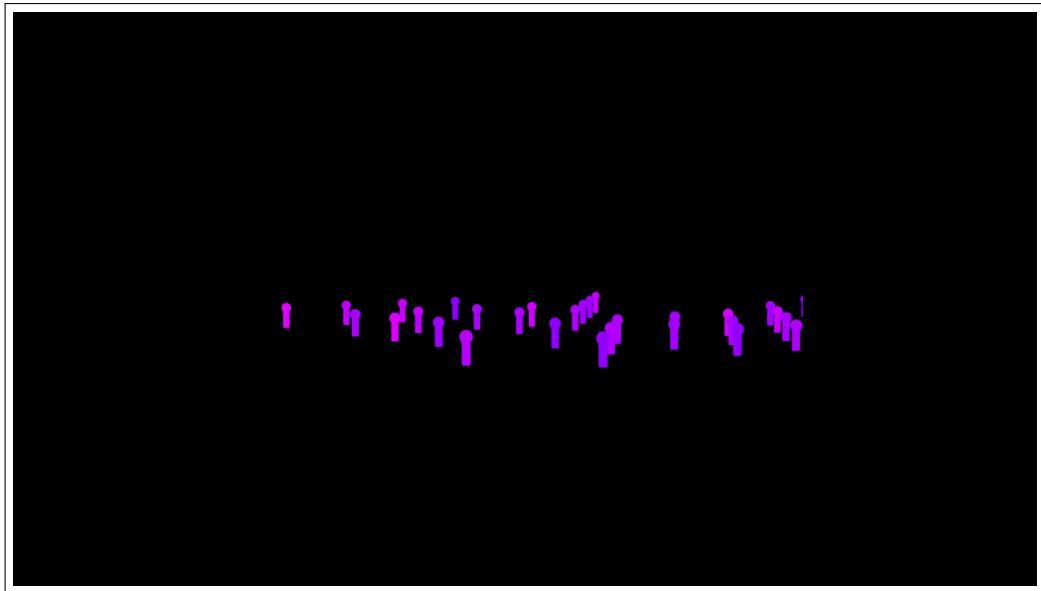


Figure 37. Représentation des Color ID

6.2.5 Les facilities

Les facilities (aménagements) fonctionnent de la même manière que pour les tortues, la seule véritable particularité entre les tortues et les facilities, réside dans les bâtiments principaux. En effet, nous nous sommes rendu compte que les premiers bâtiments déposés par les civilisations n'étaient pas forcément gérés de la même façon, car lorsque l'on redirigeait simplement la sortie de la vue 2D, le bâtiment n'était pas dessiné dans la vue 2D donc pour corriger ce problème nous avons opté pour une méthode simple. Nous avons simplement créé une fonction que l'on appelle une fois, et qui pour toutes les civilisations présentes dans la simulation, pose un bâtiment de type « setlement » à la position de départ de la civilisation.

Listing 46. Fonction de création des bâtiments principaux des civilisations

```

synchronized public void initMainFacilitys(){
    init = false;
    for (int i = 0; i < Configuration.civilisations.size(); i++){
        for(int j=0;j<grid.length;j++){
            if(grid[j].x == Configuration.civilisations.get(i).getStartX() && grid[j].y
               == Configuration.civilisations.get(i).getStartY()){

                int cX = (int) ((grid[j].x*5) / World.getAccuracy());
                int cY = (int) ((grid[j].y*5) / World.getAccuracy());

                Vector3f position = Terrain.getHeightByTab(cX, cY);
                facilitys.add(new Facility3D(new Vector3f(position.x, position.y,
                    position.z), -1, true));

                break;
            }
        }
    }
}

```

6.2.6 Les debugs Messages

Encore un sujet que l'on avait à traiter, était de rendre les messages servant de débuggeur pour les civilisations.



Figure 38. Debugs Strings en 2D



Figure 39. Debugs Strings en 3D

Le problème principal pour rendre ces debugs strings en 3D, est qu'il fallait réaliser un chargeur de police et rendre ces Strings en 3D, à savoir que réaliser une bonne structure pour charger les fonts (polices) est loin d'être un problème aisés en OpenGL, et comme le moment où l'on a commencé cette implémentation le temps nous faisait défaut, on a alors dû trouver une structure permettant de charger les fichiers .fnt et une texture Atlas représentant la police d'écriture sur internet. À savoir qu'une texture Atlas est une texture dont on se sert comme une texture comportant plusieurs textures. Le fichier .fnt quant à lui est un fichier comportant toutes les informations nécessaires pour la texture Atlas comme la position du caractère, la taille de ce dernier, le code ASCII, le décalage qui correspond à l'écart avec le caractère précédent, ainsi que l'avancement sur l'axe x pour le caractère suivant. Nous avons donc trouvé un ensemble de classe Java permettant d'effectuer ces opérations et nous remercions grandement la personne répondant au nom de Karl. Pour notre part, nous avons amélioré sa structure pour pouvoir rendre les Strings en Billboard, un Billboard en 3D est une primitive dont les rotations de camera sont bloquées, donnant ainsi l'impression que la primitive nous fait toujours face. Pour ce faire, il suffit simplement de multiplier la matrice de model-view matrices par un vecteur (0,0,0,1) et seulement ensuite additionner la position qui est en position x et y.

6.3 Structure du moteur graphique

6.3.1 Gestion des shaders

Notre moteur graphique est un moteur utilisant les technologies récentes pour des équipements actuels. C'est à dire qu'il utilise une pipeline graphique mettant en jeu ce que l'on appelle les shaders (nuanceurs en français). Les shaders, initialement créé par Pixar en 1988 dans leur standard RIS(RenderMan Interface Specification), sont des programmes compilés et utilisés par les processus graphique allégeant de ce fait la charge du processeur, servant aux traitements des primitives. Il existe plusieurs langages de shaders. De notre côté, nous allons utiliser GLSL 3.3/4.1, qui est un langage de shaders standardisé par OpenGL. Il existe plusieurs niveaux de shaders intervenant dans la pipeline graphique, du positionnement dans l'espace jusqu'au rendu dans le FrameBuffer.

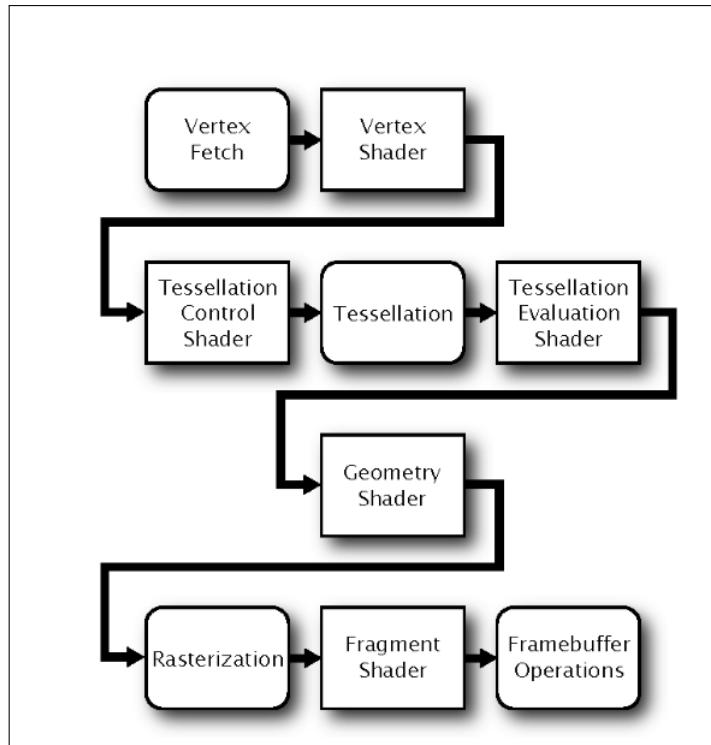


Figure 40. Présentation de la pipeline graphique simplifiée

Sur la figure 40, les boites aux bords arondis sont les fonctions fixes d'OpenGL. Il n'est pas à la charge de l'utilisateur de les définir, et parmi l'ensemble des shaders visibles seul le Vertex Shader est réellement nécessaire pour la bonne marche d'une application. Dans notre cas, les shaders supplémentaires que nous allons utiliser hors ceux obligatoires sont le Tessellation Control Shader et Tessellation Evaluation Shader pour le rendu de la mer, et le fragment Shader pour le reste.

Pour gérer efficacement nos shaders, nous avons opté pour deux structures de classe abstraite. Une pour les shaders faisant appel à la tessellation et une autre pour celle ne l'utilisant pas. Ses classes auront pour office de charger les shaders, de générer le programme du shader, d'attacher les différents niveaux de shaders à ce programme, et de créer des fonctions permettant de bind un attribut facilement, ainsi que des fonctions pour envoyer des variables uniforms. Un attribut est une donnée passée en entrée du vertex Shader. Une variables uniforms est un autre moyen de passer des données entre l'application et la carte graphique, les données uniforms peuvent être envoyées à n'importe quels niveaux de shaders. On fera en sorte que, grâce à ces classes, la récupération des locations des variables uniforms et le binding des attributs se fasse systématiquement.

Ensuite, pour chaque rendu pour lequel on voudrait utiliser un ensemble de shaders particuliers, il suffira de créer une classe étendue de ces classes abstraites. On peut ainsi pour chaque shader charger les variables uniforms que l'on veut, ainsi que facilement binder les attributs.

6.3.2 Modèles 3D

Les modèles 3D sont définis par une classe Object3D qui possède un Model tiré de la classe Model que l'on expliquera plus tard. La classe Object3D possède toutes les informations nécessaires à la création de la matrice de transformations, comme la position de rotation et l'échelle. La création du model se fait à l'aide de la classe ObjLoader que l'on abordera plus tard.

La classe Model possède un maillage Mesh, un matériel Material ainsi qu'une boîte englobante. La classe Mesh contient simplement les informations relatives au maillage comme l'identifiant vers le Vertex Array, ainsi que le nombre de vertex contenu dans le maillage. La classe Material contient quant à elle, les identifiant des différentes textures, comme nos Object3D et utilise un système de

rendu dit Physically Based Rendering. Beaucoup de types de textures sont utilisés, ces types seront expliqués plus précisément plus tard quand on traitera le susdit PBR. La classe Material contient aussi les informations pour le spéculaire et la diffuse colore.

Nous utilisons donc la classe ObjLoader pour charger des maillages 3D stockés dans des fichiers .obj (Wavefront file) qui est un format classique pour stocker des fichiers 3D. C'est un type de fichier non binaire donc plus facile à lire. Ce format ne permet pas de charger des modèles compliqués comme des squelettes d'animations. C'est un modèle de maillage indexé comme pour le terrain, il faut donc faire attention à l'indexage, sachant que les fichiers .obj peuvent poser problème car si l'on veut réaliser un indexage facile, un problème survient car les fichiers .obj possède une méthode pour définir les faces comme suit :

Listing 47. Exemple de ligne qui définit une face

```
f vertex1/textureCoord1/normal1 vertex2/textureCoord2/normal2
    vertex2/textureCoord2/normal2
```

Ce qui peu provoquait une erreur dans les coordonnées de textures, car si pour un vertex plusieurs coordonnées de textures sont fausses on aura un résultat non voulu (voir la figure 41).

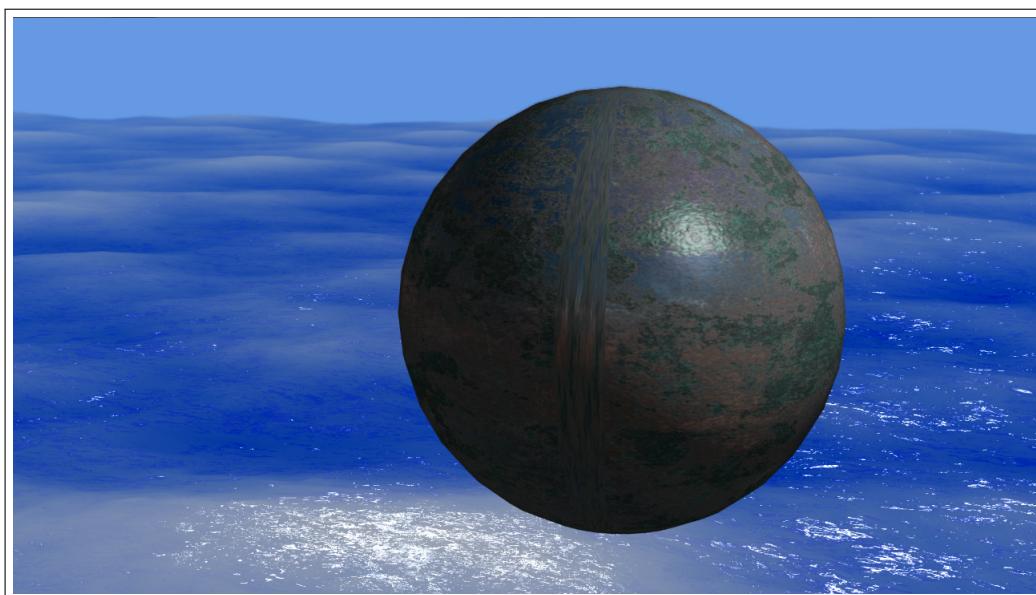


Figure 41. Sphère avec une erreur de coordonnées de textures

Donc pour corriger ce problème, nous pouvons effectuer un pré-traitement sur le maillage en créant une découpe pour éviter ce problème. Mais comme dans MetaCiv l'utilisateur peut utiliser ses propres modèles, il faut donc lui forcer à effectuer ce genre de traitement. Une autre solution est de traiter le maillage lors du chargement de sorte à régler ce problème, en effectuant un traitement si l'on rencontre un vertice que l'on à déjà traité. Pour ce faire, on crée une classe Vertex, possédant à la fois la position du vertice mais aussi l'indice de la coordonnée de texture ainsi que celle de la normale. Cette classe possède aussi un tableau de tangentes pour le calcul des tangentes que l'on expliquera plus tard.

Ainsi, il nous suffit de regarder si le vertex courant a déjà une coordonnée de texture ou une normale. Si, en effet, une des deux est déjà définie, alors on duplique le vertex avec les nouveaux indices de normales et de coordonnées de texture.

Dans notre système de rendu en PBR, nous aurons besoin de passer dans l'espace tangentiel. Donc, comme les fichiers .obj ne permettant pas de stocker les tangentes, nous avons besoin de les calculer lors du chargement du modèle, pour ce faire on va utiliser une méthode simple.

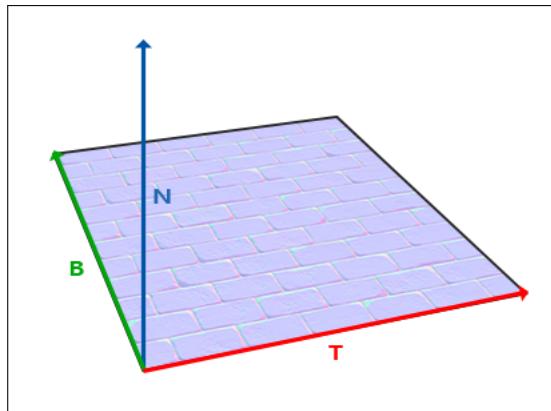


Figure 42. Axes Normale,Tangente, Bi-tangente

Comme nous pouvons voir sur la figure 42, les tangentes sont perpendiculaires aux normales et il faut que toutes les tangentes soient cohérentes entre elles (voir figure 43). Pour la bi-tangente, il s'agit juste d'un produit vectoriel entre la normale et la tangente que l'on effectuera directement dans le vertex Shader.

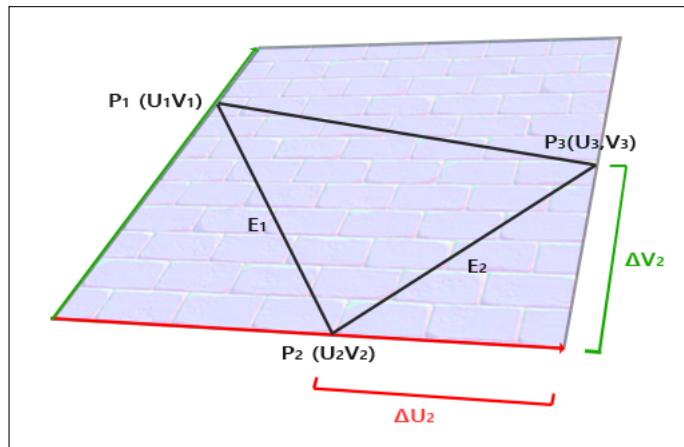


Figure 43. Schéma de calcul de la tangente

Cette méthode permet donc au tangentes de s'aligner sur les coordonnées de textures permettant ainsi au tangentes d'être toutes cohérentes entre elles (voir figure 43).

Voici la formule :

Listing 48. Exemple de ligne qui définit une face

```
//Pour un triangle donne avec trois vertices P0, P1, P2 avec les coordonnees de
texture correspondante T0, T1, T2:
```

```
deltaPos1 = P0 - P1
deltaPos2 = P2 - P0
deltaUv1 = T1 - T0
deltaUv2 = T2 - T0
```

```
f = 1 / (deltaUv1.x * deltaUv2.y - deltaUv1.y * deltaUv2.x)
```

```
tangent = (deltaPos1 * deltaUv2.y - deltaPos2 * deltaUv2.x) * f
```

Ensuite, pour chaque vertex on ajoute la tangente que l'on trouve à sa position, puis on finit par une moyenne.

Nous avons par la suite mis en place un système permettant de charger les différentes textures liées au modèle. Dans un premier temps, on va lire le fichier `material` (`.mtl`) associé à l'`object3D`, mais comme les fichiers `.mtl` ne peuvent pas contenir toutes les informations relatives aux différentes textures nécessaires au modèle de rendu en PBR. Pour y remédier, nous regardons si l'utilisateur a défini un dossier portant le nom de l'`object3D`, et si ce dossier contient des fichiers `.PNG`, avec pour nom une indication du type de texture auquel elle fait référence. Nous avons essayé d'être assez souple au niveau des noms pouvant être employés.

Listing 49. Méthode pour récupérer les noms des différentes textures

```
File f ;
if(absolute){
    f = new File(path + "/" + fileName);
} else {
    f = new File("Assets/texture/" + fileName);
}
if(f.isDirectory()){

    for (final File fileEntry : f.listFiles()) {

        if(fileEntry.getName().equals("Diffuse.png") ||
           fileEntry.getName().equals("diffuse.png") ||
           fileEntry.getName().equals("Albedo.png") ||
           fileEntry.getName().equals("albedo.png") ||
           fileEntry.getName().equals("Base_Color.png") ||
           fileEntry.getName().equals("base_color.png") ||
           fileEntry.getName().equals("Base_color.png") ||
           fileEntry.getName().equals("base_Color.png") ||
           fileEntry.getName().equals("BaseColor.png") ||
           fileEntry.getName().equals("Basecolor.png") ||
           fileEntry.getName().equals("baseColor.png") ||
           fileEntry.getName().equals("basecolor.png")){
            textureFile = fileName + "/" + fileEntry.getName();
        }
        else if(fileEntry.getName().equals("normal.png") ||
                fileEntry.getName().equals("Normal.png") ||
                fileEntry.getName().equals("Normal_OpenGL.png")){
            normalFile = fileName + "/" + fileEntry.getName();
        }
        else if(fileEntry.getName().equals("Height.png") ||
                fileEntry.getName().equals("height.png") ||
                fileEntry.getName().equals("DisplacementMap.png") ||
                fileEntry.getName().equals("displacmentMap.png") ||
                fileEntry.getName().equals("displacentmap.png") ||
                fileEntry.getName().equals("dispMap.png") ||
                fileEntry.getName().equals("dispmap.png")){
            dispFile = fileName + "/" + fileEntry.getName();
        }
        else if(fileEntry.getName().equals("Roughness.png") ||
                fileEntry.getName().equals("roughness.png")){
            reflFile = fileName + "/" + fileEntry.getName();
        }
        else if(fileEntry.getName().equals("Metallic.png") ||
```

```

        fileEntry.getName().equals("metallic.png") ||
        fileEntry.getName().equals("Metal.png") ||
        fileEntry.getName().equals("metal.png")){
    metalFile = fileName + "/" + fileEntry.getName();
}
}
}

```

C'est maintenant qu'intervient une classe Loader qui permet de charger les textures et de créer le Vertex Array.

Pour le chargement des textures, nous utilisons une librairie Slick-Util qui est une petite librairie permettant de charger des images, sons et polices d'écriture pour être ensuite utilisés par LWJGL. Nous avons aussi choisi d'utiliser le MipMapping ainsi que d'utiliser un filtre anisotropie pour une meilleure qualité des textures.

Pour la mise en place du Vertex Array Object, on commence par générer le Vertex Array, à l'aide de la fonction `glGenVertexArrays()` puis on le bind. On bind ensuite les indices à l'aide de `glBufferData` avec pour cible `GL_ELEMENT_ARRAY_BUFFER`. On met nos indices en mode `GL_STATIC_DRAW`, ce qui signifie qu'une fois bind on ne peut plus modifier le buffer. Il nous reste ensuite plus qu'à stocker les différents attributs. Pour ce faire, on utilise la même fonction que pour les indices, sauf que pour la cible on a `GL_ARRAY_BUFFER`. Ensuite, on va spécifier le numéro d'attribut ainsi que la taille à réserver pour chaque (exemple : pour une position on à une taille de 3 (x, y, z)). Pour ce faire, on utilise la fonction `glVertexAttribPointer`.

n°	Attributs
0	vertex
1	textureCoord
2	normal
3	tangente

Table 3. Représentation du Vertex Array Object

6.3.3 Gestion des rendus

Une classe importante dans notre moteur de rendu est la classe MasterRenderer. C'est elle qui s'occupe de rendre les entités et le terrain. L'intérêt principal de cette classe et de créer des batchs (Lots) d'entités à rendre, optimisant ainsi grandement le rendu des entités. Car en construisant une hashMap dont la clef est le model et une liste object3D associée à chaque modèle 3D. Pour toutes les fois où l'on veut rendre un certain modèle 3D, nous n'aurons pas à bind à chaque fois le Vertex Array avec les attributs et toutes données uniforms que l'on veut commun à ce type d'objet. Mais pour chaque objet contenu dans la hashMap on pourra quand même créer une model Matrice propre à chaque objet, ainsi que toutes les données uniforms que l'on veut propre à un objet.

Listing 50. Méthode d'ajout dans la hashMap

```

public void processEntity(Object3D entity, int i, Vector3f colorAction){
    Model entityModel = new Model(entity.getModel());
    entityModel.setColorID(Helper.IntegerToColor(i));
    entityModel.setColorAction(colorAction);
    List<Object3D> batch = entities.get(entityModel);

    if(batch!=null){
        batch.add(entity);
    }else{
        List<Object3D> newBatch = new ArrayList<Object3D>();

```

```

        newBatch.add(entity);
        entities.put(entityModel, newBatch);
    }
}

```

On regarde donc si pour certain model un batch à déjà été créé si oui on rajoute un object3D sinon on crée un nouveau batch et on ajoute l' object3D au nouveau batch créé.

Listing 51. Méthode de rendu des entités

```

public void render(Map<Model, List<Object3D>> entities, Map<Models, List<Object3D>>
    entities2, SeaFrameBuffers fbos, float distanceFog)
{
    for (Model model : entities.keySet()) {
        prepareModel(model, fbos, distanceFog);
        List<Object3D> batch = entities.get(model);

        for (Object3D entity : batch) {
            shader.loadColorAction(entity.getColorAction());
            shader.loadColorID(entity.getColorID());
            prepareInstance(entity);
            GL11.glDrawElements(GL11.GL_TRIANGLES,
                model.getRawModel().getVertexCount(), GL11.GL_UNSIGNED_INT, 0);
        }
        unbindModel();
    }
}

```

Pour la hashMap on va parcourir l'ensemble de ces clefs qui sont donc des models. Pour chacune d'elles, on va appeler la méthode `prepareModel` qui va s'occuper de bind les attributs chargés, les textures et autres données uniforms communes au model. Ensuite, on récupère un batch pour chaque model et on construit la transformation Matrix dans la méthode `prepareInstance`, puis on rend l'Object3D à l'aide de la méthode `glDrawElements`. Et on finit par appeler la méthode `unbindModel` qui, comme son nom l'indique se charge de désactiver les vertexs attributs, ainsi que de debind le vertex array.

Notre structure de rendu s'effectue en plusieurs passes de rendus. Plusieurs passes sont dédiées au rendu de la mer, un rendu pour la réflexion, une autre dédiée pour la réfraction, on rend dans celle-ci que le terrain et le ciel, on ne rend pas les Object3D pour optimiser un petit peu la charge de la carte graphique. Ensuite, il s'agira de rendre le multi-sample FrameBuffer qui comporte le `outputColor` ainsi que le `colorID`. On rend à l'intérieur du multi-sample FrameBuffer la mer, le ciel ainsi que tous les object3D. Par la suite, il ne reste plus que la passe de post-processing qui ne rend finalement qu'un quad.

6.3.4 La caméra

Un des intérêt premier d'avoir une représentation en trois dimensions, est de pouvoir se mouvoir dans l'espace.

Pour ce faire, il nous faut d'abord définir la matrice de projection, nous allons pour notre utiliser une représentation perspective. Dans une projection perspective, un point est tronqué à l'intérieur d'une pyramide frustum.

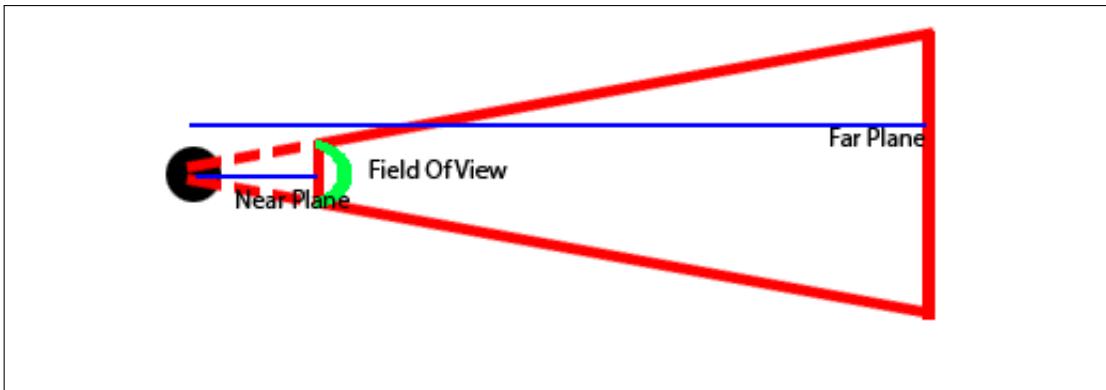


Figure 44. Frustum

Notre projection matricielle est définie comme suit :

Listing 52. Projection matricielle

```
fov = field of view = 45 sur MetaCiv
n = near_plane = 0,1 sur MetaCiv
f = far_plane = 10000 sur MetaCiv
a = aspect_ratio = display.width / display.height
```

$1/(\tan(fov/2))/a$	0	0	0
0	$1/(\tan(fov/2))$	0	0
0	0	$-(f+n)/(f-n)$	$-(2*f*n)/(f-n)$
0	0	-1	0

Table 4. Matrice de projection

Notre camera est définie par une classe, ayant pour attribut un angle pitch, un roll ainsi qu'un yaw. Le pitch correspond à l'angle sur l'axe X, le roll est l'angle sur l'axe Z et l'angle yaw correspond à l'axe Y. La camera possède une position dans l'espace, une position pour la cible de la camera, et une distance de la camera au point cible. On change la distance avec la mollette de la souris, on change le pitch avec le mouvement de la souris sur l'axe Y, le yaw est changé avec les mouvements de souris sur l'axe X. La position du point cible est modifiée à l'aide des touches ZQSD.

En utilisant la loi des Sinus, on récupère les valeurs horizontal et vertical comme suit :

Listing 53. Loi des Sinus

d : distance au point cible

```
HorizontalValue : d * cos(pitch)
VerticalValue : d * sin(pitch)
```

On peut enfin calculer la position de la camera :

Listing 54. Calcul la position de la camera

```
decalageX : HorizontalValue * sin(angle autour du point cible)
decalageZ : HorizontalValue * cos(angle autour du point cible)
```

```
position.x : poind cible.x - decalageX
position.x : poind cible.x + VerticalValue
position.x : poind cible.x - decalageZ
```

Notre camera possède aussi un système évitant à la camera de passer au travers du terrain. Pour ce faire, on récupère la position dans le tableau de vertices du terrain construit précédemment. Si la position de la camera sur l'axe Y est inférieur à la position du terrain sur l'axe Y, si oui on augmente la valeur du pitch jusqu'à 89 et on augmente ensuite la valeur de la distance au point cible.

6.4 Rendu graphique

Dans la partie qui va suivre, nous allons vous montrer comment sont construits les différents shaders qui composent les différents rendus de MetaCiv.

6.4.1 Rendu du terrain

Pour le rendu du terrain nous allons utiliser le modèle d'illumination dit de Phong. Le modèle de Phong est un modèle d'illumination qui se fait sur chaque point, ce modèle permet de calculer la lumière réfléchie sur un point, il combine pour cela trois éléments : lumière ambiante que l'on omettra dans notre rendu, lumière diffuse (modèle de Lambert) et la lumière spéculaire. Le modèle de Lambert est un modèle permettant de calculer facilement l'éclairage d'un point, il s'agit simplement de faire un produit scalaire entre la normale du point et le vecteur de la lumière.

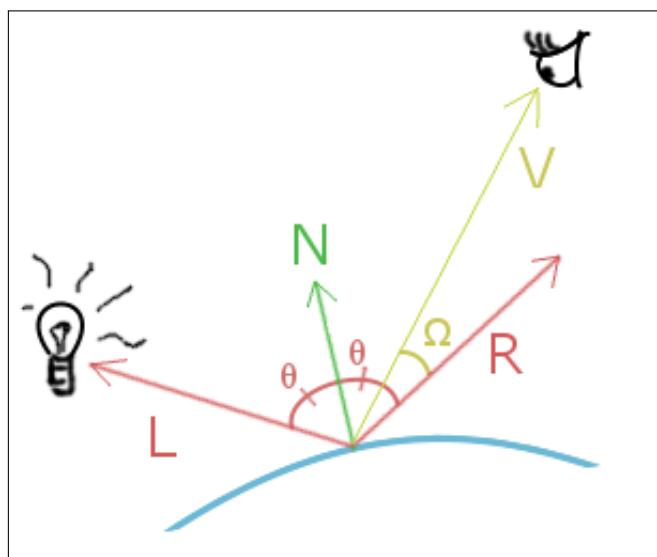


Figure 45. Modèle de Phong

Pour calculer la force de spéculaire, il suffit de récupérer le vecteur de réflexion de la lumière par rapport au point dont on souhaite calculer son éclairage. On procède avec la formule qui suit :

Soit L vecteur de la lumière, N normale au point et R le vecteur de réflexion. $R = 2(N \cdot L)N - L$

Mais GLSL possède déjà une fonction permettant d'effectuer ce calcul, `reflect` qui prend le vecteur de la lumière et la normale. Pour avoir la valeur finale du spéculaire, on prend le produit scalaire entre le vecteur R et le vecteur du point à la camera, on met ensuite à la puissance le `glossiness` ou `shine damper`, qui correspond à « l'étalement » du spéculaire, on multiplie finalement par la couleur du spéculaire. Une des particularités avec le rendu du terrain, est que l'on voulait avoir une transition lisse des textures, car dans les premières tentatives que l'on a effectuées, on apercevait la transition des textures en carreau (voir figure 46).

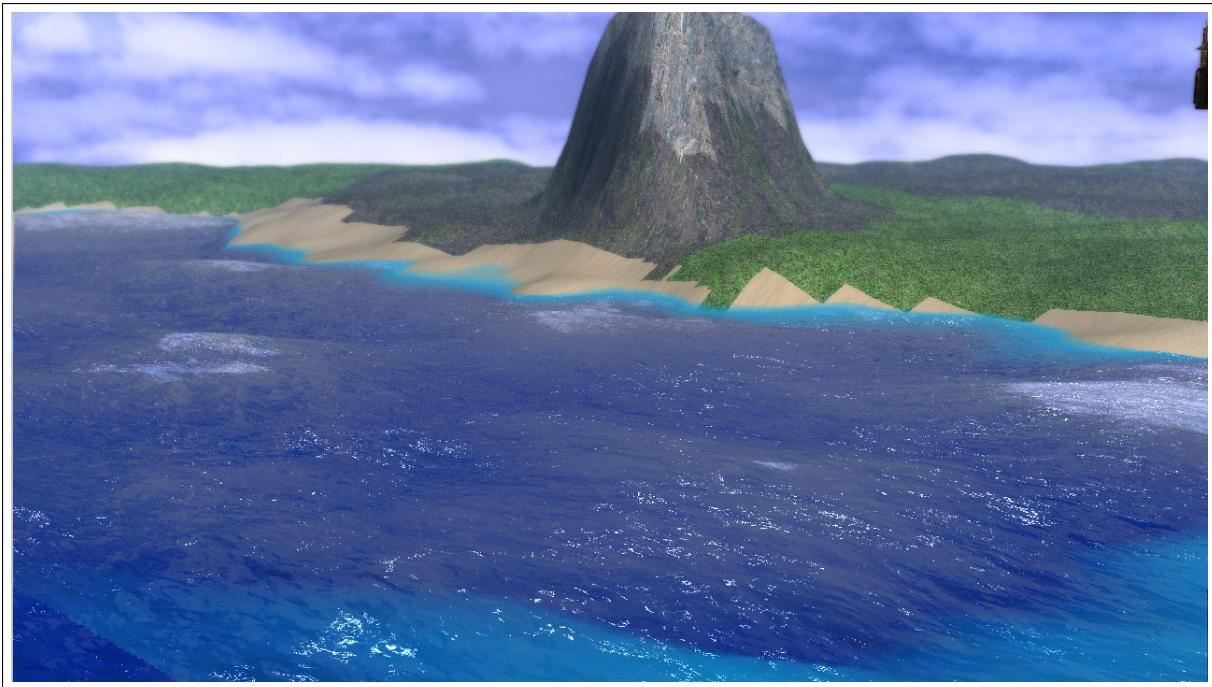


Figure 46. Ancienne version de MetaCiv avec une transition des textures grossière

Car en effet, avant on regardait la couleur de la texture tirée du `BufferedImage` que l'on récupérait et que l'on cherchait dans un tableau d'indices de textures avec une couleur correspondante, mais cette technique était plutôt coûteuse et le résultat était peu heureux.

Pour y remédier, nous avons finalement utilisé la hauteur du vertice pour matcher avec le tableau de type de terrain. Le problème, est que si l'utilisateur utilise la même hauteur pour deux types de terrains différents nous ne serions pas lequels choisir.

Listing 55. Assignation des textures à utiliser

```
float blendValue = 0;
if(fs_in.height >= heights[0].w){
    indice = 0;
} else {
    for(int i=1; i <= heights_size; i++){
        if(fs_in.height >= heights[i].w){
            indice = i-1;
            indice2 = i;

            blendValue = ((fs_in.height-heights[i].w) / (heights[i-1].w-heights[i].w));
            break;
        }
    }
}
if(indice == -1)
    indice = heights_size - 1;
```

On voit que l'on récupère deux indices et une valeur de mélange que nous utilisons pour le sampling des texture, que l'on mix entre elles.

MetaCiv possède aussi des routes qu'il nous fallait aussi représenter en 3D. Il nous suffit pour cela de regarder la couleur de la texture tirée du `BufferedImage`, si cette valeur est équivalente à la valeur noir, alors ce fragment est une route. Il nous suffit alors de plus utiliser les indices trouvés mais la texture de la route.

Pour un aspect esthétique, nous mettons sur les pentes fortes une texture de falaises. On récupère

pour ça la normale à la face directement dans le `fragment shader`. On utilise pour ça les fonction `dFdx`, `dFdy` qui prennent la position dans l'espace, et qui retournent la dérivée partielle en respectant la coordonnée `x` ou `y` de la fenêtre. Ensuite, on effectue un produit vectoriel entre les deux valeurs obtenues, on obtient ainsi la normale de la face. Ensuite, pour savoir si cette face est une pente forte on regarde la valeur `y` de la normale, si elle inférieure à 0,6 alors nous avons affaire à une falaise.

Comme le terrain utilise des textures atlas, il nous faut faire un traitement pour récupérer la coordonnée de textures des différentes textures composant la texture atlas. Pour le cas du terrain, il s'agit de la diffuse map et de la normal map. Le tout, en prenant garde de bien gérer la répétition des différentes textures.

Listing 56. Parsing des coordonnées de texture

```
float x = mod(fs_in.tc.x*tiling / 2.0, 0.5);
vec2 parseTc = vec2(x,fs_in.tc.y*tiling) / 4.0 + 0.25;
vec2 parseTcNrm = vec2(x,fs_in.tc.y*tiling) / 4.0 + (0.75);
```

On voit sur Listing 56 que l'on utilise un modulo pour assurer que la valeur `x` des coordonnées des textures soit bien contenu entre 0 et 0,5 comme on utilise deux textures par texture Atlas. Mais comme le fait d'utiliser des textures atlas avec répétition peut créer des décalages visibles, nous avons mis, par texture Atlas, deux textures diffuse suivit de deux textures de deux normal Map.

Une fois tout ça réaliser nous pouvons récupérer la normal dans les normal Maps de la bonne texture à utiliser.

Listing 57. Récupération des normales stockées dans les normales Map

```
if(!isCliff){
    if(isRoad){
        unitNormal = normalize((texture(roadMap, parseTcNrm).rgb * 2.0 - 1.0));
    } else {
        if(indice2== -1)
            unitNormal = normalize((texture(gSampler[indice], parseTcNrm).rgb * 2.0 -
                1.0));
        else
            unitNormal = normalize((mix(texture(gSampler[indice], parseTcNrm),
                texture(gSampler[indice2], parseTcNrm2), blendValue)).rgb * 2.0 - 1.0);
    }
} else
    unitNormal = normalize((texture(cliffMap, parseTcNrm).rgb * 2.0 - 1.0));
```

Puis réaliser tout nos calcul d'éclairage, et enfin sampler les textures diffuse.

Nous avons aussi offert la possibilité à l'utilisateur de mettre de la neige directement sur son terrain. Pour ce faire, nous regardons simplement si la valeur `y` de la normal de la `normalmap` est inférieur à :

$(\min((height)/(snow + snowAttenuation), snowDensity))$ alors on met en blanc en prenant soit de garder l'éclairage.

Tout nos rendus utilisent un système de brouillard qui fonctionne de la même façon pour chaque rendu. Quand la lumière est absorbée par le brouillard, ce phénomène est connu sous le nom d'`extinction`. Quand la lumière trouve un moyen de sortir du brouillard en rebondissant puis absorbée et ré-émie par les particules de brouillard, on parle alors de `inscattering`. Nous pouvons donc construire un simple modèle pour représenter l'`extinction` et `inscattering`, pour produire un brouillard simple et efficace. Dans le `fragment shader` on récupère la couleur de la texture obtenue classiquement, puis on l'applique au brouillard pour obtenir la couleur de sortie. On calcule la distance de la camera au point qui est rendu en utilisant simplement la norme du vecteur de la camera au point. Grâce à cette distance on peut calculer l'`extinction` et le `inscattering`.

$$E = e - d * Fe$$

$$I = e - d * Fi$$

Où d la distance au point, Fe facteur d'extinction, Fi facteur d'inscattering. Il suffit alors de calculer la couleur finale en multipliant la couleur de base par E pour garder ou diminuer la couleur de base. Ensuite, on additionne la couleur du brouillard multipliée par $1 - I$ pour obtenir à quelle force on doit utiliser la couleur du brouillard.

6.4.2 Rendu de la mer

Le rendu de la mer est le seul à utiliser la technologie de tesselation si l'utilisateur possède un équipement supportant la version 4.1 ou plus d'OpenGL. Sinon, par défaut nous utilisons un seul quad pour toute la mer, offrant ainsi la possibilité d'avoir une version plus légère de la mer.

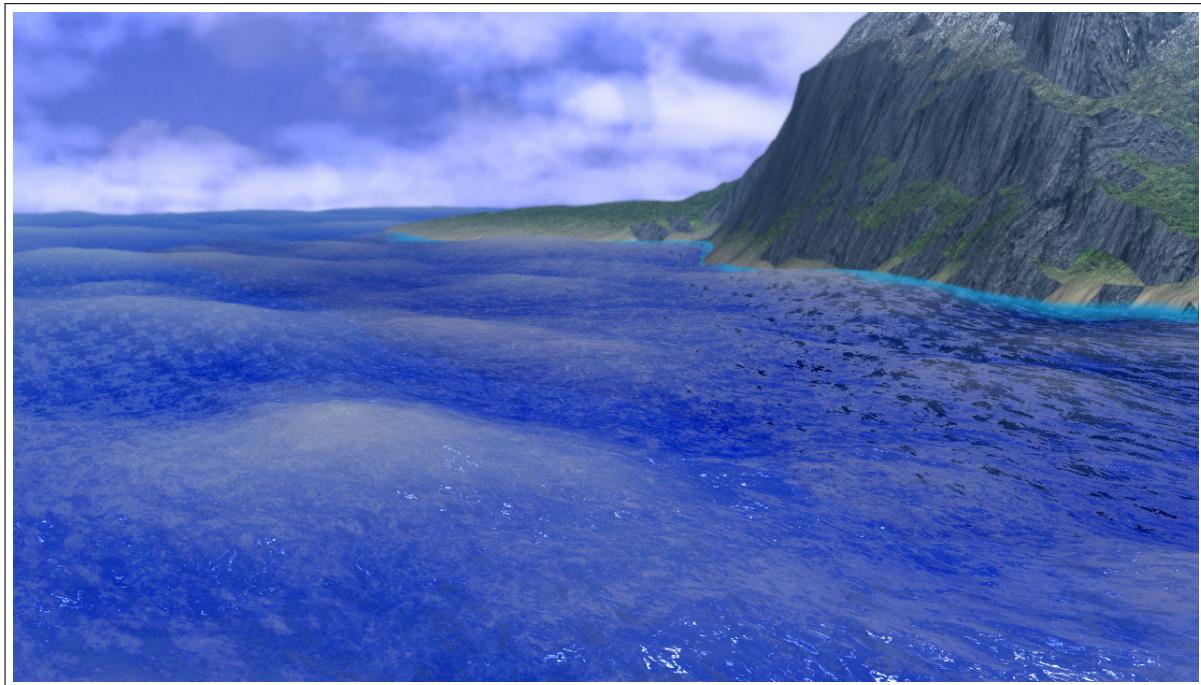


Figure 47. Mer avec tessellation



Figure 48. Mer sans tessellation

Pour donner une utilité à la tessellation, nous allons utiliser une displacement Map. Une displacement map est une texture contenant des informations relatives à la déformation d'un maillage. Nous pourrons donc pour chaque vertice obtenu, grâce à la tessellation, augmenter la hauteur sur l'axe Y.

La première étape de notre tessellation est de créer un simple vertex shader. Comme chaque patch est un simple quad, nous utiliserons donc une constante dans le shader pour représenter les quatres vertices, plutôt que de devoir mettre en place le vertex array. Nous utilisons une instance number (à l'aide de `gl_InstanceID` qui contient l'index de la primitive actuelle dans une commande de dessin instancié) pour calculer un décalage pour le patch. La mer est représentée par une grille de 64×64 patches, donc le décalage en x et y pour le patch est calculé en prenant le `gl_InstanceID` modulo 64 pour x et `gl_InstanceID / 64` pour y. Le vertex shader calcule aussi les coordonnées de texture pour le patch.

Listing 58. Vertex Shader de la mer

```
#version 410

out VS_OUT{
    vec2 tc;
}vs_out;

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
uniform vec3 lightPosition;
uniform vec3 cameraPosition;

void main(void) {
    const vec4 vertices[] = vec4[](vec4(-0.5,0.0,-0.5,1.0), vec4(0.5,0.0,-0.5,1.0),
        vec4(-0.5,0.0,0.5,1.0), vec4(0.5,0.0,0.5,1.0));
    float x = mod(gl_InstanceID, 64);
    int y = gl_InstanceID / 64;
    vec2 offs = vec2(x, y);
```

```

vs_out.tc = (vertices[gl_VertexID].xz + offs + vec2(0.5)) / 64.0;

vec4 worldPosition = vertices[gl_VertexID] + vec4(x, 0.0, float(y), 0.0);

gl_Position = worldPosition;
}

```

Pour nous assurer que les calculs s'effectuent dans le tessellation control shader soient effectués qu'une fois on regarde si `gl_InvocationID == 0`. Si oui, alors nous pouvons mettre en place le comportement de notre tesselation. Premièrement, on projette les coins du patch dans des coordonnées normalisées en multipliant les coordonnées entrantes par la `model-view` matrice et par la `projection` matrice et on divise chaque point par sa propre homogénéité. Ensuite, nous calculons la longueur de chaque arrête du patch dans un espace normalisé après les avoir projetées dans un plan `xy` en ignorant la composante `z`. Le shader peut alors calculer le niveau de tessellation pour chaque arrête du patch comme une fonction utilisant la longueur et une échelle qui permet de contrôler le niveau de tessellation.

Pour le tessellation evaluation shader, il calcule d'abord les nouvelles coordonnées de textures du vertex généré par une interpolation linéaire des coordonnées de textures passée par le tessellation control shader. Nous effectuons la même interpolation pour la position.

Pour donner l'effet d'une mer en mouvement, on va faire défiler la `displacement map`, mais plutôt que de simplement faire défiler la map qui donnerait un résultat convenable mais trop régulier, nous allons utiliser une `dudv map` qui est une map qui a pour objectif de dériver les coordonnées de textures. La `dudv` doit être tirée de la `normal map`, elle-même tirée de la `displacement map` pour avoir un résultat cohérent.

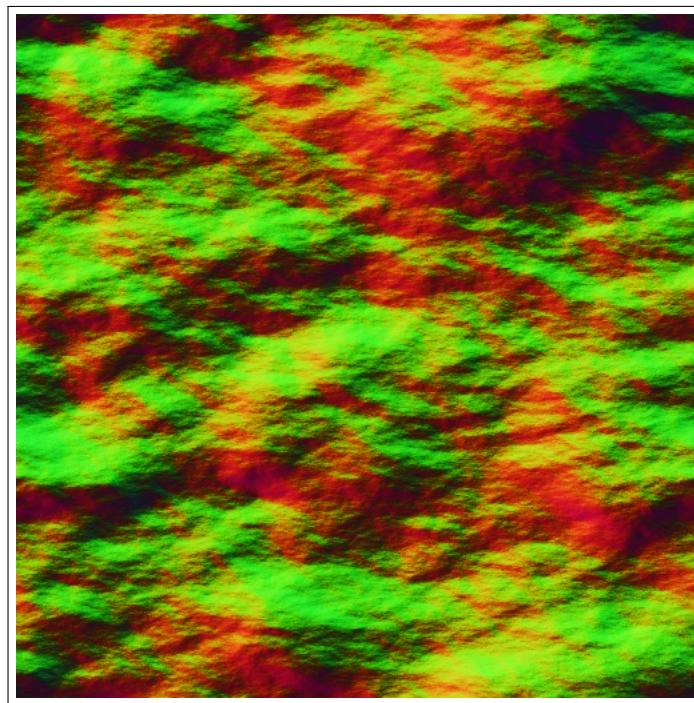


Figure 49. La DUDV map utilisée pour notre rendu

Pour récupérer nos nouvelles coordonnées de textures, il suffit de récupérer les composantes rouge et verte de la `duvd map` en multipliant par 2 et puis -1 pour passer de $[0, 1]$ à $[-1, 1]$.

Pour le rendu de notre mer nous utilisons donc une `dudv` pour récupérer une nouvelle coordonnée de texture en utilisant les anciennes coordonnées de texture, plus un mouvement de vague. Sauf que pour donner vraiment l'illusion de vague, on ne va pas utiliser qu'une seule fois la `dudv map`, car nous

récupérons aussi une nouvelle coordonnée de texture mais en utilisant les mouvements de vagues en négatif.

Listing 59. Distorsion des coordonnées de texture à l'aide d'une `dudv map`

```
distortion1 = (texture(dudvMap, vec2(tc.x/5.0 + moveFactor*0.2, tc.y/5.0 +
    moveFactor*0.2)).rg * 2.0 - 1.0) * waveStrenght;
distortion2 = (texture(dudvMap, vec2(tc.x/5.0 - moveFactor*0.2, tc.y/5.0 -
    moveFactor*0.2)).rg * 2.0 - 1.0) * waveStrenght;
```

Une fois nos nouvelles coordonnées de textures récupérées, nous pouvons calculer la hauteur du point à l'aide de la `displacement map`, nous regardons dans celle-ci la valeur de nos nouvelles coordonnées de textures puis on additionne les hauteurs.

Une fois toutes ses opérations effectuées, nous pouvons utiliser `gl_Position` pour dire à OpenGL la position du point dans l'espace, il suffit donc de multiplier la position du point par la matrice `model-view` et `projection`.

Une particularité pour faire un rendu d'eau est de considérer les réflexions et les réfractions. Pour ce faire nous aurons besoin de rendre notre scène dans deux `FrameBuffer` en prenant garde pour le cas de la réflexion d'inverser le pitch de la caméra avant le rendu. Pour pouvoir simuler une gestion de la profondeur de l'eau il nous faut pour la réfraction avoir un `FrameBuffer` avec un attachement pour la `depth texture`.

Dans le `fragment shader` de la mer, nous devons calculer les coordonnées de texture pour la réflexion et la réfraction. Il nous faut pour ça utiliser le `clipSpace` (coordonnées dans l'espace dans le `model-view-projection space`) pour ainsi faire une projection. Il nous faut ensuite normalisé les valeurs entre $[0, 1]$.

Nous simulons aussi le `fresnel`, qui est un phénomène observable, qui montre que plus l'angle avec laquelle on regarde surface est fort plus la surface paraît réfléchissante. Il suffit pour cela de regarder le produit scalaire entre le vecteur de la vue et celui de la normale de la surface. Ensuite, selon ce facteur on mixera la réflexion et la réfraction.

Nous pouvons percevoir sur les rendus que nous avons une couleur pour la côte. Pour y parvenir, on utilise la `depth texture` pour savoir à quel point l'eau est profonde.

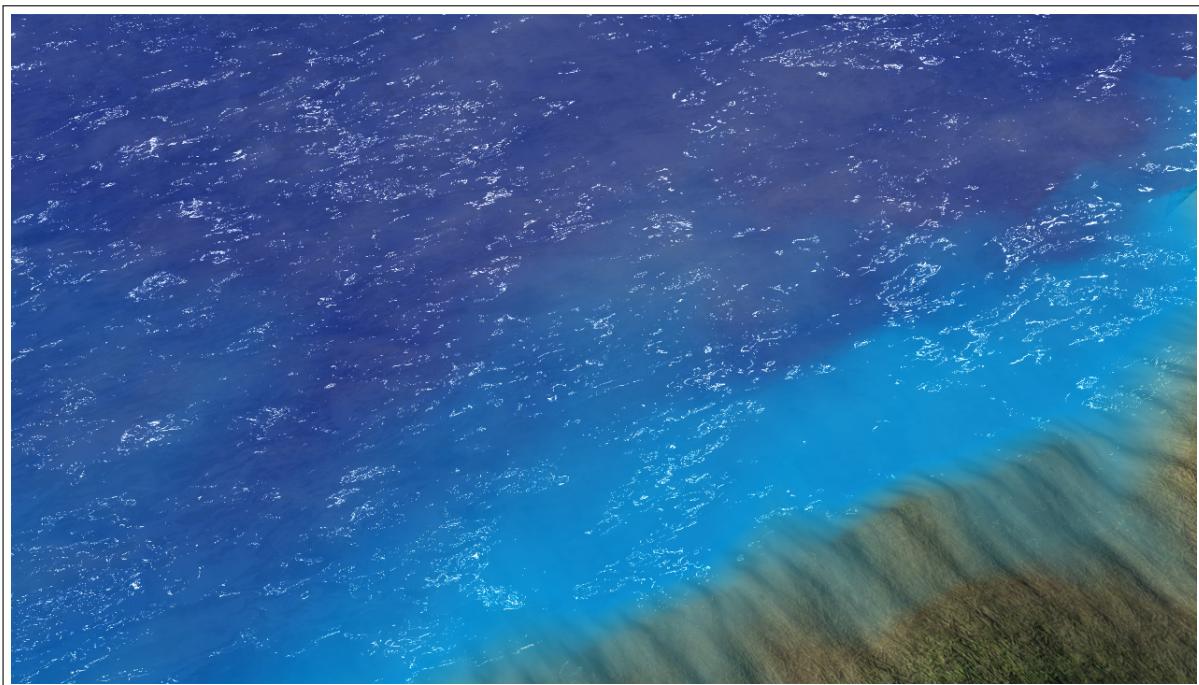


Figure 50. Profondeur de l'eau pour visualiser les côtes

Pour donner l'illusion d'avoir de la houle, on utilise les normales que l'on amplifie avec la hauteur.

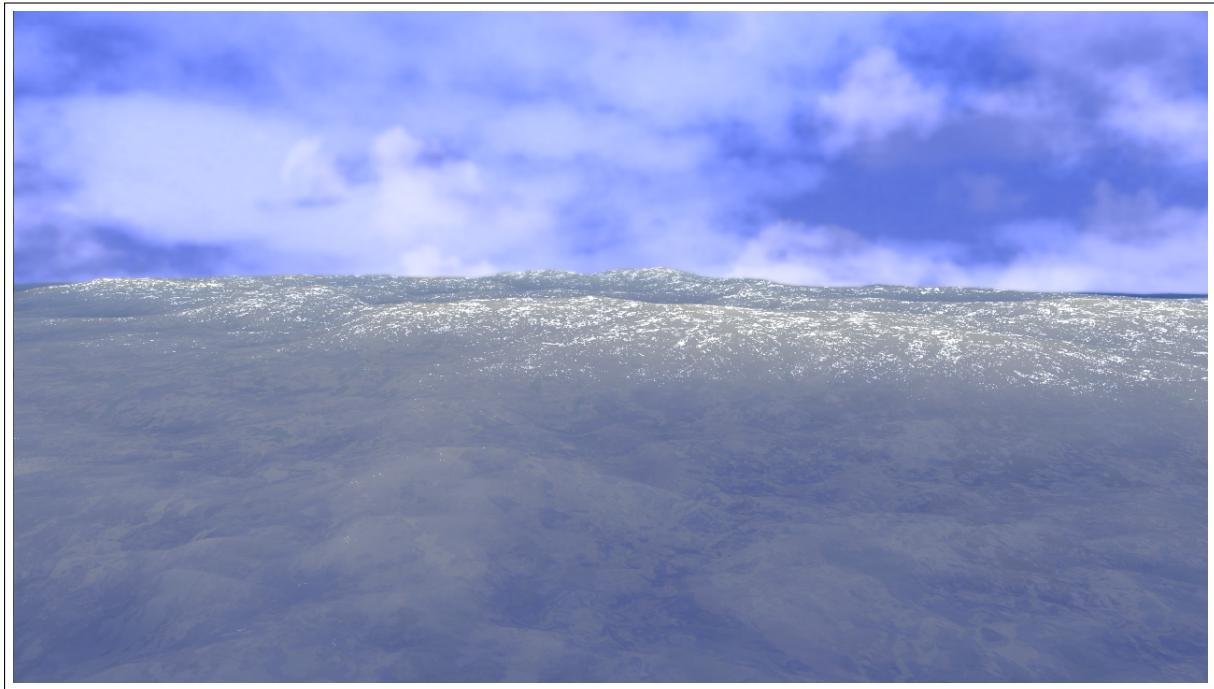


Figure 51. Houle de la mer

6.4.3 Physically Based Rendering

Le modèle de rendu de PBR (Physically Based Rendering), se base sur des rendus de lumières plus réalistes que le classique modèle d'illumination de Phong. Il utilise un système de réflexion. En utilisant un panel large de types de textures, on peut réaliser ce que l'on appelle des matériaux intelligents. Il nous faut pour cela une texture de diffuse qui définit la couleur de l'objet, une normale map qui définit les normales des aspérités de l'objet, une texture de rugosité qui définit la restitution de la lumière par l'objet et une texture de métal qui définit à quel point une surface est réfléchissante. Notre modèle PBR est un modèle simplifié car nous n'utilisons pas la global illumination.

Pour optimiser la simulation nous allons utiliser une texture equirectangular simple pour la réflexion, à la place de rendre une véritable cube map à chaque objet.

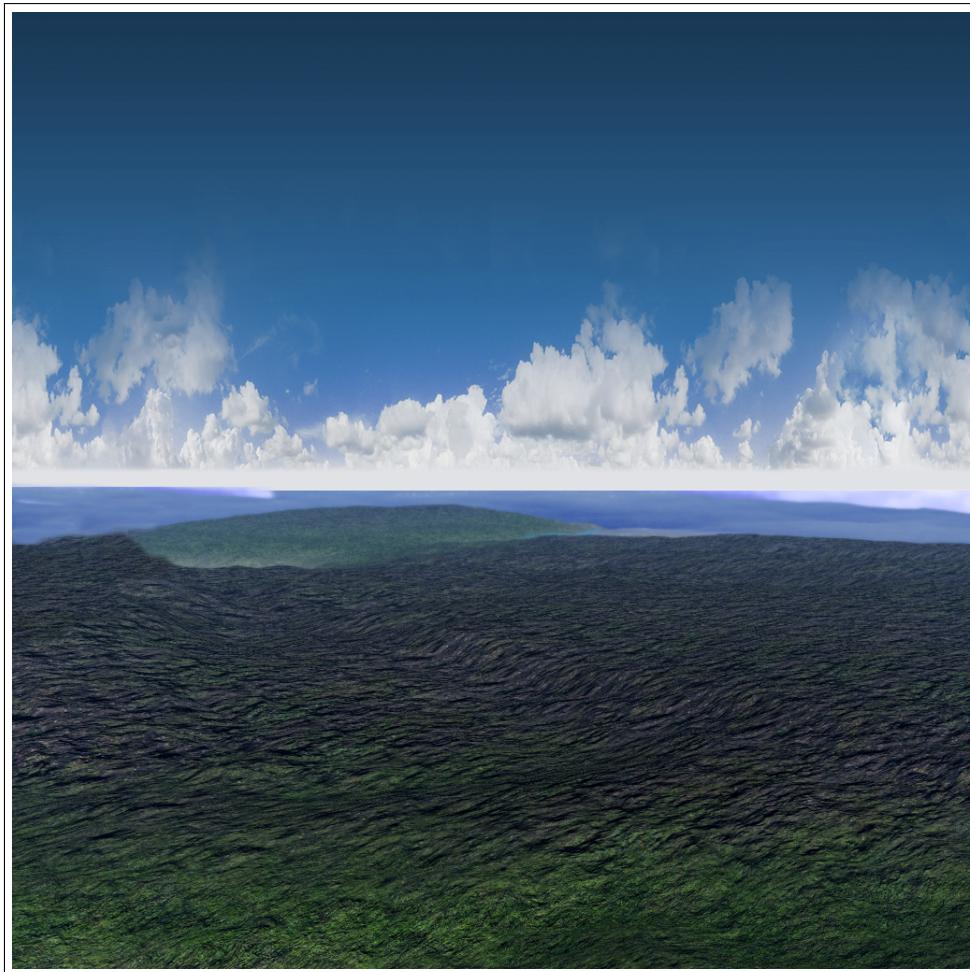


Figure 52. Equirectangulaire texture pour la réflexion

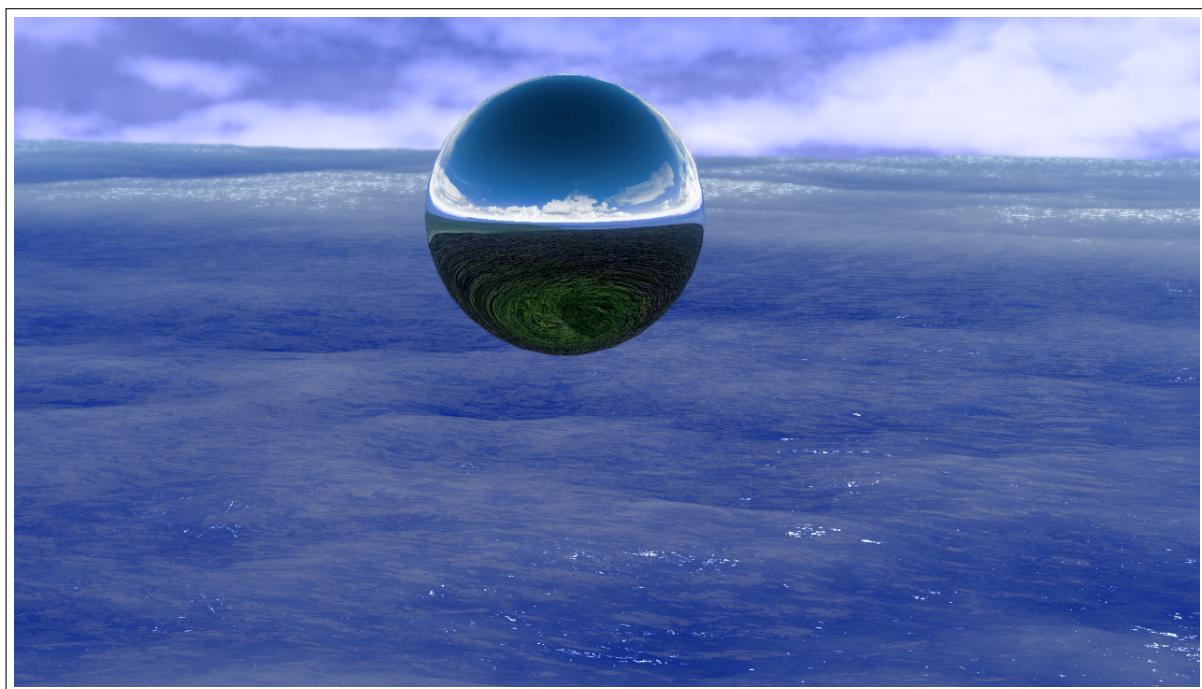


Figure 53. Rendu de la réflexion sur une sphère

Pour obtenir les coordonnées de texture permettant de rendre la réflexion basée sur la direction du vecteur de réflexion, il faut obtenir ce vecteur grâce à la fonction glsl reflect. La rugosité est simulée quant à elle grâce à un flou de l'image de réflexion. On utilise pour ça le niveau de détail de la texture de réflexion que l'on fait varier avec le niveau de gris de la texture de rugosité. Et pour le côté métal on récupère le niveau de gris de la texture de metal et on mix la couleur de base et la couleur de réflexion en prenant en compte le fresnel.

Listing 60. Gestion des réflexions dans le PBR

```
if(reflMapped){  
    vec4 gloss_color;  
    float gloss = texture(roughnessMap, texCoords).r;  
    float fresnelFactor = max(1-dot(unitVectorToCamera, unitNormal), 0.8);  
    vec3 r;  
  
    if(normalMapped)  
        r = reflect(normalize(fs_in.toCameraVector), normalize(fs_in.RM * (unitNormal *  
            fs_in.TBN)));  
    else  
        r = reflect(normalize(fs_in.toCameraVector), normalize(fs_in.RM * (unitNormal)));  
  
    //Compute texture coordinate based on direction  
    vec2 tc;  
  
    tc.y = r.y;  
    r.y = 0.0;  
    tc.x = normalize(r).x * 0.5;  
  
    float s = sign(r.z) * 0.5;  
    tc.s = 0.75 - s * (0.5 - tc.s);  
    tc.t = 0.5 + 0.5 * tc.t;  
    gloss_color = vec4(0, 0, 0, 1.0);  
  
    float lod = (5.0 + 5.0*sin(gloss))*step(tc.x, tc.y);  
    gloss_color = textureLod(reflexion_map, tc, gloss*10*2.5);  
  
    vec4 final_gloss = FragmentColor0 * gloss_color;  
  
    if(metalMapped)  
        FragmentColor0 = mix(FragmentColor0, final_gloss,  
            (fresnelFactor)*texture(metalMap, texCoords).r);  
    else  
        FragmentColor0 = mix(FragmentColor0, final_gloss, 0.5);  
}
```

Pour pouvoir utiliser les normales map, pour les comparer au terrain nous avons besoin de passer dans l'espace des tangentes. Car nous pouvons avoir de forte courbe comme dans le cas de la sphère, donc si on ne passe pas dans l'espace des tangentes toutes les normales seront dans le même espace et le résultat sera décevant.

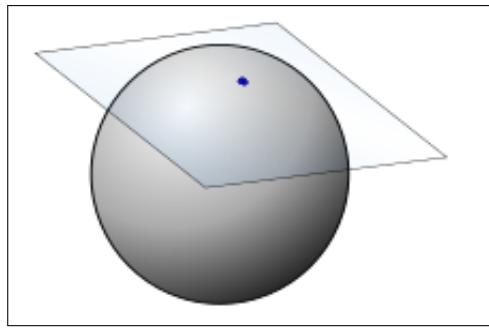


Figure 54. Espace tangentiel d'un point

Nous devons pour ça créer la matrice de l'espace tangentiel simplement comme suit :

Tangent x	Tangent y	Tangent z
BiTangent x	BiTangent y	BiTangent z
Normale x	Normale y	Normale z

Table 5. Matrice de l'espace tangentiel

Une fois cette matrice obtenue il suffit de multiplier le vecteur de vue et de lumière par cette même matrice, et d'utiliser ses vecteurs obtenus pour le calcul de lumière.

Fait important, pour utiliser la réflexion avec les normales map il faudra repasser dans l'espace du modèle en utilisant une matrice construite à l'aide de la matrice de transformation.

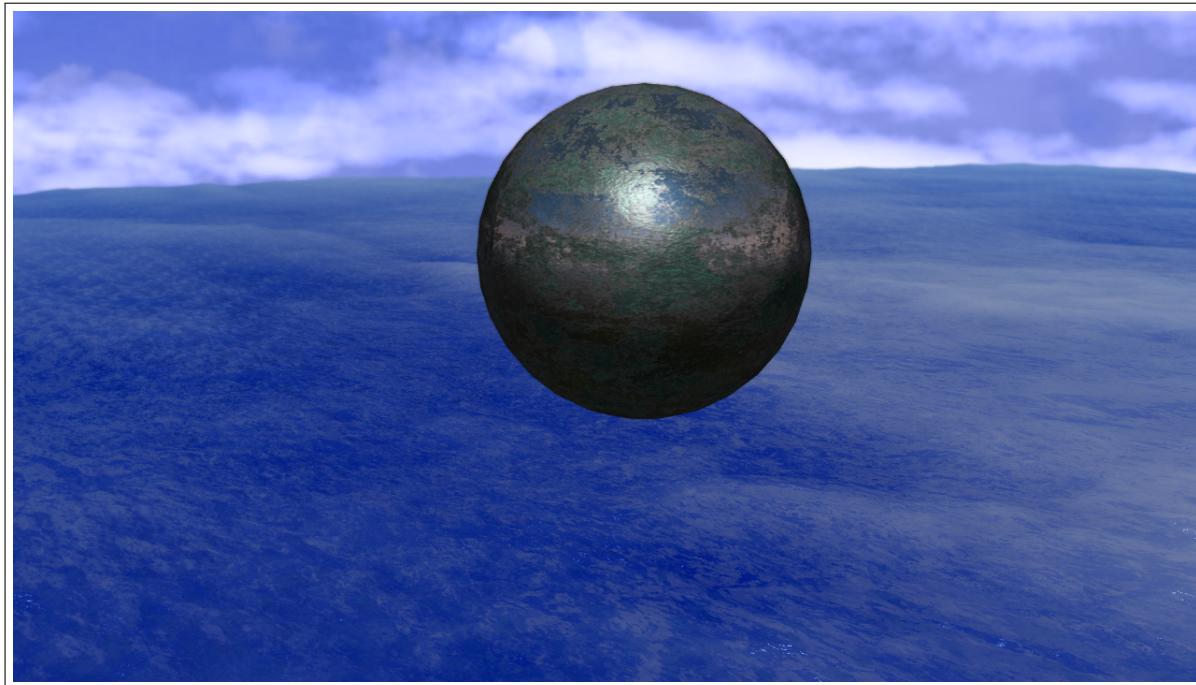


Figure 55. Rendu final du PBR

6.4.4 Post-Processing

Pour améliorer la qualité visuelle globale, nous allons utiliser une série de filtre post-processing, qui consiste à rendre une texture correspondante à la sortie du FrameBuffer, sur laquelle on va effectuer un traitement.

6.4.4.1 Bloom effect L'objectif de l'effet de bloom ou flou lumineux en français, est de rendre les objets comme s'il était éblouissant.

Lorsqu'un œil ou une caméra regarde une zone contenant une forte transition du lumineux au sombre, la lumière ainsi vue déborde sur la zone obscure. D'un point de vue physique cet effet optique est dû à la taille d'ouverture du diaphragme ou de la pupille qui produit une diffraction, chaque point lumineux devenant ainsi un disque d'Airy (figure résultant de la traversée d'un trou circulaire par la lumière).

La méthode utilisée consiste à récupérer la luminance de l'image en rajoutant un seuil pour éviter un glow trop important que l'on stocke dans une autre image. On applique ensuite sur cette image un gaussian blur. On peut grâce au gaussian blur augmenter l'étalement du bloom, soit en augmentant le noyau, soit dans le cadre d'une application en temps réel diminuer la dimension de l'image à blur. Cette image est ensuite additionnée à l'image de base.

Pour notre flou gaussien, nous allons effectuer deux filtres, un vertical et un horizontal. Pour ce faire, nous récupérons des coordonnées de texture horizontalement ou verticalement de 11 fragments puis nous effectuons un filtre gaussien.

Listing 61. Récupération des coordonnées de texture pour le flou horizontal

```
gl_Position = vec4(position, 0.0, 1.0);
vec2 centerTexCoords = position * 0.5 + 0.5;
float pixelSize = 1.0 / targetWidth;

for(int i = -5; i < 5; i++){
    blurTextureCoords[i+5] = centerTexCoords + vec2(pixelSize * i, 0.0);
}
```

Listing 62. Méthode pour récupérer la luminance

```
Y = rouge * 0.2126 + vert * 0.7152 + bleu * 0.0722;
```

6.4.4.2 Depth of field En optique, le depth of field (champ de profondeur) rend un objet plus flou selon la distance à l'objectif.

Pour simuler cette propriété optique, nous allons utiliser le depth texture qui stocke des informations relatives à la distance de l'objet, à la caméra, en niveau de gris.

Listing 63. Mix de la texture floutée avec la texture Sharp selon le depth buffer

```
#version 150

in vec2 textureCoords;

out vec4 out_Colour;

uniform sampler2D colourTexture;
uniform sampler2D blurTexture;
uniform sampler2D depthBuffer;

void main(void){
    float depthValue = (1-texture(depthBuffer, textureCoords).r) * 1800;
    vec4 sceneColour;
    if(depthValue<0.99)
        sceneColour = mix(texture(blurTexture, textureCoords), texture(colourTexture,
            textureCoords), depthValue);
    else
        sceneColour = texture(colourTexture, textureCoords);
```

```
    out_Colour = sceneColour;  
}
```

6.4.4.3 Contrast Balance En dernier lieu, nous ajustons un peu le contraste, comme suit :
$$FinalCouleur = (Couleur - 0,5) * (1,0 + contraste) + 0,5$$

6.4.5 Amélioration possibles

Les améliorations possibles pour la vue 3D de MetaCiv sont légions :

1. Dans un premier temps, une chose très importante qui manque dans notre version, est la représentation de la végétation ou des minéraux. Car MetaCiv présente un système de phéromone pour les baie, bois, métal, etc... pour leur représentation le moteur possède déjà un système de rendu instancié, que nous utilisions pour les routes.
2. Une autre amélioration possible est d'utiliser des animations pour les agents, ce qui permettra de visualiser les actions des agents plus efficacement qu'avec de simple couleur et les debug string. Mais comme la mise en place d'un système d'animation, de la lecture de fichiers stockant les animations à la lecture de celles-ci, est un processus très fastidieux et chronophage nous n'avons pas pu le mettre en place pour cette version de MetaCiv.
3. Certaines améliorations visuelles sont aussi possibles, en commençant par les ombres, on peu voir dans le code les traces d'une tentative de shadow mapping que nous avons pas eu le temps d'aboutir, mais elle peut servir de base pour une éventuelle implémentation. Au niveau du ciel aussi beaucoup d'effet son manquant, comme l'atmosphère scattering ainsi que la visualisation du soleil et tous les effet de post-production qui lui son lié, comme les light shaft (puit de lumière) ou god ray ainsi que les lens flare.
4. L'implémentation d'un véritable menu pour régler les paramètres graphiques, plutôt que devoir passer par des raccourcis.

6.4.6 Tutoriel

6.5 Contrôle de la vue 3D

- ZQSD : contrôle de la camera
- Déplacement Souris : rotation camera
- Mollette de la souris : zoom / de-zoom
- Clique sur un agent : Suivi de l'agent
- Barre espace : Suivi du premier agent
- Clique droit : Apparition du menu World permettant de faire apparaître les debug string, et voir les informations d'un agent si sélectionné.
- Alt+entrée : Plein écran
- F1/F2/F3 : Choix de la résolution
- 1/2 : Réglage de la force de tessellation
- 4/5 : Réglage de la vitesse de la simulation

6.6 Contrôle du terrain

Le contrôle du terrain ce fait dans l'éditeur d'environnement.

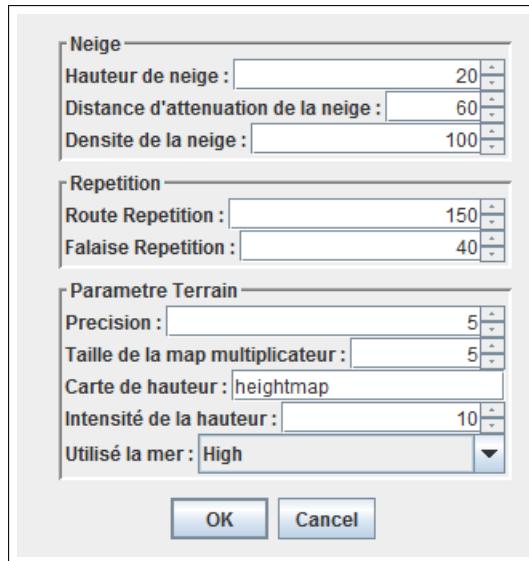


Figure 56. Éditeur d'environnement

Paramètres globaux de la vue 3D liés au terrain (voir figure 56), accessible via le bouton Paramètre 3D. Dans cette fenêtre on peut régler toutes les options liées à la neige, la répétition de la texture de la route et de la falaise. Ainsi que les paramètres du terrain, comme la précision qui correspond à quel point le terrain correspond à la vue 2D, le multiplicateur pour augmenter la taille du terrain en 3D sans avoir à toucher la taille de celle en 2D, choisir la heightMap, ainsi que son intensité et la qualité de la mer.

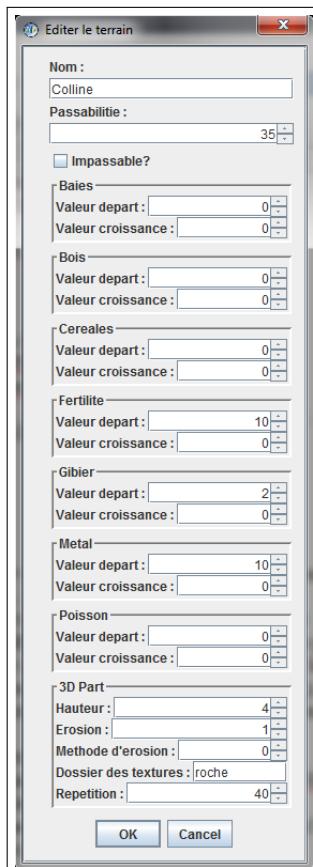


Figure 57. Éditeur de terrain

Avec cette fenêtre 57 on peut régler la hauteur pour ce type de terrain, son érosion, la méthode d'érosion, le dossier où sont stockées les textures pour ce type de terrain ainsi que leur répétition.

6.7 Logique des fichiers pour Tortues/Facilities/Terrain

Tous les fichiers personnelles que l'utilisateur souhaite utiliser, pour sa simulation, sont stockés dans un dossier Skin, à la racine du modèle. Pour tous les types de terrains, on stockera dans un dossier, que l'on spécifiera. Pour les aménagements, il faut un dossier Skin/Amenagement/(nom de l'aménagement)/(nom de l'aménagement).obj

Pour les civilisations, il faut un dossier Skin/Civilisations/(nom de la civilisation).

Pour stocker les textures il faut un dossier du nom de l'objet 3D, en respectant une certaine nomenclature.

- Pour la diffuse : Diffuse.png, diffuse.png, Albedo.png, albedo.png, Base_Color.png, base_color.png, Base_color.png, base_Color.png, BaseColor.png, Basecolor.png, baseColor.png, basecolor.png
- Pour la normale Map : normal.png, Normal.png, Normal_OpenGL.png
- Pour la rugosité Map : Roughness.png, roughness.png
- Pour la métal Map : Metallic.png, metallic.png, Metal.png, metal.png

6.8 Conclusion de la partie 3D

Pour conclure sur la partie 3D de MetaCiv, nous pouvons dire que ce fut une expérience forte d'enseignement. Cela nous a permis de prendre en main beaucoup de technologies récentes comme la tessellation. Ainsi que beaucoup de techniques pour représenter des phénomènes physique observables, comme l'eau, bloom... Ce fut aussi une grande expérience pour la mise en place d'une technologie sur une application déjà existante. Certes cela a été une tâche peu aisée et chronophage mais nous sommes satisfait du résultat final.

7 Conclusion

En conclusion, nous avons terminé nos tâches avec succès. Le sujet nous a motivé dès le début et l'encadrement du projet à fait que nous n'avons jamais perdu cette motivation. Nous aurions aimé développer une civilisation nouvelle, avec de nouvelles capacités, mais nos tâches étaient déjà bien trop importantes. Nous avons tout de même amené la 3D dans MetaCiv, ce qui, nous pensons, va révolutionner le logiciel.

La reprise d'un projet qui a déjà vu plusieurs équipes se succéder n'est jamais facile. Nous avons dû prendre un temps de découverte pour pouvoir voir tous les tenants et aboutissants. Puis, nous avons essayé de coder en appliquant les consignes de génie logicielle, pour permettre l'évolution du projet ainsi que sa maintenance.

Ce TER nous a demandé une multitude de compétences, notamment en IHM, génie logicielle, 3D, Java, OpenGL et en gestion de projet. Nous sommes satisfait du travail accompli malgré les difficultés rencontrées. Ainsi que d'avoir pu mettre toutes ces compétences au sein d'un projet aussi vaste que MetaCiv. Nous pensons avoir amélioré le logiciel, et nous espérons que cette incrémentation va satisfaire les utilisateurs finaux.

Bibliographie

- [1] Fabien Michel Jacques Ferber, Olivier Gutknecht. Madkit website, 2016. URL <http://www.madkit.net/madkit>.
- [2] Fabien Michel Jacques Ferber, Olivier Gutknecht. Turtlekit website, 2016. URL <http://www.madkit.net/turtlekit/>.
- [3] MIT Media Lab Lifelong Kindergarten. Scratch website, 2016. URL <https://scratch.mit.edu/>.
- [4] Oracle. Java website, 2016. URL <https://www.java.com/fr/>.