Département de génie logiciel et des TI
École de technologie supérieure

Rapport final

Évaluation numéro	3
Étudiant	Francis Charette Migneault
Code permanent	CHAF06089108
Code universel	AK02470
Cours	MTI835
Session	Été 2016
Groupe	01
Professeur(e)	Éric Paquette

1. Environnement de développement

L'engin de jeu *Unity* est celui qui est employé pour ce projet afin de réaliser un jeu de type *Real-Time Strategy (RTS)*. Cet *API* emploie principalement une interface graphique de modélisation hiérarchique conviviale permettant d'éditer facilement et rapidement des liens unissant divers objets présents dans la scène de jeu. Aussi, une librairie de ressources (*Asset Store*) est disponible afin d'y retirer des implémentations de codes courantes dans les jeux. On y retrouve aussi de multiples objets 3D modélisés avec grand détail pour les utiliser dans les jeux développés. Ces deux caractéristiques en fait donc un outil très intéressant pour la conception aisée des fonctionnalités de base, permettant alors de travailler plus en profondeur sur les éléments avancés du jeu désiré. Aussi, les rendus visuels des modèles dans *Unity* sont particulièrement impressionnants considérant la vitesse à laquelle ils peuvent parfois être réalisés en utilisant les fonctionnalités supportées par défaut avec les engins de physique, de collisions et d'éclairage. De plus, la communauté et la documentation sont extrêmement grandes pour cet outil de développement, ce qui aide beaucoup pour trouver tant des solutions à des problèmes que pour découvrir de nouvelles techniques plus performantes ou standardisées d'application des concepts d'infographie. Les applications réalisées dans *Unity* peuvent être déployées sur plusieurs plateformes, ce qui limite la quantité de code à produire et accélère la production des jeux.

Pour l'implémentation de fonctionnalités particulières, *Unity* fait usage de scripts *C#* pouvant être attachés aux objets dans la scène. Dans le cadre de ce projet, l'*IDE MonoDevelop* inclut avec l'installation d'*Unity* est employé pour éditer ces scripts. Aussi, l'interface *UnityEditor* est employée pour la mise en place des quelques liens relationnels simples entre les objets ou de leurs options de paramétrisation. Les liens entre les objets sont éditables tant avec de simples glissements (*drag-and-drop*) des éléments dans l'interface usager, que par des méthodes programmées pour avoir des comportements selon des liens dynamiques. Ces liens sont donc gérés selon les besoins de l'application. Les *GameObject* contiennent par défaut (sauf si vide) des classes pour les transformations géométriques, la texture, l'affichage du rendu et le calcul des collisions. Plusieurs autres classes très spécialisées peuvent être rattachées comme l'éclairage, une caméra, des modules audio, des scripts et même d'autres *GameObjets* selon les fonctionnalités recherchées, ce qui rend leur utilisation très polyvalente. Lors de la programmation avec *Unity*, il faut se rappeler d'employer un système d'axes main-gauche. Par contre, il est toujours possible de se référer aux vues de l'interface graphique pour valider les comportements et positions des objets dans la scène 3D.

2. Objectifs

Comme mentionné précédemment, ce projet consiste à réaliser un jeu *RTS*. Ainsi, plusieurs des requis qui suivront sont portent sur le positionnement et la modélisation de bâtiments et unités de combat sur la carte. Aussi, la gestion des menus d'affichages pour les informations d'unités et de bâtiments disponibles est à réaliser. Certains autres aspects touchent plus aux contrôles caméras et du terrain de jeu. Au fur et à mesure que le développement du jeu progressera, les fonctionnalités de simulation et d'intelligence artificielle plus avancées pourront être ajoutées.

2.1. Tableau récapitulatif

TABLEAU RÉCAPITULATIF :

Objectif	Évolution	Atteinte	Temps	Facteur
Contrôle de caméra RTS	terminé	100%	3	**1.5**
Bordures de terrain dynamiques	terminé	100%	0.5	**0.5**
Positionnement centré du terrain	terminé	100%	0.5	**0.5**
Création du graphe de scène des unités	terminé	100%	3	**1.5**
Application des textures aux modèles	terminé	100%	1	**1.0**
Animation du canon d'une unité	terminé	100%	5	**2.0**
Sélection d'unités et commande d'attaque	terminé	100%	3	**3.5**
Importer et implémenter les bâtiments	terminé	100%	6	**2.5**
Rotation et positionnement de bâtiments	terminé	100%	8	**3.5**
Interface pour choisir les bâtiments	terminé	100%	4	**1.0**
Sélection de bâtiments existants	terminé	100%	2	**1.5**
Modélisation d'un terrain de base	terminé	100%	3	**0.5**
Déplacements de base des unités	terminé	100%	5	**1.5**
Contournement d'obstacles	terminé	100%	24	**2.0-3.5**
Projectile et effet d'impact	terminé	100%	8	**2.5**
Affichage de mini-carte	terminé	100%	7	**2.0**
Sélection par région de multiples unités	terminé	100%	6	**1.0-2.0**
Affichage de la vie d'une unité	terminé	100%	4	**1.0-2.0**
Effet de poussière d'unité en mouvement	terminé	100%	5	**0.5-1.5**
Destruction d'une unité	terminé	100%	5	**1.0-1.5**

LÉGENDE :

Gras	Objectif qui sera travaillé d'ici le prochain rapport
Italique	Objectif qui a été travaillé depuis la dernière évaluation

2.2. Objectifs terminés

Objectif : Contrôle de caméra RTS

Description :
La caméra permettant l'affichage de la scène doit être contrôlée semblablement aux jeux typiques de type *RTS*. Le but est de modifier un *Asset* de base disponible sur l'*Asset Store* afin d'avoir une structure initiale d'une caméra qui accomplit les fonctionnalités générales comme capter les flèches du clavier pour calculer les vecteurs de déplacement et les appliquer. Ainsi, la caméra est initialisée avec une vue *top-down* à 45º pour visualiser le terrain. L'angle de vue horizontal peut toutefois être ajusté pendant la partie en maintenant « Right-Click » enfoncé et en déplaçant la souris. Les déplacements vertical et horizontal de la caméra devront aussi être limités par les bordures du terrain formant la carte de jeu pour ne pas déborder de ses limites. La caméra pourra être déplacée le long du plan du terrain autant à l'aide des touches « WASD », des flèches de direction ou en plaçant la souris sur la bordure de l'écran vers la direction voulue de déplacement panoramique. La position de la caméra en hauteur, soit le zoom par rapport au terrain, pourra être contrôlée tant avec des boutons pré-assignés ou avec la roulette de la souris en respectant des limites minimales et maximales pour ne pas traverser le terrain et pour ne pas être trop éloigné en hauteur.

Réalisation :
L'*Asset* employé [25] implémente, entre autres, les déplacements panoramiques le long de la carte de jeu ainsi que les rotations avec la souris et certaines touches. Les mouvements principaux des jeux typiques *RTS* sont supportés avec les entrées clavier et souris standard. Ces entrées sont toutes définies avec des paramètres qui pourraient plus tard être ajustés avec un menu d'option.

Puisque l'exemple de référence employé ne limite pas particulièrement les angles de vue, les ajustements des paramètres ont été accomplis pour obtenir la disposition à 45º présentée à la Fig. 1. Cet angle de visée de la caméra est imposé pour que celle-ci ne puisse pas être tournée autour de son axe X, ce qui la renverserait verticalement et poserait problème.

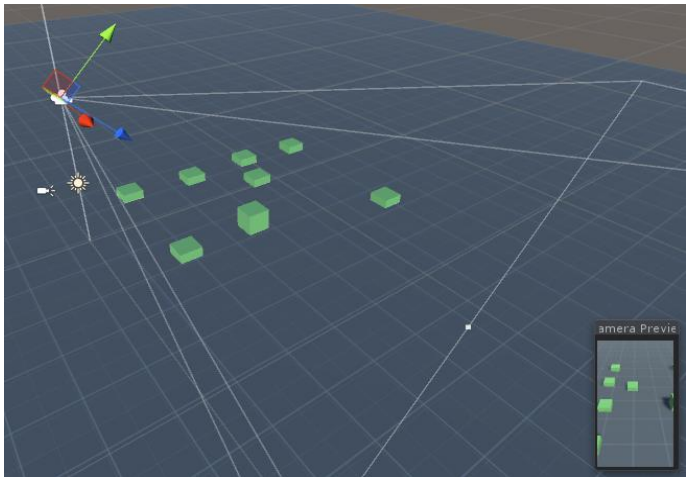

Fig. 1: Caméra visualisant le terrain à 45º vers le bas en style *RTS*

Le zoom qui était implémenté dans le script de référence se faisait seulement selon la hauteur normale au plan de jeu plutôt que d'en avant vers l'arrière par rapport à l'axe Z de la caméra (où elle regarde). Donc, le script a été modifié de sorte à accomplir la fonctionnalité voulue telle qu'illustrée à la Fig. 2 en effectuant des translations colinéaires relatives au vecteur en Z de la caméra. Dans ce script, les limites de hauteurs minimale et maximale ont aussi été ajustées de sorte à s'assurer de ne pas passer à travers le terrain tout en limitant l'éloignement du terrain à une distance raisonnablement rapprochée.

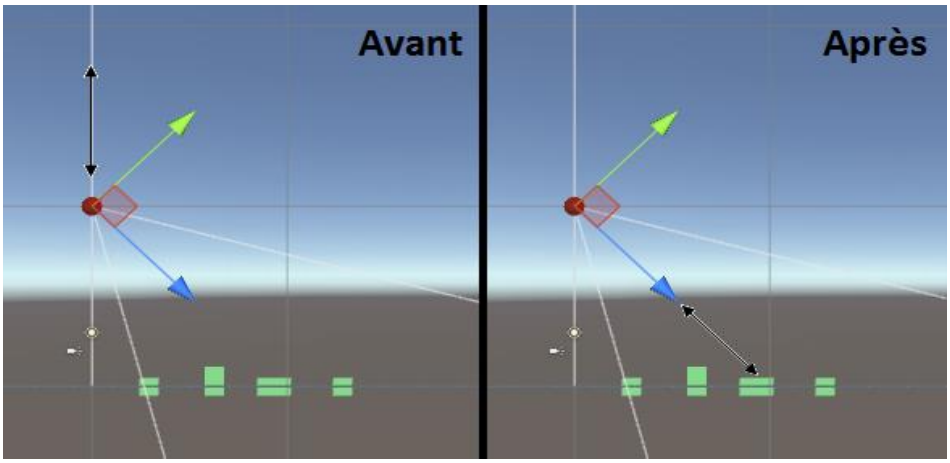

Fig. 2: Ajustement de la méthode de zoom de la caméra

Finalement, les valeurs des vecteurs de position et de rotation de la caméra ont été affichées sur le *canvas* de l'interface usager à l'aide d'un script de déverminage nommé « *DisplayDebugText.cs* » afin de vérifier la fonctionnalité du code. La vidéo « *camera-controls-limits.mp4* » démontre l'application des mouvements implémentés de la caméra *RTS*. Dans cette vidéo, les fonctionnalités de zoom, rotation, translation et limitation par dimensions du terrain et par hauteur sont toutes présentées en utilisant en combinaison la souris et le clavier. Les limites appliquées en Y sont de 2 à 50, alors que les limites en XZ sont automatiquement de ±50 étant donné que le terrain est de dimension 100x100 et est centré à l'origine selon le script qui lui est attaché (voir objectif « *Positionnement centré du terrain* »).

<u>Objectif :</u> Bordures de terrain dynamiques

<u>Description :</u>
Les limites de terrain n'étant pas initialement dynamiques en fonction la taille de la carte pouvant varier faisait que le contrôle de la caméra *RTS* n'était pas consistant à l'objectif « *Contrôle de caméra RTS* ». Un script permettant la mise à jour des dimensions du terrain est nécessaire.

<u>Réalisation :</u>
Un script nommé « *ApplyTerrainBorderLimits.cs* » prenant comme entrée le terrain à visualiser a été implémenté pour ajuster les limites de déplacement de la caméra en fonction des dimensions de la carte. Le script effectue simplement une mise à jour des positions ne

pouvant pas être dépassées par la caméra lors de l'initialisation en fonction des dimensions courantes du terrain. Ce script a été attaché à la caméra transférée en *prefab* et le terrain, également en *prefab*, a été associé par la variable publique. Maintenant, l'ajustement des limites s'accomplit automatiquement si le terrain se voit redimensionné avec l'éditeur, ce qui permet maintenant de ne plus avoir besoin de se soucier si les ajustements sont reflétés sur la caméra.

Une vidéo nommée « *terrain-dynamic-border-limits.mp4* » démontre la mise en application de ce script où le terrain est initialement de dimension 400x400, et se voit redimensionné à 100x200. Pour les deux cas de dimensions appliquées, les limites des bordures sont respectées par la caméra qui ne sort pas du terrain alors que les flèches du clavier sont maintenues pour aller vers l'extérieur de la carte.

Objectif : Positionnement centré du terrain

Description :
L'objet de type *Terrain* est employé pour simuler la surface de jeu où les unités peuvent s'affronter, tel que décrit à l'objectif « *Modélisation d'un terrain de base* ». Par contre, cet objet n'est pas centré à la référence *World* par défaut puisque son origine (0,0,0) est située dans son coin inférieur gauche, comme démontré à la Fig. 3. Lorsque de nouvelles unités ou que des bâtiments doivent être instanciées sur le terrain durant la partie, il est peu commode de toujours avoir à gérer des translations par rapport au terrain pour s'assurer qu'ils s'y trouvent au-dessus, surtout en considérant que les dimensions du terrain ne sont pas fixes (voir « *Bordures de terrain dynamiques* »). Ainsi, il est plus pratique de recentrer le terrain à l'origine *World*, ce qui permettra d'instancier les objets à l'origine tout en s'assurant qu'ils se trouvent sur le terrain simultanément. Cela facilitera la gestion des positions des divers éléments de la partie qui seront alors toutes par rapport à la même origine *World*. Cet objectif consiste donc à créer un script qui sera affecté au terrain afin de faire correspondre son centre avec l'origine du monde, tout en respectant ses dimensions dynamiques.

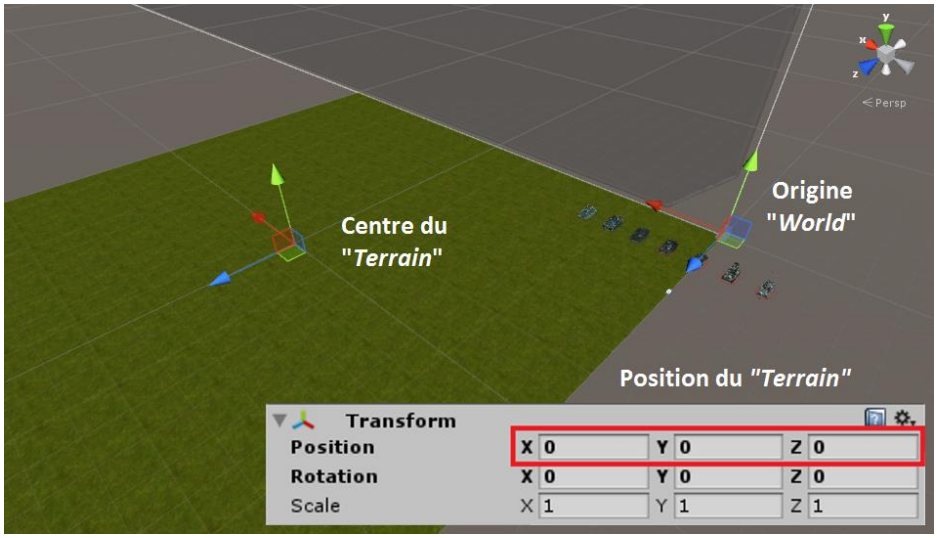

Fig. 3: Position par défaut du terrain par rapport à l'origine *World*

Réalisation :
Un script appelé « *CenterTerrain.cs* » a été attaché à l'objet « *Terrain* » afin d'appliquer une translation de $(-½, 0, -½) \times TerrainSize$ au vecteur « *transform.position* » de l'objet, permettant d'assurer une position centrée à l'origine peu importe la largeur et la hauteur qui lui sont appliquées. Le résultat avant et après application du script permettant de centrer le terrain est démontré à la Fig. 4.

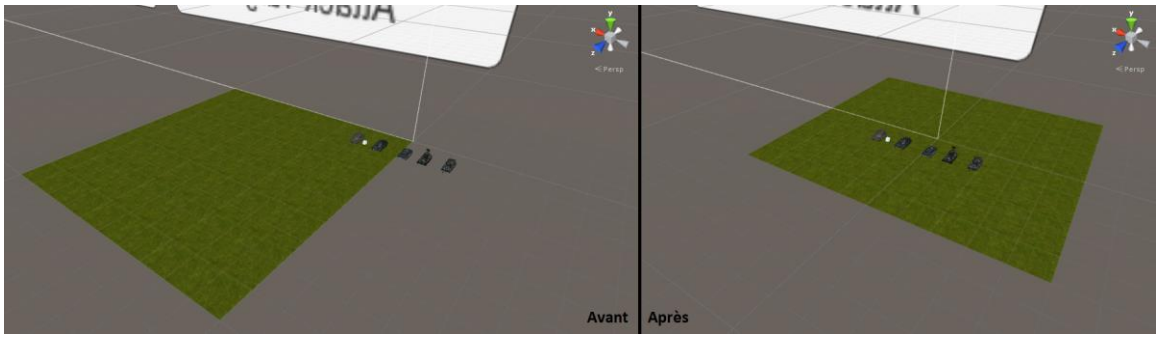

Fig. 4: Repositionnement du terrain centré à l'origine *World*

Également, une vidéo nommée « *terrain-center-world-dynamic.mp4* » est disponible pour illustrer la mise en application de ce script alors que la partie est lancée. Dans cette vidéo, une unité jaune est placée exactement à l'origine *World*, afin d'aider à la visualisation de l'origine du monde lors du recentrage du terrain au lancement de la partie. En mode édition, on voit que le terrain retourne à sa position initiale non centrée, ce qui illustre l'exécution adéquate du script à chaque nouvelle exécution du jeu. Aussi, un redimensionnement du terrain est effectué dans le vidéo, afin de démontrer que le script ajuste la position adéquatement en fonction de ses dimensions dynamiques.

Objectif : Création du graphe de scène des unités

Description :
Cet objectif incorpore plusieurs étapes, soit l'importation des modèles 3D dans *Unity*, l'édition de la structure relationnelle entre les parties des modèles et le positionnement adéquat des diverses parties l'une par rapport à l'autre. De plus, les divers paramètres requis pour le mouvement et la rotation des différentes parties des modèles doivent être assignés logiquement pour effectuer des déplacements naturels en fonction du type d'unité représentée. Ces paramètres sont plus particulièrement exploités à l'objectif « *Animation du canon d'une unité* ».

Réalisation :
Tout d'abord, des modèles 3D de tanks ont été téléchargés de sources publiques. Ceux actuellement importés dans le projet proviennent des références [2, 3, 4, 5, 8, 9, 10]. La structure relationnelle des parties formant les unités a été respectée selon l'exemple de la Fig. 5. Aussi, un modèle de bulldozer [1] a été importé afin d'avoir une unité de construction dans le jeu. Ce dernier modèle emploie une structure relationnelle légèrement différente (pas de « *Turret* ») étant donné qu'il n'a pas de parties mobiles contrairement aux tanks, mais le reste de la structure est similaire.

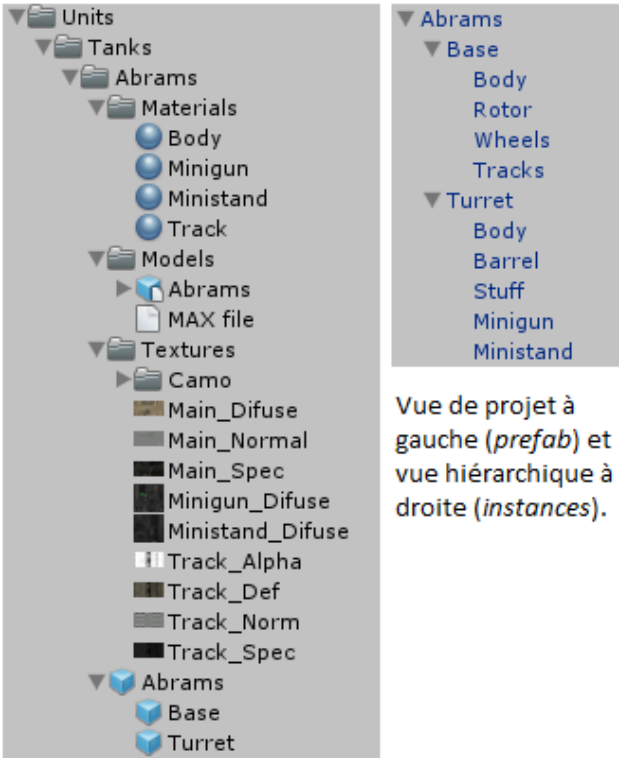

Fig. 5: Structure relationnelle des éléments composant une unité de combat

Dans la partie droite de la Fig. 5, on peut voir l'arbre de relation entre les différentes parties formant le modèle du tank « *Abrams* ». Lors de l'importation, chaque section était positionnée de manière absolue. Ainsi, pour faciliter le déplacement de toute l'unité d'un seul coup, un *GameObject* vide a été créé (*Abrams*) et toutes les pièces ont été placées relativement à l'intérieur. De manière similaire, les sections « *Base* » et « *Turret* » ont été créées avec des *GameObjects* vides afin de permettre un déplacement indépendant entre le haut et le bas du tank. Cela permet d'accomplir les rotations cohérentes comme souhaité à l'objectif « *Animation du canon d'une unité* ».

Dans la partie gauche de la Fig. 5, on peut voir les matériaux qui ont été affectés à chacune des parties. Ces matériaux font le pont entre chacun des sous-objets du modèle et les diverses combinaisons de textures sous forme d'images. Aussi, nous pouvons voir sous l'onglet « *Models* » les références aux fichiers originaux employés pour l'importation. La structure a été maintenue pour les 5 modèles importés. Aussi, suite à l'importation des modèles et de l'application de leur texture, les paramètres d'échelle ont été ajustés lorsque nécessaires pour avoir des unités de dimensions sensiblement égales.

Tous les modèles ont également été positionnés de sorte à avoir leur base faisant face dans la même direction et sens que l'axe Z. Dans certains cas, il a été nécessaire d'appliquer des rotations de 180º aux pièces pour respecter cette contrainte, mais cela permettra d'éviter des problèmes lorsque l'on voudra faire déplacer les unités dans une direction. Aussi, étant donné que les relations indépendantes ont été appliquées pour diverses sections des modèles, il est possible d'obtenir divers types de configurations de rotation des sections selon les paramètres appliqués aux modèles.

La première configuration désirée est présentée à la Fig. 6. Celle-ci permet de tourner le haut du tank (*Turret*) indépendamment de la base, mais le bout du canon (*Barrel*) est fixé à une hauteur constante. La seconde configuration présentée à la Fig. 7 permet à l'unité d'ajuster l'angle du « *Barrel* », en plus du déplacement du « *Turret* » comme la configuration précédente. Ainsi, cette configuration permet à l'unité de pointer vers un ennemi situé en altitude. Cela sera notamment employé pour les unités à fonction antiaérienne. Finalement, la troisième configuration actuellement employée est appliquée pour des unités dont la base et le « *Turret* » ne peuvent pas être tournés indépendamment. Dans le cas particulier de l'unité présentée à la Fig. 8, celle-ci attaque uniquement à grande distance. Ainsi, son canon doit pointer vers le haut à un angle précis pour former un arc d'attaque. Afin de permettre la rotation adéquate du « *Barrel* », un *GameObject* a été employé comme point de pivot. L'ensemble des configurations mentionnées seront établies en fonction des paramètres inscrits en entrée avec les champs du script « *TankManager.cs* » associé à chacun des *prefabs* correspondant à un tank.

Fig. 6: Déplacements possibles d'une unité de combat avec la configuration 1

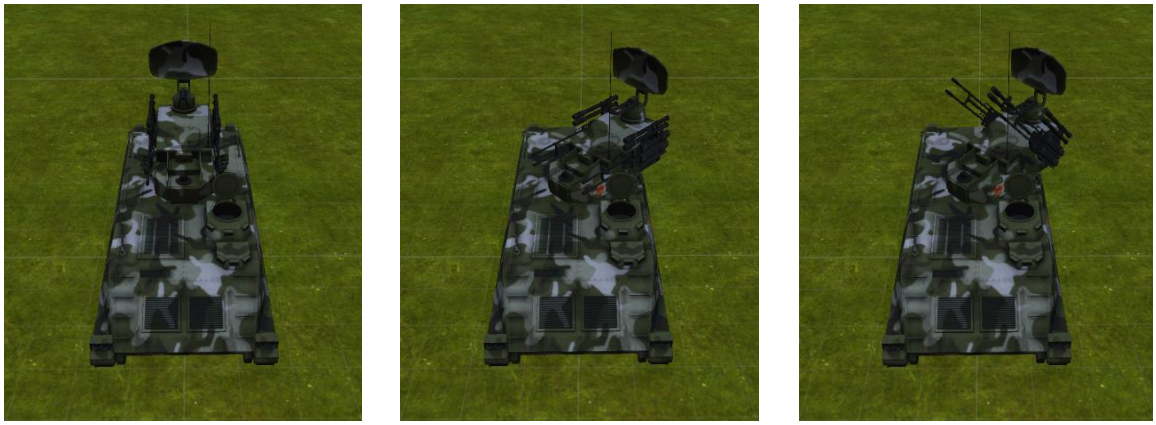

Fig. 7: Déplacements possibles d'une unité de combat avec la configuration 2

Fig. 8: Déplacements possibles d'une unité de combat avec la configuration 3

La Fig. 9 suivante présente les configurations attendues des modèles courants. Le modèle « *Bulldozer* » n'a pas de configuration puisqu'il s'agit de l'unité de construction.

Fig. 9: Configurations attendues pour les modèles importés

<u>Objectif :</u> Application des textures aux modèles

<u>Description :</u>
Cet objectif consiste tout simplement à l'application des multiples textures aux modèles selon les images source disponibles lors de leur téléchargement. Les textures doivent être appliquées adéquatement selon leur type (*normal map, diffuse, specular,* etc.). L'édition de quelques teintes est aussi requise pour que les modèles aient l'apparence voulue.

Réalisation :

Pour les unités représentant des tanks ou le bulldozer, les textures ont été appliquées en fonction des images disponibles. Étant donné que les images proviennent de la même source que les modèles 3D, le *mapping* entre les texels et les objets était déjà adéquatement réalisé, donc aucun ajustement n'a été nécessaire sur ce point. Dans la majorité des cas, il y avait deux textures, soit une diffuse pour la couleur générale de l'unité et une texture de détail pour les effets de réflexion de lumière spécifiques sur certaines parties comme les bordures et les coins des objets. Des *Shaders* « *Diffuse Detail* » ont donc été employés pour chaque « *Material* » correspondant aux multiples parties du modèle afin de pouvoir faire le *drag-and-drop* des textures dans les cases de référence adéquates. Cela est présenté à la Fig. 10. Puis, ces matériaux ont été à leur tour glissés sur chacune des pièces formant les unités, selon les liaisons logiques applicables. Par exemple, une texture était disponible pour le « corps » de l'unité, alors qu'une autre correspondait aux chenilles du tank.

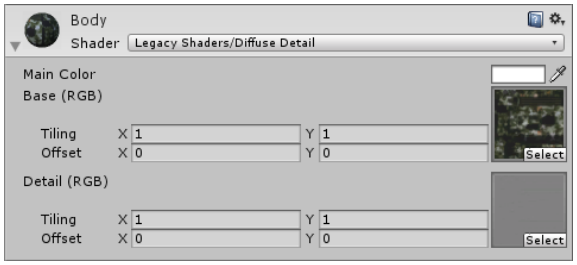

Fig. 10: Matériau de base avec textures et *Shader* simple pour quelques unités

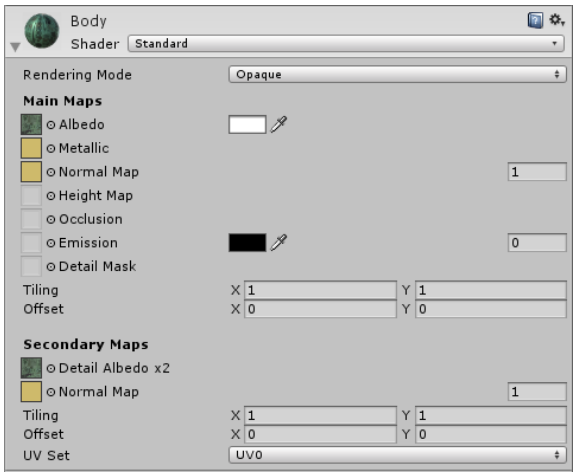

Fig. 11: Matériau avec textures de détail et *shaders* adéquat des unités « *Diplomat* » et « *Statesman* »

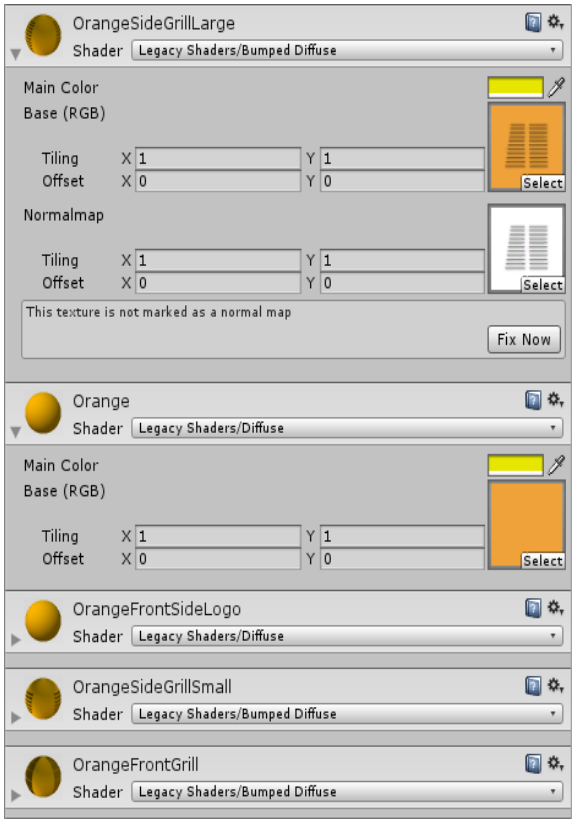

Fig. 12: Ensemble de matériaux avec textures et *shaders* pour certaines parties du bulldozer

Dans certains cas, il arrivait que beaucoup plus que deux textures soient disponibles, comme dans le cas du bulldozer où chaque pièce jusqu'au dernier boulon était composée d'un matériau. Il arrivait aussi que certaines parties, notamment pour le « corps » principal du bulldozer, qu'un ensemble de textures était requis, comme on le voit à la Fig. 12. Ainsi, un peu plus d'opérations ont dû être accomplies dans le cas de ce modèle. Aussi, quelques textures semblaient trop pâles originalement, donc le paramètre « *main color* » des matériaux correspondants ont été ajustées pour obtenir l'effet de couleur désiré.

De plus, les unités « *Diplomat* » et « *Statesman* » avaient légèrement plus de textures « *detail* » disponibles que pour les autres modèles. Donc, un *Shader* de type « *standard* » a été utilisé pour appliquer les textures « *Albedo* », « *Metallic* », « *Normal Map* » (primaire et secondaire) et « *Detail Albedo x2* » (voir Fig. 11).

Dans le cas du bâtiment, un *Shader* spécialisé appelé « *ZGON : Bumped Detail Spec 01* » (voir la Fig. 13) a été employé parce que celui-ci est fourni avec les modèles importés pour réaliser le modèle du bâtiment. Le résultat d'application de ce *Shader* est présenté à la Fig. 21 de la section « *Importer et implémenter les bâtiments* ».

Le *Shader* spécialisé accomplit essentiellement la tâche de combinaison d'une grande quantité de textures et couleurs selon un paramétrage très versatile et spécifique. Celui-ci permet aussi de combiner les différentes sections des morceaux de bâtiments, comme le ciment, la vitre des fenêtres ou les bordures colorées.

En important les modèles des blocs formant le bâtiment, le *Shader* a été appelé automatiquement avec un fichier *XML* pour générer tous les matériaux nécessaires, comme celui donné en exemple à la Fig. 13. Donc, il a suffi de glisser les matériaux générés sur les bonnes pièces pour appliquer les textures aux modèles.

Les figures suivantes présentent quant à elles les unités avant et après l'application des textures qui ont été précédemment présentées.

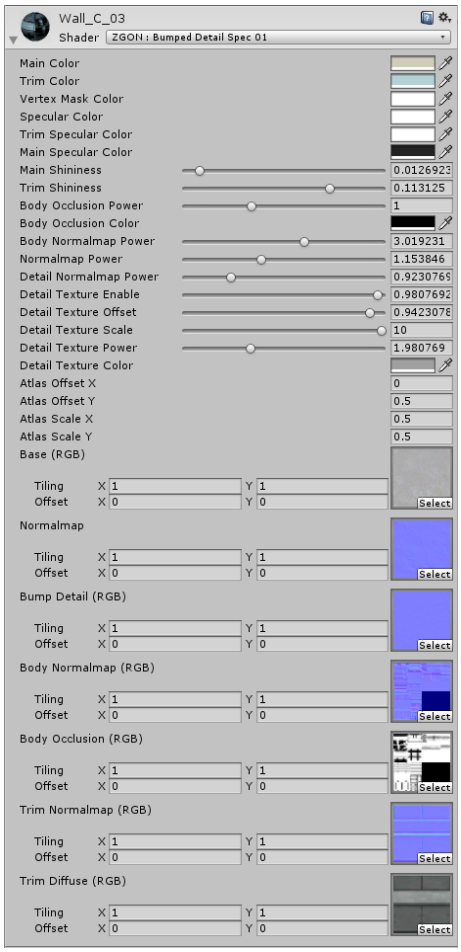

Fig. 13: Matériau avec *Shader* du bâtiment

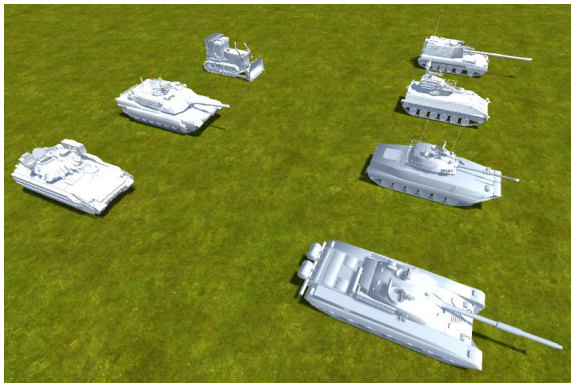
Fig. 14: Modèles avant l'application des textures


Fig. 15: Modèles après l'application des textures

Objectif : Animation du canon d'une unité

Description :
Cet objectif consiste à reprendre les paramètres spécifiés par le script « *TankManager.cs* » associés aux *prefabs* modélisés à l'objectif « *Création du graphe de scène des unités* » pour réaliser les transformations géométriques accomplissant les mouvements attendus.

Lorsqu'une unité est sélectionnée et que l'ordre d'attaquer une unité ennemie est donné par un clic de la souris, le canon de l'unité sélectionnée devra tourner en direction de l'unité visée pour accomplir l'attaque. La rotation du canon devra se faire en transition plutôt que d'un seul coup. Aussi, si l'unité ennemie à attaquer est trop éloignée ou si aucun ordre d'attaque n'est donné, le canon devra pointer vers le devant de l'unité. Ainsi, suite à une attaque complétée ou annulée, le canon devra retourner à l'état pointant vers l'avant. Plusieurs types de rotations devront être supportées, tout dépendamment de si l'unité est antiaérienne ou non, et selon si la tourelle peut être tournée indépendamment de la partie inférieure du tank.

Réalisation :
Afin de produire les rotations désirées, les paramètres présentés à la Fig. 16 sont employés. Lorsqu'un *GameObject* est assigné au champ « *Aiming Target* », le tank associé accomplit le mouvement requis pour « attaquer » cette unité ennemie. Afin de déterminer quelles sections déplacer et afin de s'assurer de préserver les références dans le cas où un modèle serait modifié ou qu'une pièce serait renommée, les paramètres « *Cannon Turret* » et « *Cannon Barrel* » doivent tous deux être indiqués. Le champ « *Cannon Rotate Speed* » permet de contrôler la rotation autour de Y de la partie supérieure d'un tank, donc une valeur de zéro permet d'accomplir la configuration #3, alors qu'une valeur supérieure à zéro indique la vitesse de déplacement pour les configurations #1 et #2. De manière similaire, le champ « *Barrel Rotate Speed* » permet d'indiquer la vitesse de rotation autour de X, donc la configuration #1 requiert zéro, tandis que les configurations #2 et #3 ont des valeurs supérieures à zéro. Le paramètre « *Barrel Lock On Target* » permet de spécifier s'il faut viser vers l'ennemi ou si un angle précis autour de X est nécessaire. Ainsi, le paramètre « *Barrel Attack Angle* » doit aussi être ajusté adéquatement selon cette valeur. Le champ « *Code* » est quant à lui seulement un *string* correspondant au type de tank.

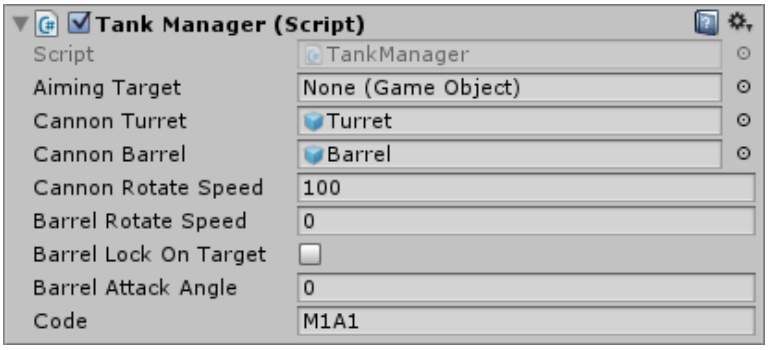

Fig. 16: Paramètres du script « *TankManager.cs* »

Pour accomplir les rotations, les fonctions « *LookRotation* » et « *RotateToward* » sont principalement employées avec « *Time.deltaTime* » pour réaliser l'effet de transition des pièces. Dans le cas spécifique de la configuration #3 nécessitant un angle précis, les intervalles de rotations sont plutôt calculés et appliqués selon les axes relatifs du tank, du « *Turret* » ou du « *Barrel* » selon le cas requis. Les différentes valeurs de vitesses de rotation appliquées par différent type de tank permettent d'avoir des effets de transition personnalisés pour des unités plus rapides ou plus lentes. L'ensemble des mouvements réalisés sont toutefois accomplis à vitesse constante (aucune accélération ou décélération). Afin de tester le script « *TankManager.cs* », des boutons « *Attack Target* » et « *Stop Attack Target* » ont été placés sur le *canvas* pour respectivement appliquer un *GameObject* cubique comme « *Aiming Target* » à chacun des 5 tanks dans la scène ou remettre les champs à *null* pour arrêter l'attaque. La vidéo « *cannon-animation.mp4* » montre la mise en application du script.

Objectif : Sélection d'unités et commande d'attaque

Description :
La sélection des unités devra être rendue possible à l'aide de clics de souris sur celles-ci et les unités sélectionnées pourront recevoir des commandes d'attaque envers d'autres unités. Les commandes d'attaque pourront aussi être annulées en effectuant un clic sur le terrain. L'état de sélection actuel des unités devra être affiché à l'aide d'ellipses correspondantes sur le sol et un clic sur le terrain désélectionnera toutes les unités sélectionnées. Les boutons de souris désirés pour exécuter les commandes devront aussi être spécifiés comme paramètres d'entrée afin de permettre une modification facile de la configuration par le joueur ultérieurement. La sélection des unités devra permettre tant la sélection unique ou multiple des unités, selon si un bouton additionnel du clavier est appuyé ou non. Ce dernier devra aussi supporter une configuration selon un paramètre d'entrée. Lorsque plusieurs unités sont sélectionnées, elles recevront toutes la commande pour lancer ou arrêter l'attaque, selon le cas demandé avec les clics de souris. Finalement, les commandes d'attaque devront être acceptées seulement si l'unité à attaquer respecte les distances minimale et maximale de portée (voir la Fig. 46 de la section « *Projectile et effet d'impact* ») et si les conditions logiques d'attaque sont respectées (unités de sol ou aériennes pouvant attaquer celles situées au sol ou en l'air).

Réalisation :
La détection des clics de souris est maintenant supportée plutôt que d'employer les boutons de test sur le *canvas*. Pour ce faire, un script appelé « *SelectorManager.cs* » a été créé et est attaché à la caméra *RTS* de sorte à accepter les clics en tout temps.

Par défaut, un clic de souris gauche sur une unité la sélectionne. L'état de sélection de cette unité est alors affiché par une ellipse, tel que présenté à la Fig. 19. Aussi, la sélection multiple est supportée par défaut avec la touche « *Left CTRL* », et si l'unité est déjà sélectionnée dans ce cas, elle est plutôt désélectionnée. Les clics de souris droite sont actuellement employés pour les commandes d'attaque, mais l'ensemble des commandes sont modifiables en entrée du script « *SelectorManager.cs* » (Fig. 18). Le respect des distances de portée et des conditions d'attaque est quant à lui géré par le script

« *UnitManager.cs* » affecté à chacune des unités existantes. La figure Fig. 17 montre les divers paramètres disponibles avec ce script. Ainsi, lorsque des clics sont effectués, le script « *SelectorManager.cs* » gère les conditions et, si des unités sont effectivement impliquées avec les clics détectés, les instructions sont déléguées adéquatement aux scripts « *UnitManager.cs* » des unités correspondantes pour la gestion des conditions d'attaque. Le script « *UnitManager.cs* » effectue quant à lui une délégation subséquente de tâche au script « *TankManager.cs* » si nécessaire afin d'exécuter les animations de canon requises.

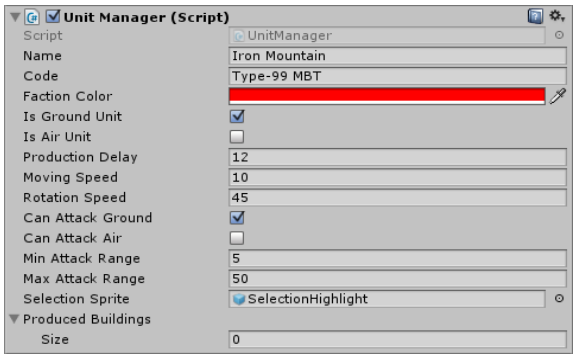

Fig. 17: Paramètres du script « *UnitManager.cs* »

Fig. 19: Affichage de l'état sélectionné d'une unité

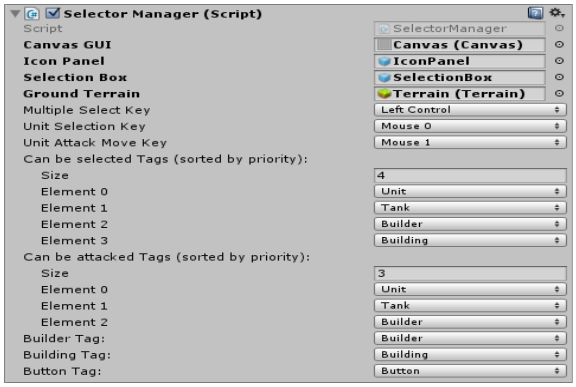

Fig. 18: Paramètres du script « *SelectorManager.cs* »

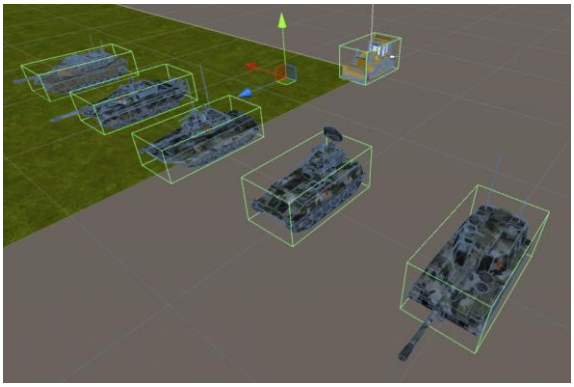

Fig. 20: Objets *BoxCollider* appliqués aux unités

L'état de sélection des unités est réalisé à l'aide d'un *Sprite* correspondant à une bordure de cercle blanc avec un centre transparent. Ce cercle est ensuite attaché à chaque unité et est étiré (*scaling*) selon les besoins respectifs des unités. La couleur est ajustée selon la valeur inscrite dans « *Faction Color* » de « *UnitManager.cs* ». La visibilité est ensuite gérée selon si l'unité est sélectionnée ou non.

Pour accomplir la sélection et l'attaque d'unités, des *Raycasts* sont employés en combinaison avec les *Tags* affectés aux unités qui indiquent si les commandes sont applicables. Les *Raycasts* sont lancés à partir de la position 2D de la souris convertie dans l'espace 3D. Puis, des objets *BoxCollider* appliqués sur les unités (boîtes à la Fig. 20) permettent de déterminer si un *Raycast* la « frappe », ce qui indique si un clic est effectué sur l'unité. Cette méthode est inspirée de plusieurs tutoriels tel que démontré dans [18]. Chaque type de clic est géré indépendamment par différents *Raycasts* émis avec « *SelectorManager.cs* ». Chaque instance d'unité sélectionnée est mémorisée dans une liste pour gérer la sélection multiple et ainsi transférer à chacune les demandes d'attaque.

Les paramètres « *Select Tags* » et « *Attack Tags* » du script « *SelectorManager.cs* » permettent de filtrer le type d'objets acceptés pour ces commandes respectives, ce qui assure une gestion plus étroite des cas désirés. Par exemple, en changeant le *Tag* d'une unité particulière pour une valeur non existante dans « *Attack Tags* », il serait possible de rendre cette unité invulnérable aux commandes d'attaque.

Les fonctionnalités mentionnées pour cet objectif sont démontrées dans la vidéo « *unit-selection-attack-command.mp4* ». Plus particulièrement, on y voit la sélection unique, la sélection multiple, la désélection et les commandes d'attaque ou d'arrêt d'attaque. Aussi, les commandes d'attaque sont plus particulièrement présentées pour démontrer le respect des conditions avec deux cubes dans les airs, l'un étant trop proche pour être attaqué et l'autre étant dans la portée. De plus, une sphère ayant un *Tag* non supporté pour l'attaque, mais étant autrement bien positionnée pour être attaquée, est disponible dans la scène pour démontrer l'effet du filtrage par *Tags*. Pour chacun des cas visionnés, un suivi de déverminage est présenté dans la console pour indiquer les commandes captées et appliquées selon les divers clics effectués.

Objectif : Importer et implémenter les bâtiments

Description :
Les modèles 3D représentant les différents bâtiments permettant d'accomplir diverses fonctions dans le jeu devront être importés afin de permettre leur affichage sur le terrain. Les modèles importés dans la partie devront contenir plusieurs informations spécialisées permettant de les distinguer selon leur fonctionnalité. Quelques-unes des informations seraient entre autres leur faction, les unités disponibles pour la création et la chaîne de production d'unités actuelle. Chacune de ces informations sera gérée selon le type de bâtiment. Aussi, une petite animation sera nécessaire lorsqu'une unité aura terminé sa production pour la faire sortir du bâtiment.

Réalisation :
Puisque différents types de bâtiments auraient différentes fonctionnalités, un grand nombre d'objectifs supplémentaires seraient nécessaires. Ainsi, la modélisation a plutôt été limitée à un seul bâtiment. Pour fabriquer ce bâtiment, des modèles 3D tirés de l'*Asset Store* ont été employés. Les blocs présentés à la Fig. 21 proviennent de [35] et permettent un assemblage quelconque pour former un bâtiment ayant la forme désirée. Ceux-ci ont donc été importés dans *Unity* et les diverses textures requises leur ont été appliquées, comme il l'est décrit à l'objectif « *Application des textures aux modèles* ».

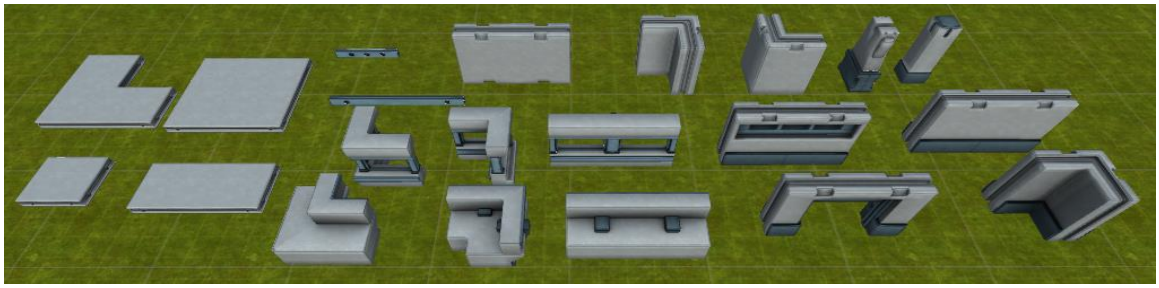

Fig. 21: Pièces employées pour assembler le bâtiment

Les pièces sont utilisées pour créer le bâtiment de la Fig. 22, appelé « *WarFactory* » dans le projet. Une porte de garage a également été empruntée de [24] afin de pouvoir implémenter l'animation d'une porte qui s'ouvre lorsqu'une unité termine sa production pour la faire sortir du bâtiment. Deux prismes de couleur noire ont aussi été placés à l'intérieur du bâtiment afin qu'il ne soit pas possible de voir à l'intérieur à cause des fenêtres transparentes. Afin de modéliser ce bâtiment, il a simplement été question de positionner les blocs adéquatement les uns par rapport aux autres pour obtenir l'apparence voulue. L'architecture du graphe de scène de ce bâtiment est présentée sommairement à la Fig. 23, où plusieurs des blocs de la Fig. 21 sont répétés une multitude de fois.

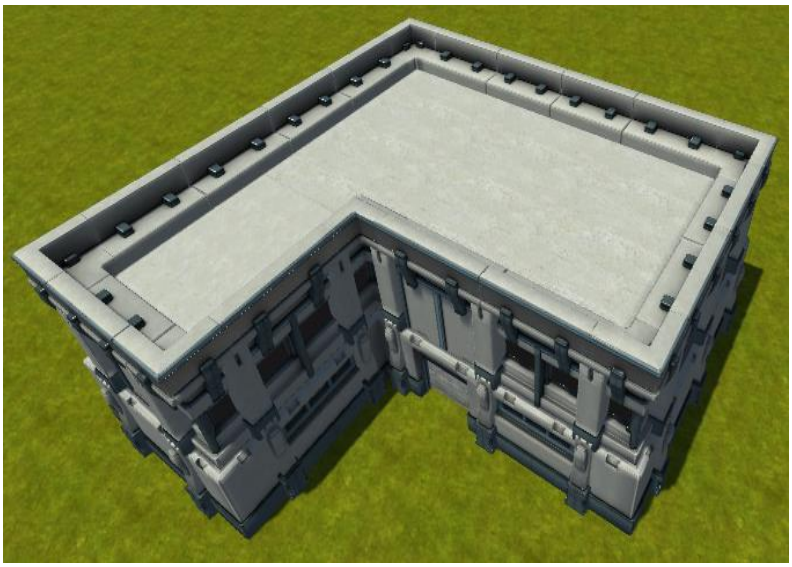

Fig. 22: Bâtiment « *WarFactory* » produisant les unités de combat

Note :
Pour les besoins d'affichage, seuls les modèles de la sous-section « *RightSide* » du bâtiment sont présentés avec la Fig. 23, mais il est facile de voir que chacune de ces sections contient des sous-blocs répétés formant chaque mur, toit et coin du bâtiment.

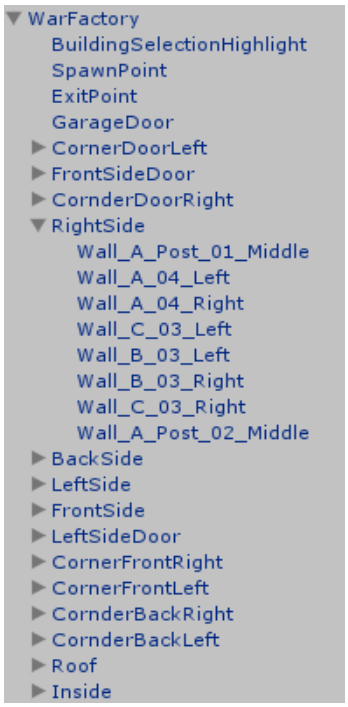

Fig. 23: Graphe de scène du bâtiment « *WarFactory* »

Ensuite, un script nommé « *BuildingManager.cs* » permettant la gestion des informations du bâtiment a été créé et attaché au modèle « *WarFactory* ». Celui-ci permet plus particulièrement de contenir les données générales, comme le nom du bâtiment, ainsi que les références importantes, comme « *SpawnPoint* » et « *ExitPoint* » qui indiquent où l'unité produite est instanciée et où elle doit se diriger pour quitter le bâtiment.

Aussi, les paramètres par rapport à l'ouverture et la fermeture de la porte sont indiqués dans le script « *BuildingManager.cs* », afin de permettre l'animation de celle-ci quand l'unité finit d'être produite suite à son délai de production. Une liste « *Produced Units* » contenant les unités disponibles pour la production avec ce bâtiment est spécifiée dans le script. Le délai de production de chaque unité dans cette liste est ainsi extrait de leur référence « *UnitManager.cs* » respective lorsque des instances correspondantes sont demandées. Il est également possible de contrôler la taille de la file de production avec « *Production Queue Size* ». Tous ces paramètres sont présentés à la Fig. 24.

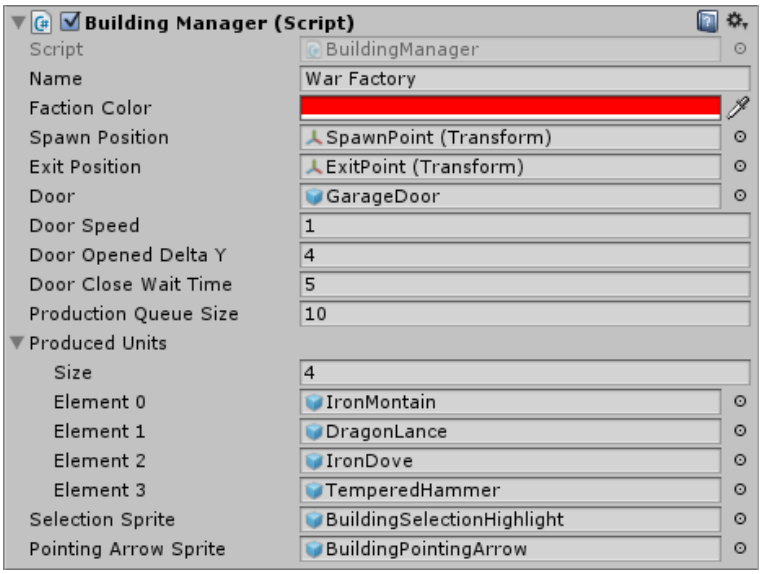

Fig. 24: Interface du script « *BuildingManager.cs* » pour le bâtiment « *WarFactory* »

Le script « *BuildingManager.cs* » contient aussi toutes les fonctions nécessaires à l'implémentation de l'animation de la porte, l'attente des délais de production des unités ainsi que de son déplacement allant de « *SpawnPoint* » vers le point de sortie « *ExitPoint* ». Aussi, le script contient plusieurs méthodes qui effectuent la gestion de la file employée pour maintenir les unités en attente de production suite à des demandes de l'utilisateur. Cette file permet de produire les unités dans le bon ordre, de limiter le nombre d'unités produites simultanément, ainsi que de garder les références d'ici à ce que leur délai de production s'achève séquentiellement.

Afin d'accomplir la gestion des temps de production, plusieurs *Coroutines* faisant appel aux fonctions « *WaitForSeconds* » et « *WaitUntil* » sont employées. Pour réaliser l'animation faisant sortir l'unité du bâtiment, une translation linéaire partant de la position d'où l'instance de l'unité est générée vers le point à l'extérieur du bâtiment est employée. La méthode « *Vector3.MoveToward* » est particulièrement utilisée dans une boucle *while* contenue dans une *Coroutine* afin d'y parvenir.

Une vidéo intitulée « *building-production-queue.mp4* » est disponible pour démontrer les diverses fonctionnalités de cet objectif. On y voit la gestion de la porte s'ouvrant et se fermant à la production de chaque unité et le déplacement de l'unité vers le point de sortie. Aussi, afin de démontrer la gestion de la file de production et des délais de chaque unité, des informations de déverminage sont affichées pour montrer tant le contenu de la file, le temps restant pour produire une unité ou le temps restant pour gérer le déplacement de la porte selon le cas. Les unités ont aussi été affectées avec des temps de production de 3 à 6 secondes respectivement selon l'ordre affiché sur l'interface, simplement pour montrer l'effet variable du délai de production. Finalement, la vidéo montre des demandes d'ajout de nouvelles unités à la file de production alors qu'une autre action est en cours (mouvement de porte par exemple), afin de démontrer la gestion en parallèle de la mise à jour des multiples fonctions et mouvements à l'aide des *Coroutines*.

Objectif : Rotation et positionnement de bâtiments

Description :
Une fois que les modèles de bâtiments seront créés, ceux-ci devront supporter un positionnement fixe sur le terrain de jeu sans le traverser. Aussi, le positionnement des bâtiments devra s'assurer que ceux-ci n'entrent pas en collision avec d'autres éléments déjà positionnés sur la carte. Si c'est le cas, ils devront s'afficher avec une teinte rougeâtre pour indiquer l'erreur d'emplacement et le positionnement ne sera alors pas permis. Avant que le positionnement des bâtiments ne soit accepté par l'appui du bouton gauche de la souris, ceux-ci devront permettre une rotation autour de l'axe normal au terrain. Pour ce faire, la gestion du « *Right-Click* » maintenu enfoncé avec les déplacements de souris devra être accomplie sans interférer avec le mouvement de la caméra qui emploie également cette méthode de rotation. Une flèche sera également affichée pour pointer vers la direction de rotation actuelle lors de l'application de ces rotations. Une touche de clavier devra être assignée pour accomplir le même genre de rotation des bâtiments, mais selon des incréments d'angles fixes et spécifiés comme paramètre d'entrée. Une touche additionnelle doit aussi être prévue pour annuler le positionnement. Finalement, les bâtiments en cours de positionnement devront être affichés avec une teinte verdâtre pour bien démontrer qu'ils ne sont pas encore placés, mais que la position actuelle est valide, c'est-à-dire lorsqu'elle n'est pas en collision avec un autre élément sur le terrain.

Réalisation :
Afin de réaliser la gestion du positionnement du bâtiment, des tutoriels [13, 17] ont été consultés à titre de référence. Un script nommé « *BuildingPlacementManager.cs* » a ensuite été réalisé afin de contenir toutes les méthodes qui réalisent les fonctionnalités de cet objectif. Ce script est appliqué au modèle du bâtiment « *WarFactory* » pour permettre la création d'une instance de ce bâtiment spécifiquement. Dans ce script, une fonction appelée « *RequestNewBuildingPlacement* » est disponible et est employée pour initialiser plusieurs paramètres gérant les conditions de placement du bâtiment, pour activer un drapeau indiquant qu'un bâtiment est en état de placement actuellement, et générer la nouvelle instance du bâtiment à placer sur le terrain. En initialisant les paramètres seulement lorsqu'applicables pendant le positionnement d'un nouveau bâtiment avec cette fonction, on s'assure de ne pas faire plusieurs appels « *Update* » inutiles à tout autre moment dans le jeu. La méthode mentionnée peut être appelée de l'extérieur de la classe, comme avec des évènements de boutons sur l'interface. Cela sera discuté plus en profondeur à l'objectif « *Interface pour choisir les bâtiments* ».

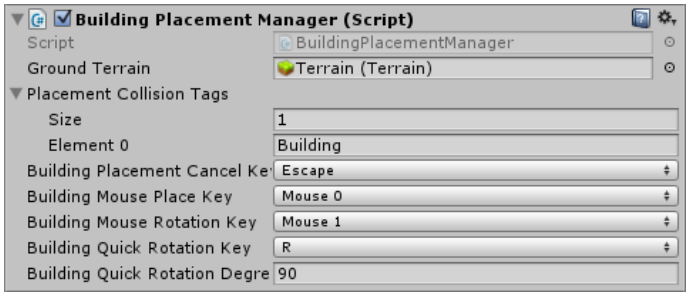

Fig. 25: Paramètres du script « *BuildingPlacementManager.cs* »

La Fig. 25 présente les divers paramètres disponibles avec le script, ceux-ci sont principalement la configuration des touches à reconnaître pour appliquer les multiples fonctionnalités. Aussi, une référence au terrain est spécifiée, afin de permettre le positionnement du bâtiment à plat sur celui-ci sans le traverser. Finalement, une liste « *Placement Collision Tags* » est disponible pour indiquer quels *Tags* considérer comme une collision lors du placement du bâtiment.

Lorsque le drapeau d'état de placement est activé avec « *RequestNewBuildingPlacement* », la capture des touches de clavier et clics de souris débute et se poursuit à chaque nouvelle mise à jour jusqu'à ce que le bâtiment soit placé ou que l'opération soit annulée. L'état de placement activé au début permet de désactiver temporairement la fonction de rotation de la caméra, ce qui l'empêche de tourner lorsque l'on applique les rotations de bâtiment. La fonction de rotation de la caméra est rétablie lorsque la gestion du placement termine.

Selon les entrées captées par « *GetKey* » et « *GetKeyUp* » lorsque l'état de placement est activé, les actions adéquates sont effectuées. Plus particulièrement selon les paramètres spécifiés à la Fig. 25, les opérations suivantes sont accomplies :

- Si la touche « R » est appuyée, des rotations de 90 degrés dans le sens horaire par rapport à la normale du terrain sont effectuées au bâtiment en multipliant son vecteur de rotation « *Up* » par 90 et en utilisant « *transform.Rotate* ».
- Pour une touche « *Escape* », le placement se termine en gérant les divers paramètres de contrôle et l'instance du bâtiment est détruite.
- Si un bouton droit de souris est maintenu enfoncé, le bâtiment cesse toute translation et accomplit les rotations en pointant dans la direction de la souris. Cela est accompli à l'aide de la fonction « *LookAt* ».
- Si un bouton gauche de souris est effectué, le placement définitif du bâtiment est accompli si aucune collision n'est détectée à cette position.
- Si aucune des touches précédentes n'est détectée, la position actuelle du bâtiment est mise à jour continuellement selon la projection dans l'espace 3D de la position du curseur de la souris sur le terrain.

Pour obtenir la position de la souris sur le terrain, un *Raycast* est employé et une vérification du *Tag* du terrain est effectuée pour déterminer la distance caméra-terrain. Puis, la position correspondante à l'endroit où le *Raycast* a atteint le terrain est calculée avec cette distance pour l'appliquer au bâtiment.

Afin de gérer les collisions avec les objets placés dans la scène, des objets *BoxCollider* sont employés en combinaison avec des objets *Rigidbody*. Le *BoxCollider* du bâtiment « *WarFactory* » est spécifiquement présenté à la Fig. 27. L'objet *Rigidbody* attaché au bâtiment est quant à lui en mode *« isKinematic* » afin que l'engin de physique ne gère pas les impacts et déplacements du bâtiment, ce qui poserait problème puisque l'on veut ici imposer la position de la souris nous-mêmes. Le *Rigidbody* est nécessaire puisque, sans lui, la détection de collisions ne lançait aucun évènement. Les collisions sont alors détectées grâce aux évènements « *OnTriggerEnter* » et « *OnTriggerExit* », mais le *BoxCollider* est placé en mode « *isTrigger* » seulement à l'activation de l'état de

placement, ce qui évite les évènements inutiles lorsqu'ils ne sont pas nécessaires. Chaque évènement lancé incrémente ou décrémente respectivement un compteur de collisions. Ce dernier permet de gérer des conditions de collision avec plusieurs objets sur le terrain simultanément. Aussi, ce compteur est seulement ajusté avec les *Tags* contenus dans « *Placement Collision Tags* », ce qui permet par exemple d'accepter une position de placement du bâtiment au-dessus de certains objets comme des arbres, mais pas au-dessus des autres bâtiments. Par contre, peu importe les *Tags* indiqués, tout objet dont l'on désire permettre la détection de collision devra avoir un objet de type *BoxCollider*.

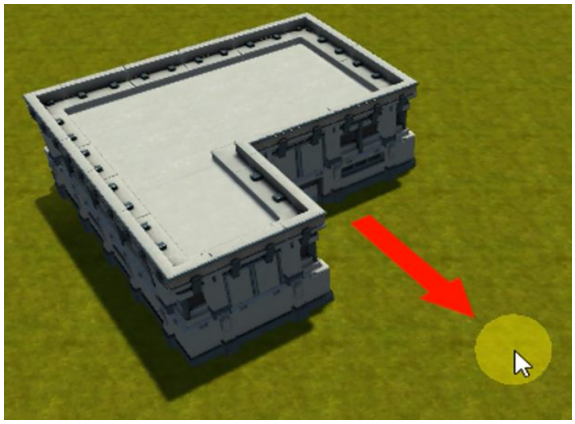

Fig. 26: Affichage d'une flèche lors de la rotation d'un bâtiment en placement

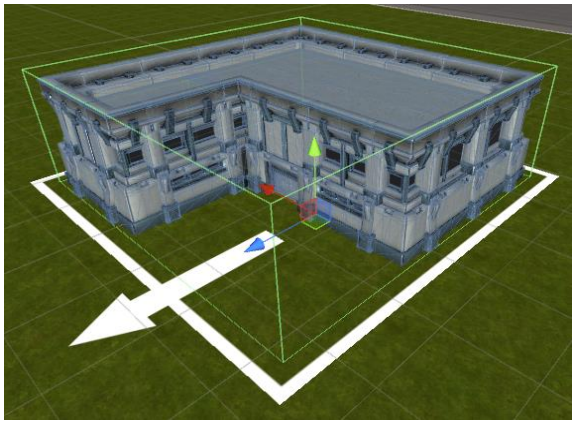

Fig. 27: Bâtiment « *WarFactory* » avec *BoxCollider*, sa flèche de rotation et son carré de sélection

Seulement lors de la rotation du bâtiment, la flèche présentée à la Fig. 26 est affichée avec la couleur de la faction attribuée au bâtiment. Cette flèche est un *Sprite* importé et est déjà pré-positionnée par rapport au point de pivot du bâtiment avant le début de la partie, comme montré à la Fig. 27. Son état d'affichage est simplement géré lorsque nécessaire.

Finalement, la couleur du bâtiment jusqu'à ce qu'il soit définitivement placé est ajustée selon les collisions détectées. Cela se fait en modifiant temporairement les valeurs des matériaux du modèle de manière similaire à ce qui est proposé dans [27]. Les deux états sont présentés avec la Fig. 28 et la Fig. 29.

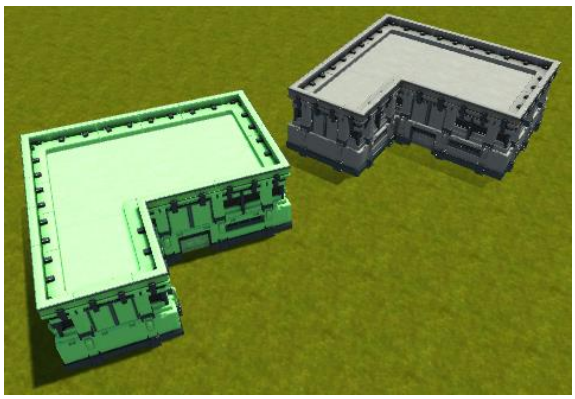

Fig. 28: Affichage d'une position valide sans collision d'un bâtiment en placement

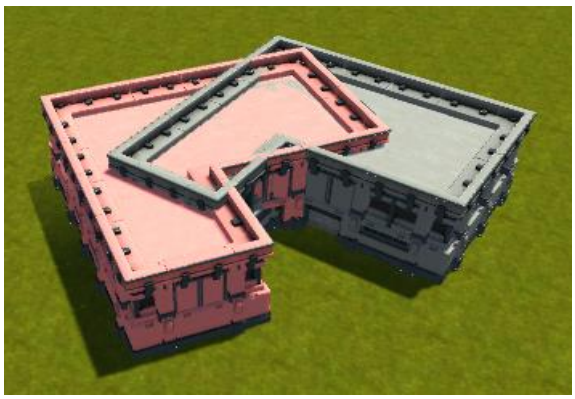

Fig. 29: Affichage d'une position invalide par collision d'un bâtiment en placement

Afin de pouvoir rétablir la couleur originale du bâtiment lorsqu'il est placé définitivement, une variable contient en tout temps un « multiplicateur de couleur ». Une position valide correspond à l'application d'un multiplicateur $(1,2,1)_{RGB}$ sur les composantes de couleur de tous les matériaux, alors que la position invalide utilise un multiplicateur $(2,1,1)_{RGB}$. Ainsi, à chaque nouvelle application d'une teinte désirée, le multiplicateur courant est inversé avec (1/R,1/G,1/B) afin de retourner aux teintes originales, puis le multiplicateur désiré est appliqué.

Cet objectif a nécessité notablement beaucoup d'analyses de cas très particuliers pour s'assurer qu'autant de conditions d'opération désirées que possible soient respectées en tout temps. Par exemple, l'utilisation du compteur de collision plutôt que d'employer une simple valeur booléenne résulte d'une analyse de cas où plusieurs bâtiments étaient simultanément en collision, car ils étaient assez proches les uns des autres pour le permettre. Sans le compteur, le positionnement devait valide quand une collision « quittait » l'une des zones d'un *BoxCollider* malgré que le bâtiment soit toujours à l'intérieur d'un autre *BoxCollider*. Cela se produisait étant donné que l'évènement « *OnTriggerExit* » avec été capté au moins une fois. Un autre exemple est l'ajustement de la position de la porte du bâtiment pour la production d'unités qui devait être corrigée à cause du positionnement dynamique apporté avec ces nouvelles fonctionnalités. Plusieurs détails d'implémentation sont spécifiés en commentaire dans le script « *BuildingPlacementManager.cs* » selon les ajustements accomplis, et ceux-ci ne sont pas tous répétés ici puisque l'objectif s'allongerait grandement. Aussi, certains cas sont extrêmement spécifiques ou difficiles à expliquer sans visualiser directement le code.

Pour finir, une vidéo nommée « *building-placement.mp4* » est disponible et permet de visualiser les fonctionnalités discutées à cette section. Quelques messages de déverminage de base sont affichés afin de démontrer les instants où des clics sont effectués et que le placement d'un bâtiment est refusé, car il est en collision.

Objectif : Interface pour choisir les bâtiments

Description :
Une interface comportant un menu de sélection des bâtiments disponibles pour leur positionnement dans la partie devra être réalisée. Des icônes des bâtiments en question seront affichées à l'intérieur de ce menu de sélection et lorsqu'elles seront appuyées, une instance de ce bâtiment sera instanciée pour son positionnement. Les icônes de bâtiments devront s'afficher sur l'interface lorsque l'unité de type « *Builder* » sera sélectionnée et disparaîtront suite au placement définitif du bâtiment.

Réalisation :
Tout d'abord, une simple image a été importée dans *Unity* et a été appliquée sur le *canvas* à l'aide de l'objet *RawImage* pour servir d'arrière-plan à l'interface usager. Ensuite, un objet appelé « *IconPanel* » qui contient un script « *Canvas Group* » et un script « *Grid Layout Group* » a été ajouté au *Canvas* afin de permettre l'ajout d'icônes de manière ordonnée et automatiquement bien disposée. Cela est inspiré d'une vidéo de tutoriel [14]. Les points d'ancrage du « *IconPanel* » ont été ajustés afin de respecter une position

convenue à l'intérieur de l'espace gris dans le bas de l'arrière-plan importé (Fig. 30). Les icônes ont été limitées à une seule rangée avec l'option « *Fixed Row Count* » et le redimensionnement du *Canvas* est priorisé pour la hauteur par rapport à la largeur. Cela permet aux icônes de respecter la position désirée malgré leur redimensionnement lorsque la fenêtre de jeu est ajustée selon la résolution d'écran. Aussi, la priorité en hauteur fait que ceux-ci préservent une taille qui respecte très finement la hauteur de l'espace gris disponible. Évidemment, s'il y a trop d'icônes à afficher et que la fenêtre est trop rétrécie horizontalement pour qu'ils puissent être contenus dans la région grise, ils la dépassent, mais ce cas ne survient pas à moins de déraisonnablement changer la taille de la fenêtre.

Puis, six objets ont été ajoutés comme enfants au « *IconPanel* » pour servir de boutons, comme présentés à la Fig. 30. Chacun de ces boutons est en fait une image vide qui possède le script de bouton effectuant la capture d'évènements de clics de souris avec « *OnClick* ». Ces images peuvent alors être référencées avec une image désirée pour l'affichage. Les images des unités et bâtiments sont actuellement conservées comme enfant à chacun de leur *prefab* créé dans le projet. Afin d'obtenir les références d'images à appliquer sur ces icônes, toutes les unités et le bâtiment ont été affectés d'un script « *IconManager.cs* ». Ce script a pour unique tâche pour l'instant de contenir la référence de l'icône, comme présenté en exemple à la Fig. 31. Ce script est donc employé pour avoir un point d'accès commun à l'icône, peu importe le type de *GameObject* sur lequel l'image doit être retrouvée, ce qui évite d'avoir à gérer plusieurs classes différentes. Aussi, ce script pourrait être utilisé ultérieurement pour appliquer des opérations standard sur les icônes sans nécessiter une restructuration de l'architecture du code.

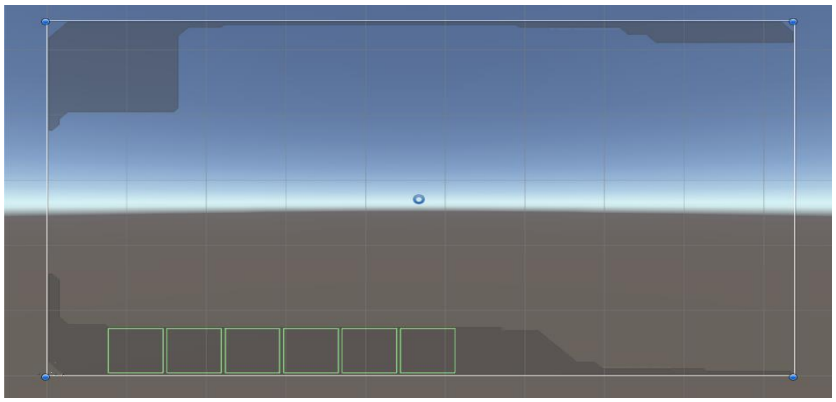

Fig. 30: Illustration de l'interface utilisateur actuelle avec les positions d'icônes mises en évidence

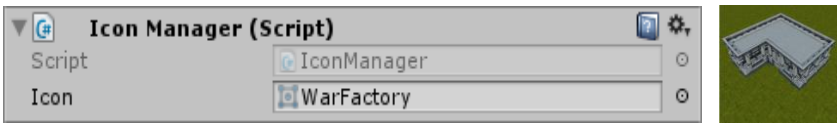

Fig. 31: Interface du « *IconManager.cs* » et de son image d'icône correspondante pour « *WarFactory* »

Donc, lorsque le script « *SelectorManager.cs* » accompli la sélection d'une unité de type « *Builder* », chacun des bâtiments référencés dans la liste « *Produced Building* » (Fig. 17) de son script « *UnitManager.cs* » peut être obtenu. Chacun de ces bâtiments donne alors accès à leur instance du script « *IconManager.cs* » pour appliquer les images correspondantes sur les boutons.

La désélection de l'unité de construction fait quant à elle disparaître les images des icônes et ceux-ci sont également désactivés pour ne plus capter d'évènements « *OnClick* ». L'état d'affichage et d'interaction des boutons est géré dans le script « *SelectorManager.cs* » appliqué à la caméra *RTS*. L'évènement « *OnClick* » de chacun des boutons sur l'interface est associé à la fonction « *ClickButtonPanel* » aussi située dans « *SelectorManager.cs* », comme montré à la Fig. 32. Cette fonction effectue la gestion des appuis de boutons étant donné que ceux-ci peuvent être employés pour d'autres fonctionnalités que la création de bâtiments (ex. : création d'unités par un bâtiment ou d'autres fonctionnalités futures). Ainsi, lorsque l'appui d'un bouton correspond effectivement à la condition représentée par cet objectif, soit quand un « *Builder* » est sélectionné, la fonction « *ClickButtonPanel* » délègue sont appel vers la fonction « *RequestNewBuildingPlacement* » contenue dans « *BuildingPlacementManager.cs* ». Ainsi, la procédure de l'objectif « *Rotation et positionnement de bâtiments* » prend la relève à partir de ce dernier appel.

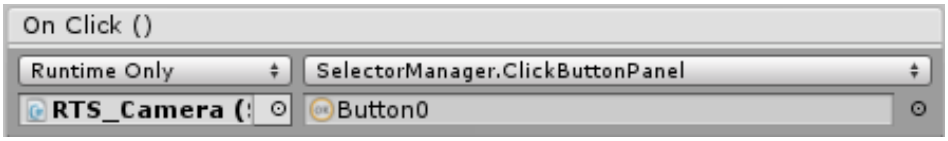

Fig. 32: Liaison de « *OnClick* » des icônes avec la fonction « *ClickButtonPanel* »

La mise en application de l'affichage des icônes avec sélection d'une unité de construction peut être observée avec la vidéo « *building-placement.mp4* ». Leur redimensionnement selon les dimensions d'écran est démontré dans « *icon-adjustment-screen-resize.mp4* ».

Objectif : Sélection de bâtiments existants

Description :
Lorsqu'un clic gauche de la souris sera effectué sur un bâtiment existant sur le terrain, celui-ci devra afficher qu'il est sélectionné à l'aide d'un « carré » l'entourant sur le sol. De plus, le bâtiment sélectionné devra afficher les actions possibles avec celui-ci dans l'interface utilisateur de la Fig. 30. Les actions disponibles seront principalement des créations d'unités de combats, mais pourraient être étendues plus tard à des recherches d'améliorations (*research upgrades*), à l'activation de propriétés propres au bâtiment ou même à la destruction du bâtiment. Chacune de ces actions devra être représentée par une icône particulière sur l'interface. Lorsqu'un bâtiment est sélectionné et que le bouton de la souris est appuyé en survolant une région vide du terrain, le bâtiment sera désélectionné. La désélection fera que le carré autour du bâtiment disparaîtra et ses icônes d'actions ne seraient plus affichées sur l'interface utilisateur.

Réalisation :
Pour accomplir la sélection de bâtiments, les *Raycasts* dans « *SelectorManager.cs* » ont été employées tout comme pour le cas des unités à l'objectif « *Sélection d'unités et commande d'attaque* ». Un *Sprite* a également été employé à nouveau, mais un carré est utilisé plutôt qu'un cercle. Le *Sprite* carré est ensuite appliqué au bâtiment et est redimensionné pour l'entourer (voir Fig. 27). Celui-ci est aussi ajusté selon la couleur de la faction. L'objectif a donc consisté principalement à distinguer les types de *GameObjects* sélectionnés avec leur *Tag*. La vidéo « *building-selection.mp4* » montre la réalisation de l'objectif.

Objectif : Modélisation d'un terrain de base

Description :
Le but de cet objectif est de générer un terrain de base qui formera la carte de jeu où les unités de combat pourront s'affronter et où les bâtiments pourront être positionnés. L'idée de cet objectif est tout d'abord d'avoir une surface simple pour tester le jeu et ses multiples fonctionnalités. Le niveau de détail du terrain n'est donc pas particulièrement visé ici, mais plus de temps pourrait y être consacré vers la fin du projet si désiré. Quelques éléments comme des arbres, une texture et quelques obstacles seront à positionner.

Étant donné que la majorité des éléments dans le jeu *RTS* entreront en interaction d'une manière ou d'une autre avec le terrain, il est nécessaire d'avoir sa structure en place dans la scène. L'évolution de cet objectif se fera en parallèle avec les autres fonctionnalités lorsque les besoins d'interactions seront à implémenter.

Réalisation :
L'*Asset* qui était employé originalement avec la caméra *RTS* importée [25] utilisait un *GameObject* cubique de base qui avait été redimensionné à l'échelle (100,0.1,100) pour former le plan de jeu. Cet objet a été remplacé par un objet spécialisé de type *Terrain* plus adapté à la tâche étant donné qu'il offre plusieurs fonctionnalités intéressantes comme le *Heightmap* et l'intégration des arbres.

La texture de gazon appliquée au terrain peut être visualisée à travers un grand nombre de figures dans les autres objectifs. Cette texture donne un effet plus réaliste sur la carte de jeu. Elle est aussi répétée dynamiquement pour remplir la surface du terrain tout en respectant le redimensionnement de l'objectif « *Bordures de terrain dynamiques* ». Cela assure donc que la texture n'est pas simplement étirée aux nouvelles dimensions, ce qui donnerait un effet visuel indésirable.

Quatre types d'arbres ont été ajoutés au terrain. Les arbres employés proviennent de la source [23] et ont été importés à partir de l'*Asset Store*. Ceux-ci ont été référencés dans l'objet de type *Terrain* employé et ont ensuite été placés sur la surface à l'aide des outils disponibles à partir de l'éditeur. Les options « *Mass Place Trees* » ainsi que le pinceau (*brush*) ont été respectivement utilisés pour le placement des arbres de sorte à avoir des distributions aléatoires ainsi que certaines régions avec une concentration d'arbres plus dense représentant des forêts. Ceci est présenté à la Fig. 33. Afin d'afficher des arbres sur le terrain de manière plus réaliste, leurs paramètres de détails et de distances de rendu ont été ajutées pour employer des « *Billboards* » plutôt que les objets 3D détaillés. Cela a permis d'améliorer la performance d'exécution du jeu puisque les arbres sont observés majoritairement à plus grande distance dans le jeu *RTS.* Aussi, cela a permis de supprimer plusieurs effets visuels nuisibles qui survenaient avec les paramètres par défaut. Par exemple, les arbres changeaient de couleur et de dimension alors qu'ils passaient du mode « détaillé » à « *billboard* », ce qui était très irritant visuellement. Aussi, les ombrages étaient très inconsistants selon la distance par rapport à la caméra. Les vidéos « *terrain-trees-<before|after>.mp4* » démontrent la différence avant-après des ajustements appliqués et de l'utilisation plus adéquate des *Billboards* pour corriger les effets visuels indésirables.

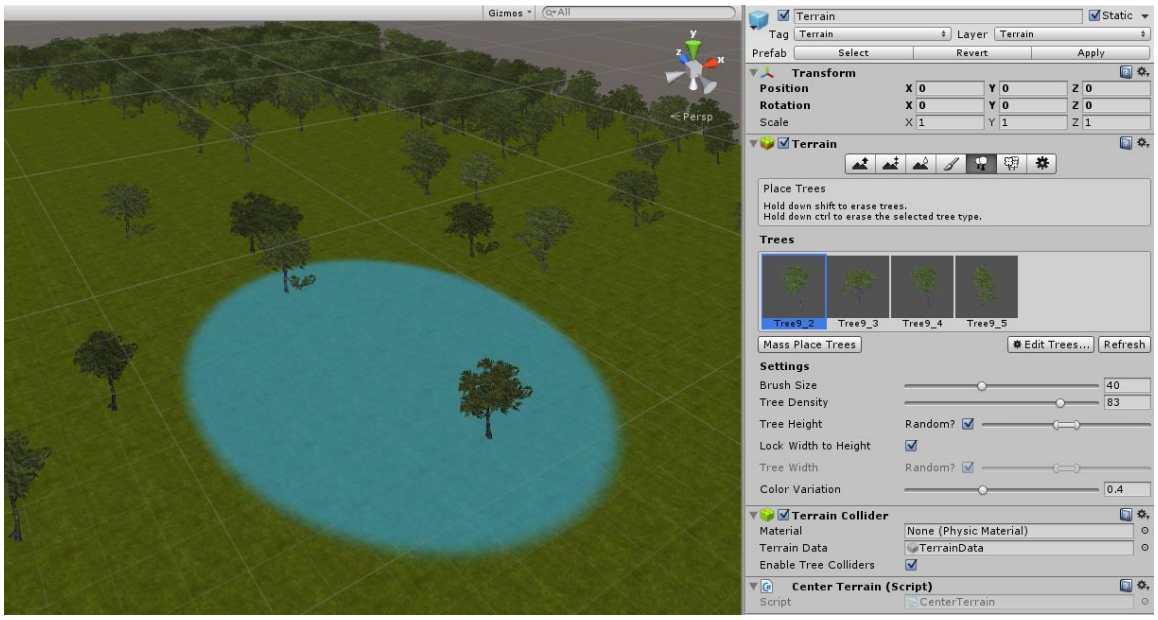

Fig. 33: Placement des arbres sur le terrain à l'aide des outils de l'éditeur

De plus, en prévision de l'objectif « *Contournement d'obstacles* », des éléments *SphereCollider* ont été ajoutés à la base des arbres tel que démontré à la Fig. 34. Ceux-ci sont employés ultérieurement à cet objectif pour déterminer la région autour des arbres qui ne peut pas être parcourue par les unités.

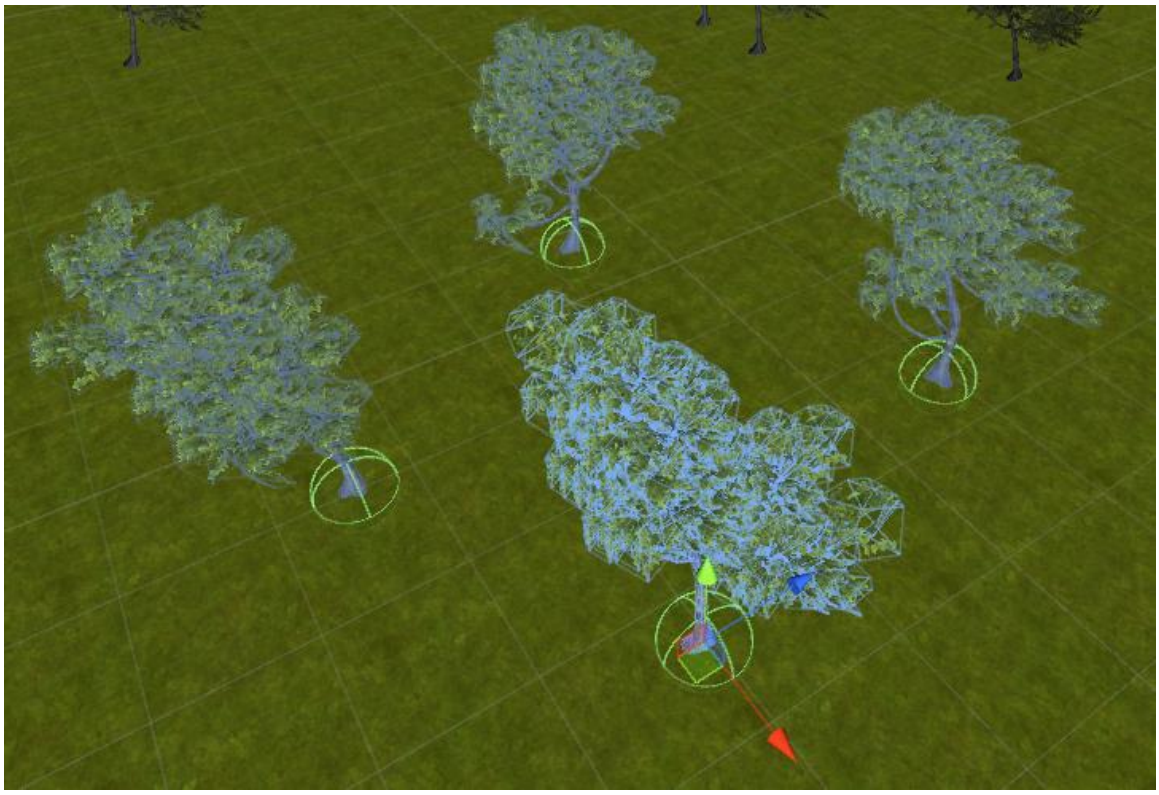

Fig. 34: Application de *SphereColliders* aux arbres sur le terrain

Objectif : Déplacements de base des unités

Description :
Cet objectif consiste à accomplir les translations et rotations des modèles d'unités afin de simuler leur déplacement linéaire sur le terrain. Pour cet objectif, la capture des commandes de déplacement devra être accomplie à l'aide d'un clic de souris sur le terrain alors qu'une ou plusieurs unités sont sélectionnées. L'unité devra tourner sur elle-même dans la direction adéquate avant de commencer à avancer tout droit.

Si une nouvelle commande de déplacement est donnée alors qu'une unité sélectionnée se déplace déjà, elle devra interrompre le mouvement actuel et commencer le nouveau déplacement demandé à partir de sa position courante. Aussi, lorsque les unités sont désélectionnées, celles-ci devront poursuivre leur déplacement jusqu'à ce qu'elles atteignent leur destination.

De plus, si un clic est spécifié sur une autre unité comme étant une commande d'attaque, l'unité sélectionnée devra maintenant se déplacer jusqu'à ce qu'elle soit en portée d'attaque. Dès qu'elle sera en portée, elle s'arrêtera immédiatement plutôt que de se rendre jusqu'à la position de l'unité ennemie. Il sera donc important de distinguer le type de clic selon s'il s'agit d'un déplacement simple, d'une attaque directe, car l'unité est déjà en portée, ou d'une combinaison entre déplacement et commande d'attaque.

Pour cet objectif, tous les déplacements seront accomplis linéairement sans se soucier des diverses contraintes du terrain comme les collisions, les obstacles ou les bordures de la carte. Ceux-ci seront gérés par la suite avec l'objectif « *Contournement d'obstacles* ». Cet objectif consiste donc à avoir les déplacements minimums fonctionnels avant de tenter d'incorporer des notions d'intelligence artificielle plus évoluées.

Réalisation :
Les déplacements linéaires qui avaient été réalisés à l'objectif « *Importer et implémenter les bâtiments* » avec la commande « *MoveToward* » employée pour faire sortir les unités du bâtiment a été transférée au script « *UnitManager.cs* » afin de permettre une utilisation globale de la procédure. Suite à la réalisation de cet objectif, tous les déplacements d'unités sont maintenant gérés par cette classe, peu importe le contexte où le mouvement est requis.

Toute la structure permettant la détection des clics de souris était déjà en place à l'intérieur du script « *SelectorManager.cs* » qui avait été réalisé lors d'un objectif précédent. Ceux-ci sont donc employés pour mettre à jour des variables de contrôle qui spécifient aux unités quelles actions elles devront prendre selon les types de clics captés. Afin de spécifier des commandes d'attaque ou de déplacement, le bouton droit de la souris est utilisé par défaut. Par contre, puisque le « *Right-Click* » était auparavant aussi employé pour faire tourner la caméra, il arrivait que plusieurs commandes se chevauchaient et posaient problème. Ainsi, les rotations de caméra ont été affectées d'une touche supplémentaire, soit la combinaison « *CTRL+Right-Click* », tandis que les commandes d'unités consistent maintenant aux clics droits simples. Ceci est un ajustement qui n'avait pas été prévu initialement, mais qui a été nécessaire pour distinguer les commandes de jeu entre elles.

Lorsque des commandes sont captées avec le bouton droit de la souris et que des unités sont sélectionnées, la position correspondante à l'endroit du clic est inscrite comme destination désirée de ses unités. Cela est accompli en ajustant une variable contenue dans « *UnitManager.cs* », et ce, pour toutes les instances d'unités actuellement sélectionnées. Chacune de ces instances effectue ensuite une vérification en boucle avec la fonction « *Update* » pour diriger l'unité vers la destination désirée si elle pointe dans la bonne direction. Sinon, les commandes « *LookRotation* » et « *RotateToward* » sont employées pour faire tourner progressivement l'unité en direction de la position désirée avant de lui permettre de se déplacer linéairement. Les commandes d'attaque consistent quant à elles à appliquer une destination égale à la position de l'unité visée, et de mettre à jour une autre variable du script « *UnitManager.cs* » pour référencer vers l'unité à attaquer. Le script « *UnitManager.cs* » délègue par la suite les commandes d'attaques au script « *TankManager.cs* » qui s'occupe d'accomplir les animations de canon pour l'unité correspondante selon ses paramètres de configuration. La seule différence entre la détection d'un déplacement simple ou d'un déplacement suivit d'une attaque consiste à la mise à jour de la variable de l'unité à attaquer selon l'objet détecté par le *Raycast* émit lors du clic de souris. Si le terrain est détecté, il s'agit d'un déplacement simple. Sinon, si l'objet détecté possède un *Tag* faisant partie d'une liste de contrôle spécifiée dans le script « *SelectorManager.cs* », l'attaque est permise en ajustant la variable vers l'unité visée. La liste de contrôle de *Tags* est employée ici simplement pour s'assurer de gérer et filtrer les conditions d'attaque permises, par exemple, pour ne pas attaquer un arbre si celui-ci était détecté par le clic de souris.

Les déplacements et rotations vers une unité à attaquer se poursuivent tant que celle-ci n'est pas en portée d'attaque. S'il n'y a pas d'unité à attaquer, le déplacement se fait tout simplement vers la destination désirée. La procédure employée dans « *UnitManager.cs* » se résume donc à faire les étapes suivantes :

```
Update()
{
   Si ennemi spécifié et en portée -> Attaque()
   Sinon si pas aligné avec la destination -> Rotation()
   Sinon si pas arrivé à la destination -> Translation()
}
```

Aussi, puisque chaque unité possède sa propre instance du script « *UnitManager.cs* », chacune peut attaquer indépendamment une cible spécifique, ou plusieurs unités peuvent attaquer une cible commune si chaque instance possède la même référence à attaquer.

Tout comme il l'avait été anticipée au rapport précédent, un angle « permissif » de rotation a été employé, c'est-à-dire que l'unité n'a pas besoin d'être parfaitement alignée avec sa cible pour permettre le déplacement dans sa direction. Un angle de 1° est employé, ce qui fait que l'unité est capable de se déplacer légèrement avant d'avoir besoin de se réajuster en rotation vers sa cible si cette dernière est également en déplacement. Ce petit angle fait que l'unité doit tout de même être relativement bien alignée avec sa cible, mais évite que celle-ci ne demeurer perpétuellement à sa position actuelle à tourner sur elle-même pour tenter de pointer correctement vers sa cible.

La mise en application des commandes réalisées à cet objectif est présentée dans la vidéo « *unit-basic-movement.mp4* ». Cette vidéo démontre l'exécution des déplacements des unités selon divers contextes de test et configurations d'unités existantes. On peut y voir que les unités se déplacent jusqu'à ce qu'elles atteignent la destination indiquée et qu'elles appliquent les rotations lorsque nécessaires pour s'aligner vers leur destination. Aussi, la vidéo présente l'exécution des animations de canon déléguées de « *UnitManager.cs* » vers « *TankManager.cs* » lorsqu'une cible est spécifiée. Nous pouvons aussi y voir que les unités se dirigent vers l'unité spécifiée pour l'attaque, qu'elles s'arrêtent lorsque leur cible entre en portée d'attaque, et qu'elles vont même jusqu'à suivre leur cible vers sa position variable lorsque celle-ci se déplace également. On y voit aussi que, lorsque les cibles sont déjà en portée, les tanks effectuent plutôt une rotation de leur canon sans accomplir de déplacement inutile pour rapidement pointer vers leur cible. De plus, la vidéo montre que, si une cible étant déjà en portée d'attaque sort de la portée de l'unité l'attaquant parce qu'elle s'est suffisamment déplacée, l'unité qui l'attaquait recommence à se déplacer automatiquement pour retourner en portée, sans avoir à lui spécifier une nouvelle commande d'attaque. Les unités sont donc continuellement à l'affût des mises à jour de la position de leur cible.

Dans la vidéo, il est possible de voir également que les variables de contrôle de la destination et de l'unité à attaquer sont mémorisées localement par chaque instance puisque les mouvements se poursuivent adéquatement même lorsque les unités se sont plus sélectionnées. Cela permet alors de rapidement passer d'une unité à l'autre pour indiquer des commandes distinctes. De plus, les nouvelles commandes captées avec les clics de souris ne viennent pas interférer avec les anciennes commandes spécifiées si les unités ne sont pas sélectionnées. Par contre, on voit aussi dans la vidéo que de nouvelles commandes spécifiées pour une unité qui accomplissait déjà un mouvement ou une attaque se voient adéquatement renouvelées. Donc, une unité en déplacement vers une destination cesse l'exécution de cette commande si on lui demande d'aller vers une nouvelle destination.

Les configurations d'attaque introduites lors des objectifs précédents sont toujours respectées avec l'ajout des fonctionnalités de cet objectif. Comme démontré dans la vidéo, une des commandes d'attaque envers une unité de sol est spécifiée pour l'unité pouvant seulement attaquer les ennemis aériens. Dans ce cas, l'attaque est annulée puisque l'unité concernée ne peut pas accomplir la requête et celle-ci s'arrête alors à sa position actuelle.

Objectif : Contournement d'obstacles

Description :
Cet objectif est une extension de « *Déplacements de base des unités* ». Il consiste à ajouter des notions d'intelligence artificielle plus avancée dans les mouvements de base qui seront implémentés. Entre autres, les obstacles positionnés sur le terrain devront être contournés automatiquement par les unités. Celles-ci devront donc trouver la trajectoire à emprunter afin d'atteindre leur cible. Par exemple, si un bâtiment se trouve à mi-chemin entre une unité et sa position finale, elle devra en faire le tour sans le frapper.

Réalisation :

Afin de réaliser un contournement d'obstacles dynamiques sur le terrain, un algorithme de recherche de trajectoire *A** est employé comme point de départ et a été implémenté en suivant les propositions mentionnées dans le tutoriel [16]. Ce tutoriel a permis de mettre en place la structure de base des scripts et procédures nécessaires pour appliquer l'algorithme *A** à des *GameObjects* simples. Les scripts ont ensuite été adaptés et améliorés progressivement selon les besoins du jeu afin d'ajouter d'autres notions d'intelligence artificielle encore plus évoluées et optimisées. Les réalisons relatives à chaque partie impliquée sont présentées par la suite.

Tout d'abord, une grille 2D de *Nodes* est générée à l'aide d'un script « *Grid.cs* ». Ce script permet de générer les cellules qui contiennent l'information respective à chaque position disponible sur le terrain. Ces cellules sont requises par l'algorithme *A** afin d'évaluer les positions parcourables en fonction de l'état des informations qui y sont contenues. Les *Nodes* ont une taille ajustable, mais une dimension de 2x2 est actuellement employée (*radius=1*) et cela donne d'assez bons résultats sans surencombrer le terrain de subdivisions de positions parcourables. Le script est affecté à un *GameObject* nommé « *Pathfinder* » (Fig. 35) qui maintien actif l'ensemble des procédures qui accomplissent le contournement d'obstacles grâce à une recherche de parcours pour les unités.

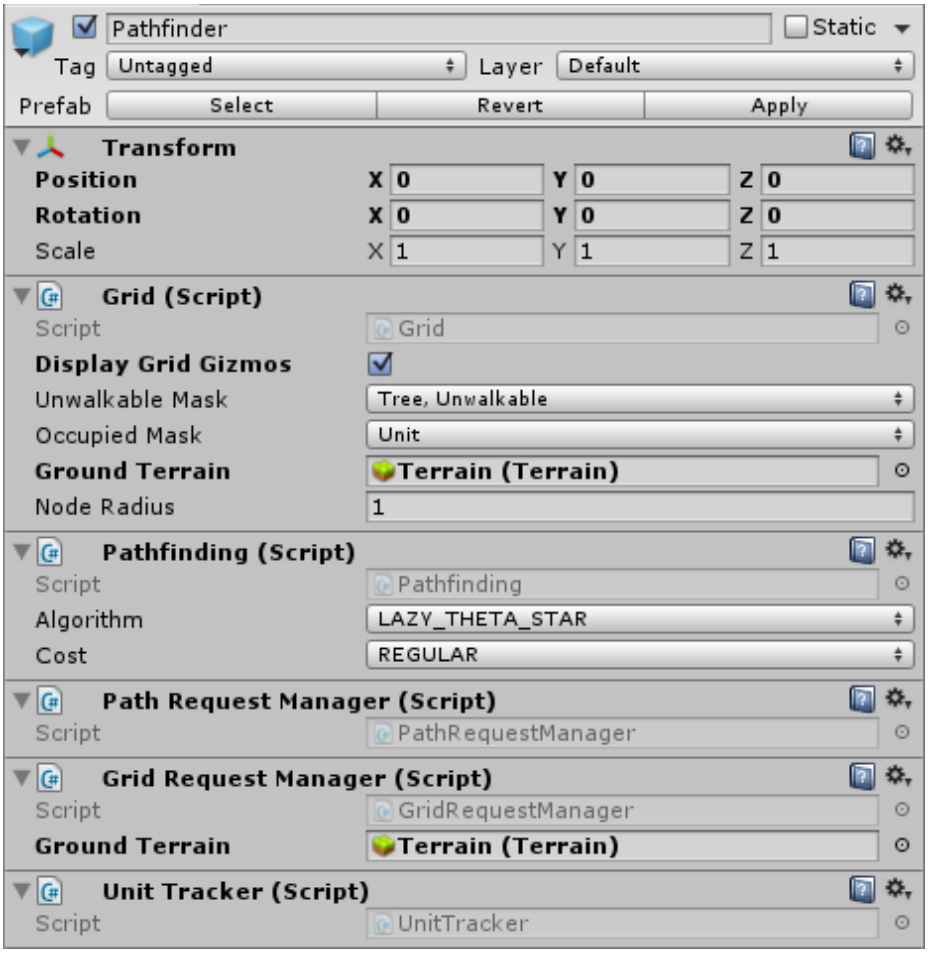

Fig. 35: Objet *Pathfinder* contenant tous les scripts requis pour la recherche de trajectoire

Lorsque la grille 2D de *Nodes* est générée, chaque case est testée à l'aide de la méthode « *Physics.CheckSphere* » qui évalue la position de la *Node* correspondant à l'aide d'une sphère et d'un masque de *Layer*s pour déterminer s'il y a un chevauchement d'objets de type *Collider*. Ainsi, comme il l'est démontré à la Fig. 35, deux types de masques sont employés, soit « *Unwalkable Mask* » qui correspond à des cases complètements bloquées et qui ne pourront jamais être traversées par les unités, ainsi que « *Occupied Mask* » qui correspond aux positions traversables, mais actuellement occupées par un objet. En d'autres termes, tous les *GameObjects* qui possèdent un *Collider* et dont le *Layer* leur étant appliqué correspond à « *Tree* » ou « *Unwalkable* » sont nécessairement annotés comme non-traversables sur la grille. Similairement, les unités affectées du *Layer* « *Unit* » sont annotées pour occuper les *Nodes* pour lesquelles l'évaluation de la sphère retourne une détection de collision positive. Évidemment, les valeurs de masques présentées à la Fig. 35 peuvent être ajustées tel que nécessaire afin d'accommoder l'ajout de nouveaux éléments. Afin de différencier les deux types de masques, et pour observer la mise en opération des *Nodes* à travers l'ensemble du projet, celles-ci sont affichées dans la scène avec différentes couleurs grâce aux scripts activés par *OnDrawGizmos*. En activant les *Gizmos*, les *Nodes* affichées à l'écran respectent un code de couleur constant, soit blanches si elles sont disponibles pour le parcours, jaunes si occupées et rouges si bloquées.

Initialement, il avait été envisagé que le même *BoxCollider* que celui utilisé pour la sélection du bâtiment (Fig. 27) serait employé pour déterminer sa région non parcourable par les unités sur le terrain. Par contre, la forme de cette boîte créait une région bloquée par les unités là où elles devaient se retrouver lorsqu'elles sortaient du bâtiment, ce qui posait problème, car les unités ne trouvaient jamais de trajectoire. Ainsi, deux *BoxColliders* additionnels ont été ajoutés de sorte à mieux respecter la forme du bâtiment, tel que visibles à la Fig. 36. Les *BoxColliders* ajoutés sont très proches du sol afin de s'assurer, lors des clics de souris pour la sélection du bâtiment à l'aide de *Raycasts*, que ceux-ci n'interfèrent pas avec l'autre *BoxCollider* à détecter pour la sélection. Ceci était parfois rencontré pendant les tests, mais, en plaçant ces boîtes supplémentaires près du terrain, nous obtenons les *Nodes* bloquées désirées sans nuire à la procédure de sélection. Les *BoxColliders* sont ici affectés au *Layer* « *Unwalkable* » pour empêcher de traverser le bâtiment, tel que présenté à la Fig. 37.

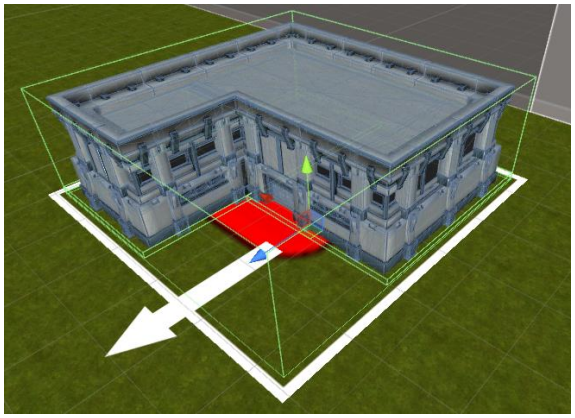

Fig. 36: Application de *BoxColliders* additionnels au bâtiment pour définir la région non parcourable

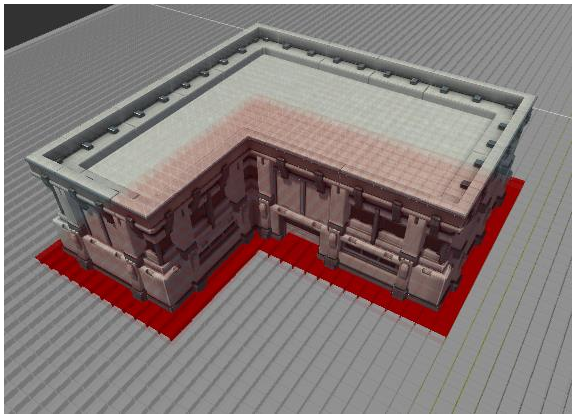

Fig. 37: Affichage des *Nodes* non parcourables sur la grille par la détection des *BoxColliders* du bâtiment

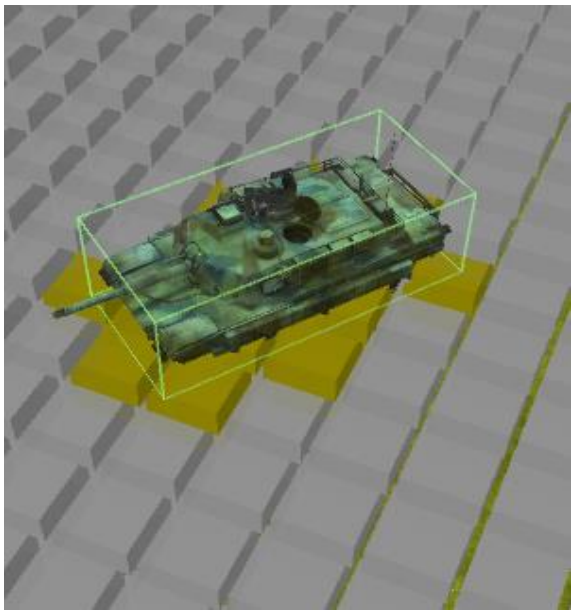

Fig. 38: Affichage des *Nodes* occupées par une unité sur la grille avec la détection de son *BoxCollider*

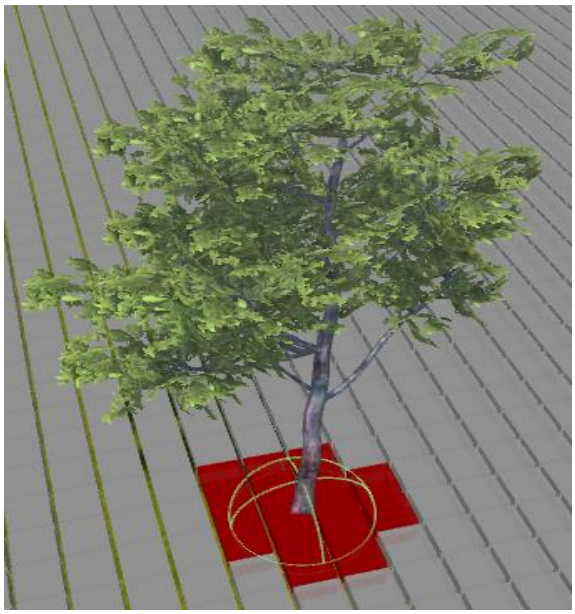

Fig. 39: Affichage des *Nodes* non parcourables sur la grille avec la détection du *SphereCollider* d'un arbre

Les objets *BoxCollider* déjà appliqués à chacune des unités sont quant à eux directement utilisés pour déterminer les endroits où les déplacements sont impossibles. Par contre, les unités emploient le « *Occupied Mask* » plutôt que le « *Unwalkable Mask* » pour annoter les *Nodes* comme temporairement indisponibles tel que démontré à la Fig. 38. Les cases temporairement occupées par les unités sont continuellement mises à jour lorsque l'unité se déplace ou tourne. Cette mise à jour est accomplie en utilisant les limites « *bounds* » du *BoxCollider* afin d'ajuster localement les *Nodes* impliquées autour de la position actuelle de l'unité en sauvegardant l'ID assigné à cette unité à l'aide du script « *UnitTracker.cs* ». Ce script est employé au lieu de la méthode « *GetComponentID* » étant donné que celle-ci retourne un ID différent selon la référence mémoire de la variable des unités, ce qui posait parfois problème lors des transferts de référence entre les scripts « *UnitManager.cs* et « *Pathfinding.cs* ». De manière similaire, une mise à jour locale de la grille est réalisée lorsque l'unité est détruite où lorsqu'un bâtiment est ajouté sur le terrain pour immédiatement appliquer les changements survenant sur l'aire de jeu. Cette mise à jour locale de la grille est gérée en combinaison par les scripts « *GridRequestManager.cs* » et « *Grid.cs* ». Également, les *SphereColliders* auparavant appliqués à la base des arbres sont employés pour produire des obstacles non traversables supplémentaires sur le terrain comme présenté à la Fig. 39.

Avec l'ajustement des informations précédentes sur la grille, l'algorithme de recherche de trajectoire a pu être implémenté et testé. Le script accomplissant cette tâche est nommé « *Pathfinding.cs* » et est appliqué à l'objet *Pathfinder*. Trois algorithmes de recherche de trajectoire ont été implémentés, chacun étant une amélioration du précédent. Le premier est l'algorithme *A** de base [6], le deuxième est *Theta** [19, 20] et le troisième est *Lazy Theta** [12, 21]. L'idée générale d'*A** est d'explorer les cases voisines à une case actuellement évaluée en partant par la *Node* de départ, de calculer la distance parcourue

entre les cases explorées est la case de départ pour obtenir le poids du chemin nécessaire pour atteindre cette case, et de calculer une distance par rapport à la case objectif pour favoriser le choix de *Nodes* situées dans cette direction. Les cases voisines explorées sont itérativement évaluées selon le chemin le plus cours à chaque itération et les calculs de poids des chemins sont continuellement mis à jour. La minimisation du poids permet de trouver le trajet le plus cours, direct et préférable possible. Deux listes (*openset* et *closedset*) sont employées pour maintenir l'état des cases déjà explorées, jusqu'à ce que la case finale soit atteinte ou qu'aucune trajectoire n'existe (ex. : case isolée ou complètement bloquée). Une caractéristique importante de cet algorithme est qu'il trouve nécessairement une trajectoire si une solution existe et celle-ci est optimale pour atteindre la cible le plus rapidement possible. Un exemple de calcul de trajectoire optimale par l'algorithme *A** est présenté à la Fig. 40, où l'on peut également voir les valeurs de poids du trajet calculées.

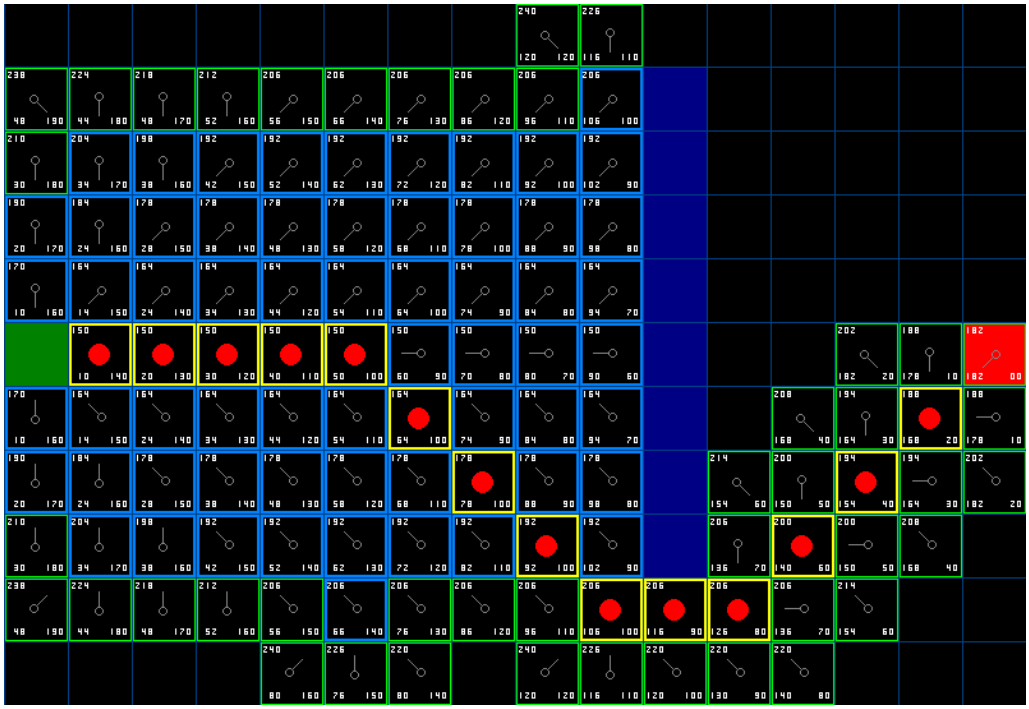

Fig. 40: Visualisation du calcul des poids de parcours avec l'algorithme de recherche *A** [36]

L'algorithme *Theta** qui a ensuite été implémenté se base sur *A**, mais utilise une notion additionnelle de « *Line-of-Sight* » afin de trouver des trajectoires plus lisses. En effet, comme on peut le voir avec la Fig. 40, l'algorithme *A** est limité à des connexions de cases voisines à 8 directions (mouvements droits ou diagonaux). Cela faisait que les unités réussissaient à trouver une trajectoire valide pour contourner des obstacles sur la grille 2D, mais les mouvements étaient peu naturels puisqu'ils étaient très « carrés » et saccadés. L'addition du « *Line-of-Sight* » par *Theta** fait que n'importe quel angle de parcours est permis au lieu des tranches de 45°, tant que deux cases peuvent directement se voir sur une grande distance. Ainsi, cette amélioration permet de trouver des trajectoires plus naturelles, telles que montré à l'exemple de la Fig. 41. La dernière version *Lazy Theta** est simplement une amélioration algorithmique qui utilise un prétraitement des cases pour accélérer le temps de calcul, mais qui offre le même résultat final que *Theta** de base.

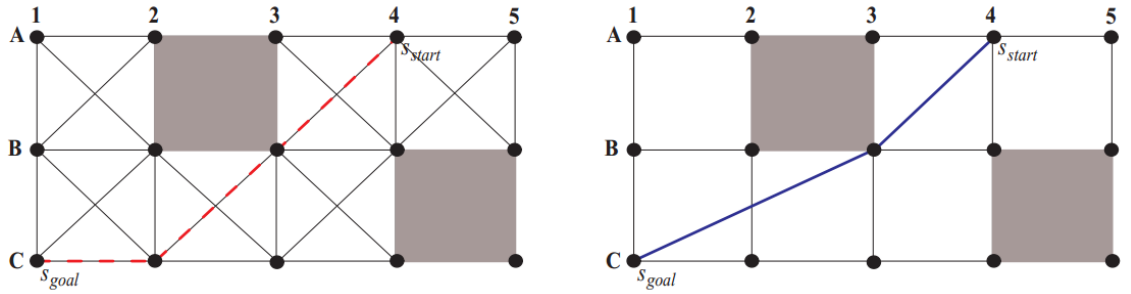

Fig. 41: Différence de trajectoires obtenues par *A** (gauche) et *Theta** (droite) [20]

Avec les algorithmes de recherche de parcours implémentés, les commandes précédemment employées pour effectuer les mouvements linéaires à l'objectif « *Déplacements de base des unités* » sont tout simplement redirigées vers des requêtes de « *pathfinding* » en spécifiant la position courante de l'unité (début de trajectoire) et la destination désirée ou la position de l'unité à attaquer selon le cas (fin de trajectoire). Les trajectoires retournées par l'algorithme de recherche sont une liste de « *waypoints* » qu'il faut simplement assigner itérativement comme nouvelles destinations. Donc, lorsque la position actuelle de l'unité est égale au « *waypoint 1* », la position désirée est ajustée au « *waypoint 2* » et ainsi de suite jusqu'à la fin de la trajectoire, tel qu'illustré à la Fig. 42. Cela est géré dans le script « *UnitManager.cs* » à l'aide d'un tableau de « *Vector3* ».

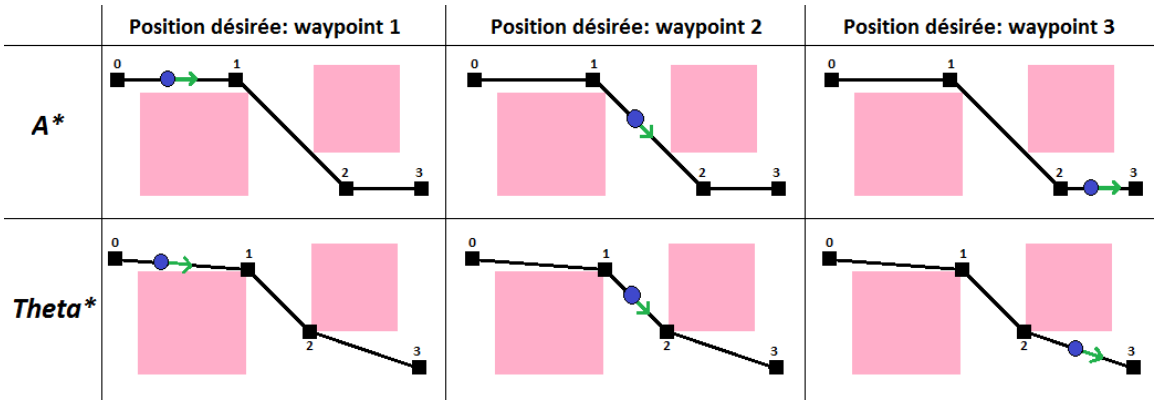

Fig. 42: Mise à jour itérative des « *waypoints* » de la trajectoire obtenue pour éviter les obstacles

Afin de permettre le traitement des requêtes de trajectoires, un système de file de traitement des requêtes accumulées est utilisé avec le script « *PathRequestManager.cs* » attaché à l'objet *pathfinder* afin de répartir les demandes de trajectoires des multiples unités à travers plusieurs « *frame updates* ». Cela est nécessaire, sans quoi le jeu figerait souvent, car les calculs assez lourds de recherche de trajectoire empêcheraient une mise à jour constante et rapide de la scène. De plus, les appels de mise à jour des trajectoires ont été limités à l'aide de quelques délais minimums entre les requêtes ainsi que de certaines conditions déterminées expérimentalement pour limiter les calculs redondants ou superflus. Par exemple, la distance entre la position actuelle d'une unité à attaquer et sa dernière position évaluée lors de la requête précédente est utilisée pour déterminer si une mise à jour de la position de l'unité ciblée est nécessaire, comme illustré à la Fig. 43.

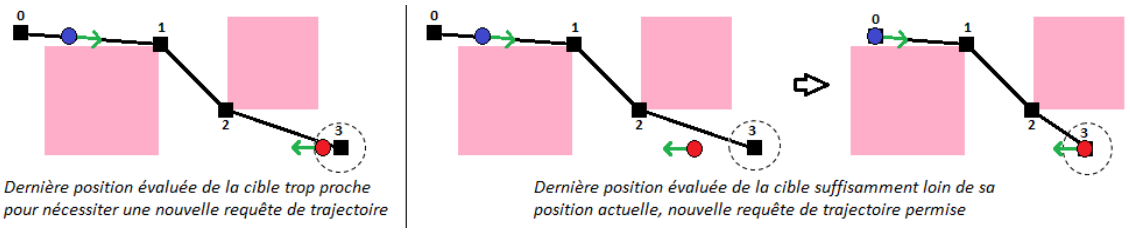

Fig. 43: Mise à jour de la trajectoire si l'unité attaquée s'est suffisamment déplacée

Il est à noter qu'à chaque fois qu'une mise à jour de position de l'unité attaquée est effectuée, la trajectoire est complètement réévaluée, et non seulement le dernier segment. Cela avait été considéré comme possible amélioration de performance, mais produisait des cas où les unités ne considéraient pas le chemin le plus optimal ou omettait les nouveaux obstacles qui se dressaient entre elle et sa cible. Certains de ces cas sont illustrés à titre d'exemple à la Fig. 44 pour justifier le choix de mise à jour complète réalisé.

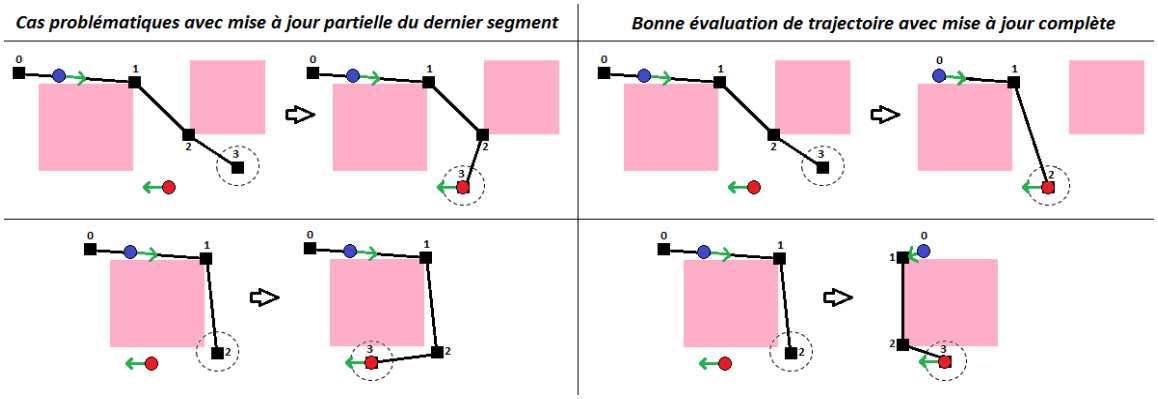

Fig. 44: Trajectoires erronées par mise à jour partielle (gauche) par rapport aux trajectoires voulues (droite)

En plus des requêtes de trajectoire employées pour contourner les obstacles et les cases occupées dynamiquement par d'autres unités en mouvement, plusieurs petits ajustements ont été apportés afin de produire des comportements plus intelligents par les unités. Par exemple, lorsqu'une unité possède une cible d'attaque, mais que celle-ci n'est pas directement visible (*Line-of-Sight*), la procédure d'alignement du canon n'est pas accomplie inutilement. L'unité attend donc d'avoir sa cible en ligne de mire avant de pointer vers elle. Aussi, si une unité visée se retrouve trop proche d'une unité attaquante, ce qui l'empêche d'attaquer à cause de la condition de la valeur minimale de portée, l'unité essaie automatiquement de s'éloigner de sa cible dans la direction opposée pour devenir en portée d'attaque. Auparavant, une telle condition faisait que l'unité continuait à se diriger vers sa cible jusqu'à qu'elle soit directement par-dessus, mais maintenant elle tente de résoudre le problème de distance d'attaque en ajustant sa position. De plus, les unités ne s'arrêtent plus immédiatement lorsqu'elles entrent en portée d'attaque (valeur maximale), mais poursuivent légèrement leur course pour se trouver à mi-chemin entre la valeur minimale et maximale de portée. Cela fait que l'unité attaquante n'a pas besoin de continuellement réajuster sa position par petits intervalles si l'unité attaquée se déplace légèrement, et qui permet de donner une impression plus réaliste des mouvements, car ils s'en retrouvent moins saccadés. Ces améliorations sont illustrées à l'aide de la Fig. 45.

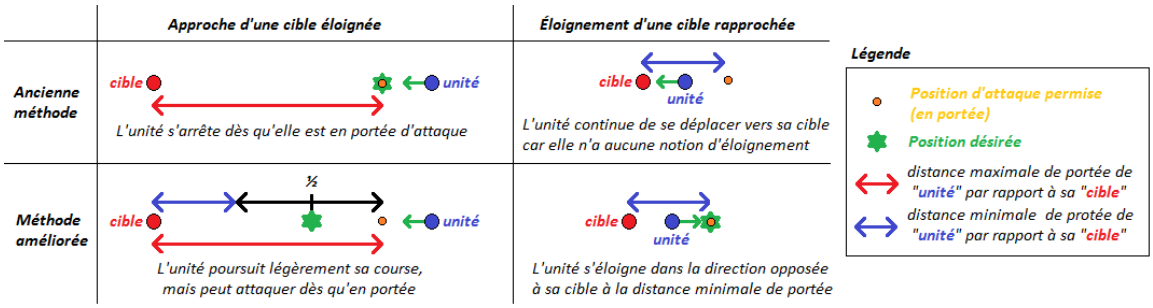

Fig. 45: Améliorations apportées par rapport à la logique de positionnement des unités

Afin de visualiser les chemins obtenus par les algorithmes lors des tests, ceux-ci ont été affichés à l'aide de *Gizmos*. Les algorithmes sont tout d'abord présentés dans la vidéo « *unit-pathfinding-algorithm-comparison.mp4* » qui illustre l'ajout de réalisme de trajectoire en appliquant *Lazy Theta** au lieu de l'algorithme de base *A** étant donné que les chemins trouvés ne sont plus limités à des angles de 45°. Plusieurs exemples de contournement d'obstacles, de mise à jour dynamique des trajectoires des unités selon la position de leur cible variante, et d'ajustement du canon seulement lorsque l'unité visée est directement dans le champ de vision sont quant à eux tous présentés dans la vidéo « *unit-pathfinding-examples.mp4* ». Les vidéos nommés « *unit-pathfinding-attackmove-initial-method.mp4* » et « *unit-pathfinding-attackmove-improved-method.mp4* » démontrent plus en détail l'ajustement du canon selon le champ de vision des unités. Ces vidéos illustrent également la réduction de la quantité de requêtes de « *pathfinding* » effectuées grâce aux délais de temps et selon les variations de position de l'unité ciblée à l'aide d'affichages des requêtes dans la console. L'ajustement de position selon la distance de portée maximale est plus difficile à voir, mais il peut être observé aux deux vidéos précédents où l'on remarque que l'unité prend beaucoup moins de temps à rattraper sa cible près du coin de l'obstacle puisqu'elle n'arrête pas de bouger dès lorsqu'elle entre en portée d'attaque. Les vidéos « *unit-minrange-problem.mp4* » et « *unit-minrange-resolved.mp4* » démontrent le cas mentionné par rapport à la Fig. 45 pour la distance minimale. La vidéo « *grid-node-update.mp4* » montre quant à elle l'ajustement dynamique des cases formant la grille de parcours du terrain selon les unités en déplacement, la position des arbres et l'ajout de nouveaux bâtiments sur le terrain. Finalement, la vidéo « *unit-destruction-grid-node-update.mp4* » illustre l'ajustement des *Nodes* lorsqu'une unité finit d'exécuter sa procédure de destruction pour libérer l'espace qu'elle occupait sur la grille.

<u>Objectif :</u> Projectile et effet d'impact

<u>Description :</u>
Lorsqu'une unité en attaque une autre, un projectile allant logiquement de celle qui tire vers la cible devra être affiché. Le projectile partira du bout du canon et terminera sa course à l'intérieur de l'unité ciblée pour disparaître. Selon le type d'unité, la trajectoire à suivre pourrait être soit en ligne droite ou selon un arc. Par exemple, les unités à configuration #1 dans l'objectif « *Création du graphe de scène des unités* » tireront en ligne droite, alors que celles à configuration #3 nécessitant un angle pointant vers le haut tireront selon un arc. La Fig. 46 présente de manière approximative les résultats de trajectoires attendues.

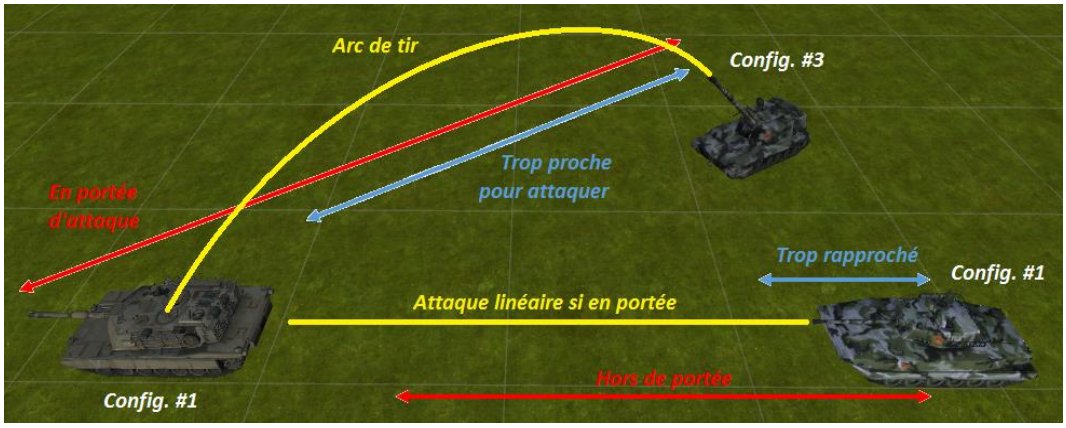

Fig. 46: Trajectoire des projectiles pour l'attaque selon la configuration

Un effet d'impact comme une explosion ou l'émission de particules sera ensuite affiché lorsque la cible est atteinte. De plus, l'émission de quelques particules lorsque le projectile quitte le bout du canon pourrait être appliquée.

Réalisation :

Afin de limiter le nombre d'instances créées et détruites pour simuler les projectiles et les divers effets de particules, ceux-ci seront affichés et cachés à répétition, ce qui est moins lourd en termes de ressources d'allocation que de les générer et les détruire à chaque fois. Étant donné que chaque unité ne peut pas tirer plus d'un projectile à la fois, il possible de limiter la quantité d'instances à créer par unité et de les réutiliser à chaque nouveau tir. Ces instances sont présentées en pâle à la Fig. 47.

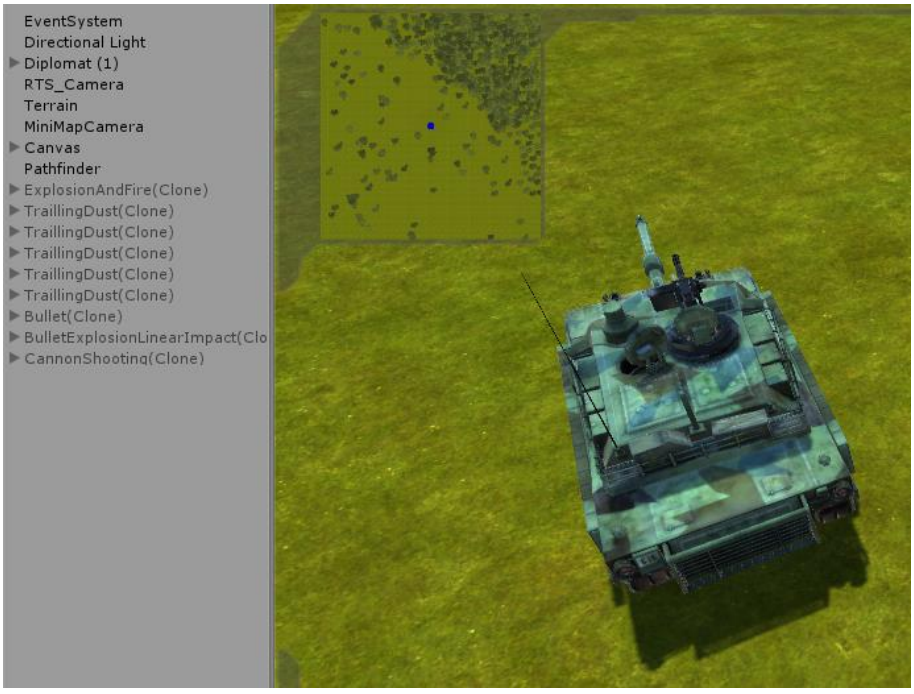

Fig. 47: Ensemble des instances générées (objets pâle) pour chaque unité existante

À la Fig. 47, plusieurs des effets (ie : « *TraillingDust* » et « *ExplosionAndFire* ») sont employés pour les objectifs « *Effet de poussière d'unité en mouvement* » et « *Destruction d'une unité* ». Les effets « *BulletExplosionLinearImpact* » et « *CannonShooting* » sont ceux qui s'appliquent à cet objectif. Dans le cas de l'unité tirant en angle, un effet appelé « *BulletExplosionArcImpact* » est plutôt employé pour ajouter quelques variations spécifiques à l'angle d'impact pour être plus réalistes. Plus réalité, chacun des effets précédents consiste en un ensemble de « *ParticleSystems* » qui affichent chacun différentes particules formant un effet global. Par exemple, un effet d'impact est une combinaison de *Sprites* pour afficher de la fumée, un autre pour afficher des flammèches et un supplémentaire pour donner un effet de *flash*. L'ensemble des groupements d'effets de particules ont été réalisés grâce aux tutoriels et des exemples tirés de l'*Asset Store*, et ont ensuite vu leurs paramètres modifiés directement à partir de l'éditeur de particules qui offre une interface très conviviale et détaillée pour obtenir des résultats précis selon la couleur, la durée de vie des particules, la vélocité, la forme des émissions, les délais, etc. La documentation d'*Unity* et les exemples d'explosion [32] et de fumée [31] sont particulièrement suivis afin de produire les effets obtenus.

Le projectile « *Bullet* » est quant à lui utilisé pour l'ensemble des unités où un projectile de tank doit être tiré et consiste simplement en un cylindre et d'une sphère de couleur gris foncé se chevauchant, tel que présenté à la Fig. 48.

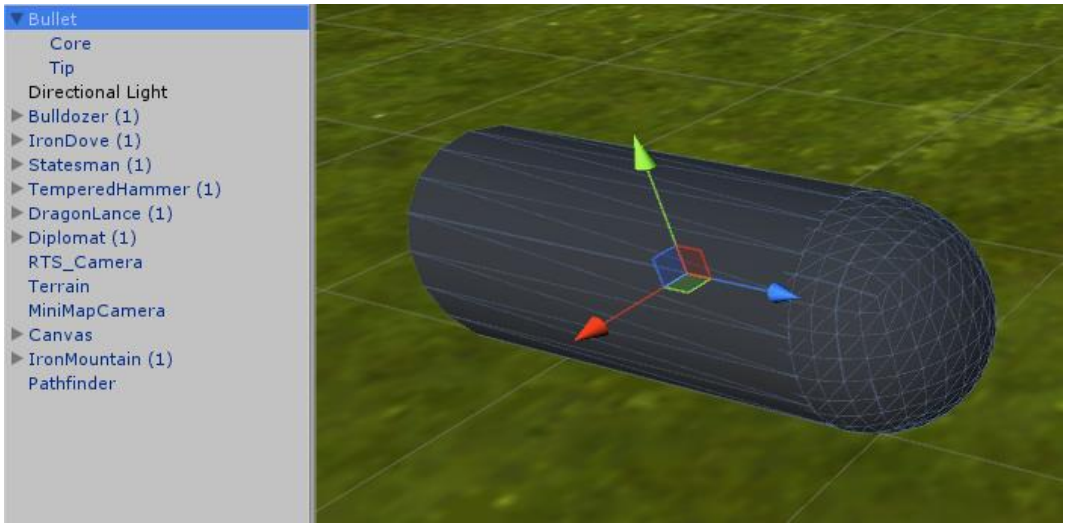

Fig. 48: Objets formant le projectile « *Bullet* »

Les calculs de trajectoires ont été accomplis selon les sources de références [26, 29, 30] tel que prévu initialement. Ceux-ci sont réalisés dans « *TankManager.cs* » et sont appelés avec des *Coroutines* lorsqu'une cible est attribuée à « *UnitManager.cs* » et est déléguée vers « *TankManager.cs* » comme indiqué lors d'objectifs précédents. La fonction « *ExecuteProjectileAnimation* » située dans « *TankManager.cs* » vérifie à chaque « *Update* » plusieurs conditions avant de permettre la séquence de projectile et d'effets de particules. Entre autres, des conditions comme la présente d'une unité attaquée n'étant pas encore détruite tout en étant en portée d'attaque et le respect d'un délai minimal depuis le tir du projectile précédent sont vérifiées.

La procédure effectuée pour chaque tir, selon le type (linéaire ou arc), est comme suit :

```
ExecuteProjectileAnimation()
{
   Validation des conditions de tir, sinon sortie immédiate
   Obtenir l'angle de tir pour ajuster la position de l'effet de tir
   Lancer la coroutine « ProjectileAnimation(TrajectoryFunction) »
      (avec « LinearTrajectory » ou « ArcTrajectory » selon le cas)
   Remise à zéro du délai d'attaque
}
```

La fonction « *ProjectileAnimation* » appelle tout d'abord le groupe d'effets de particules « *CannonShooting* » qui a été référencé en entrée au script « *TankManager.cs* ». Chacun des groupes d'effets de particules est rendu visible en activant les éléments, puis est animé en gérant les délais d'exécution de chacun, les conditions d'affichage et autres paramètres spécifiques aux *ParticleSystems*. Il y a en effet beaucoup de paramètres à ajuster à chaque fois, c'est pourquoi un script séparé nommé « *EffectManager.cs* » a été créé afin d'appeler plus simplement les groupes d'effets lorsque requis et ce script s'occupe de gérer tous les détails d'exécution selon les cas rencontrés par la suite.

Ensuite, « *ProjectileAnimation* » appelle la fonction de trajectoire spécifiée en entrée qui fait déplacer le projectile de manière adéquate selon le type en partant du bout du canon pour aller jusqu'à la cible. La position du bout de canon est obtenue à l'aide d'un *GameObject* nommé « *Output* » minutieusement positionné sur le modèle 3D de chaque tank comme enfant au canon pour avoir une position relative à son déplacement. Lorsque le projectile atteint finalement la cible, la fonction « *ProjectileAnimation* » affiche les effets de particules pour l'impact qui est adéquatement référencé selon le type d'unité. L'ensemble de ces paramètres ajoutés pour cet objectif sont présentés à la Fig. 49.

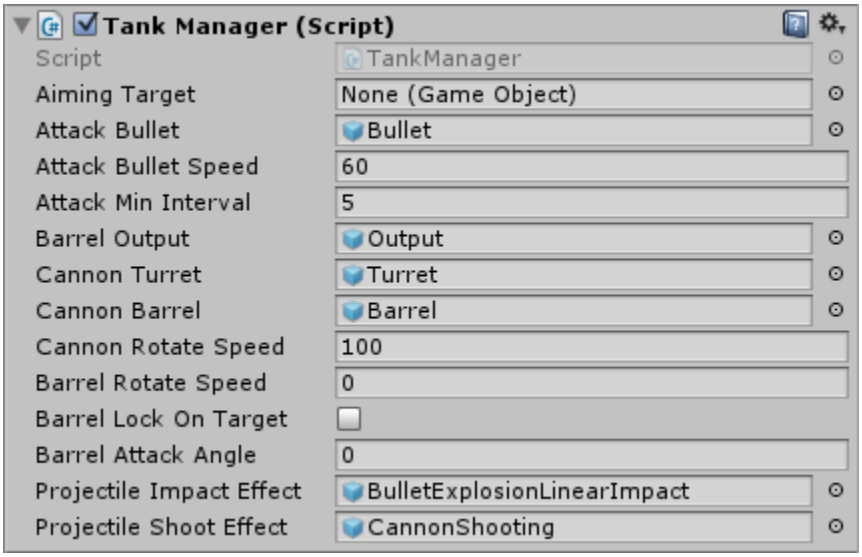

Fig. 49: Paramètres de « *TankManager.cs* » avec ajout de références pour projectile et effets de particules

Les deux types de trajectoires « *LinearTrajectory* » et « *ArcTrajectory* » sont quant à elles des fonctions « *delegate* » qui calculent, positionnent et tournent le projectile selon le

chemin correspondant. Pour la trajectoire linéaire, seul « *MoveToward* » est requis et est utilisé en combinaison avec la vitesse du projectile spécifiée (Fig. 49) et l'unité visée. Dans le cas de la trajectoire en arc, la position en XZ est calculée par interpolation linéaire par rapport à la cible alors que la position en Y est déterminée à l'aide du sinus de la variation de temps par rapport au début du tir normalisé entre [0,1]. Le calcul est inspiré des formules de vélocité de projectiles réels, mais possède quelques approximations telles que proposées dans les références consultées, ce qui permet d'avoir un effet visuel très satisfaisant tout en ne nécessitant qu'un simple calcul (seulement un sinus plutôt que plusieurs intégrales). De plus, la formule de trajectoire fonctionne bien selon différents angles et distances, ce qui permet de la réutiliser facilement si l'on change les paramètres d'entrée du script. Afin de bien afficher le projectile, la rotation de celui-ci est également ajustée en utilisant la tangente pour chaque intervalle de position calculé le long de la trajectoire arquée. Cette tangente est obtenue en effectuant la différence entre la position actuelle du projectile et la position précédente mémorisée temporairement à chaque itération de sorte à obtenir le vecteur de direction. Puis, ce vecteur est employé avec la méthode « *Quaternion.LookRotation* » pour appliquer la rotation correspondante. Puisque le projectile est symétrique en rotation autour de son axe central, seul ce vecteur est nécessaire pour bien le tourner.

Malgré qu'il soit possible de voir les projectiles et effets réalisés pour cet objectif à travers un vaste nombre de vidéos, la vidéo « *unit-projectile-impact-effects.mp4* » est offerte pour démontrer plus spécifiquement l'application des procédures impliquées par cet objectif.

Objectif : Affichage de mini-carte
Ancien nom : « *Affichage de carte aérienne (bird's eve view)* »

Description :
Comme la majorité des jeux *RTS*, une mini-carte vue du haut (*bird's eye view*) devra être affichée dans le coin de l'interface de jeu. Sur cette carte, les éléments importants positionnés sur le terrain y seront affichés comme les bâtiments et les unités de combat. Le but de cette carte est donc d'offrir une vue d'ensemble du terrain à une certaine altitude orthogonale au plan de jeu. Aussi, un clic à une position particulière sur cette carte devra accomplir le déplacement de la caméra du joueur à la position correspondante sur le terrain afin de visualiser la zone désirée. Cette mini-carte devra donc permettre les déplacements rapides à travers l'ensemble du terrain.

Réalisation :
Pour implémenter la mini-carte, une caméra additionnelle nommée « *MiniMapCamera* » a été ajoutée à la scène à la position (0,50,0). Par contre, la position en hauteur importe peu tant que les plans limites du *Frustum* (i.e. : *clipping planes* dans l'éditeur *Unity*) contiennent l'espace occupé par les objets sur le terrain. Ainsi, les valeurs minimale et maximale de ces plans de la caméra ont été ajustées à 0.3 et 100 respectivement. La position en XZ demeure fixe à l'origine puisque le terrain est déjà centré à l'origine automatiquement avec le script précédemment réalisé et présenté à l'objectif « *Positionnement centré du terrain* ».

Afin de ne pas avoir d'effets de perspective sur la mini-carte, la caméra a été ajustée pour le mode orthographique et une couleur de fond gris foncé (50,50,50) a été assignée pour remplir le Z-Buffer là où il n'y a pas élément, soit hors du terrain, ce qui donne un effet de « vide » à l'extérieur de l'aire de jeu.

Ensuite, afin de s'assurer de ne capturer que des éléments bien spécifiques avec cette caméra, le paramètre « *Culling Mask* » a été ajusté avec les *Layers* « *Minimap* », « *Terrain* » et « *Tree* ». En appliquant ces masques à la caméra, celle-ci est uniquement capable de voir les éléments qui possèdent l'un de ces *Layers* plutôt que de tout voir selon la configuration par défaut. Cette idée est tirée de [28] traitant justement de méthode pour afficher une mini-carte à l'écran. Les deux derniers masques sont donc appliqués aux objets correspondants pour les afficher sur la mini-carte, tandis que le *Layer* « *Minimap* » est appliqué à certains sous-*GameObjects* particuliers des unités, de sorte à avoir leur position tout en affichant uniquement ce sous-élément plutôt que l'ensemble de l'unité. Plus spécifiquement, un *Sprite* a été appliqué aux unités, tel que présenté à la Fig. 50. De manière similaire, un *Sprite* a été ajouté au bâtiment (Fig. 51), mais ce dernier est plus grand que celui des unités afin de pouvoir les différencier sur la mini-carte. Aussi, la couleur de ces *Sprites* est automatiquement ajustée selon la couleur de faction spécifiée pour les unités et le bâtiment avec « *UnitManager.cs* » et « *BuildingManager.cs* » lors de leur initialisation. Dans les vidéos et images qui suivent, il est d'ailleurs possible de voir que les couleurs rouge et bleu sont appliquées à différentes unités en fonction du paramètre de couleur de faction spécifié dans leurs références aux scripts respectifs.

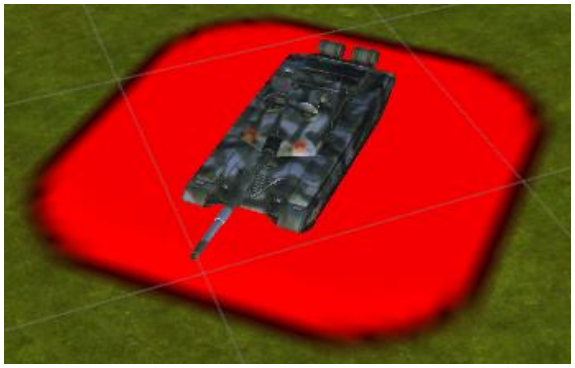

Fig. 50: *Sprite* d'une unité pour l'affichage de sa position sur la mini-carte

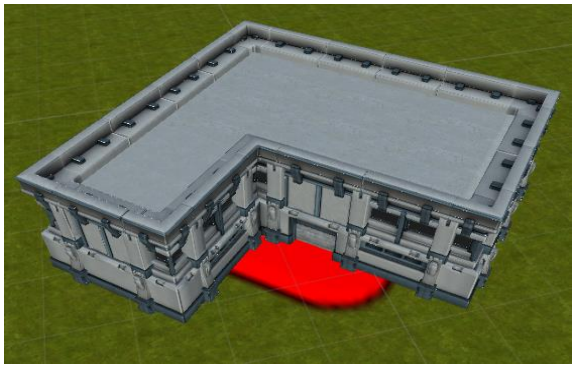

Fig. 51: *Sprite* du bâtiment pour l'affichage de sa position sur la mini-carte

Un script appelé « *MiniMapManager.cs* » a ensuite été créé afin d'appliquer les dimensions du terrain à la caméra employée pour la mini-carte à l'initialisation du jeu. Ainsi, il est possible de s'assurer de capturer l'intégralité du terrain avec la mini-carte même si celui-ci est ultérieurement redimensionné. Par contre, l'application des dimensions à la caméra se fait avec la valeur maximale entre la largeur et la longueur du terrain de sorte à garder l'aspect carré de la caméra lorsqu'elle capture le terrain. Cela permet de respecter deux propriétés simultanément, soit de ne pas disproportionner les éléments affichés par la mini-carte en fonction du ratio largeur-longueur du terrain, et d'effectivement afficher le terrain comme rectangulaire sur la mini-carte lorsqu'il l'est, plutôt que d'afficher de manière erronée qu'il serait carré si la caméra était redimensionnée en suivant le ratio d'aspect du terrain.

Afin de transférer les images capturées par la caméra vers le *Canvas* pour que la mini-carte soit toujours affichée dans le coin supérieur gauche de l'écran, un élément *Raw Image* est employé avec une texture de type « *RenderTexture* » qui est simultanément appliquée au champ « *Target Texture* » de la caméra (Fig. 52) et au champ « *Texture* » de l'image sur le *Canvas* (Fig. 54). Puisque, par défaut, cette texture remplace les pixels de fond par des pixels transparents (Fig. 54), une image supplémentaire a été ajoutée pour simuler la région hors du terrain sur la mini-carte. Cette dernière est en arrière de l'image possédant la « *RenderTexture* » afin que celle-ci affiche le terrain là où il existe et que les parties hors terrain transparentes fassent ressortir l'image positionnée en arrière.

Puis, les deux images sont regroupées sous un *GameObject* possédant un *RectTransform* dont les positions des points d'ancrage pour redimensionner la mini-carte avec le *Canvas* dynamique en fonction de la taille d'écran sont soigneusement placées dans les coins de la mini-carte, tel que présenté à la Fig. 53. La mini-carte est donc redimensionnée selon l'écran, mais la forme du terrain est tout de même bien affichée à l'écran.

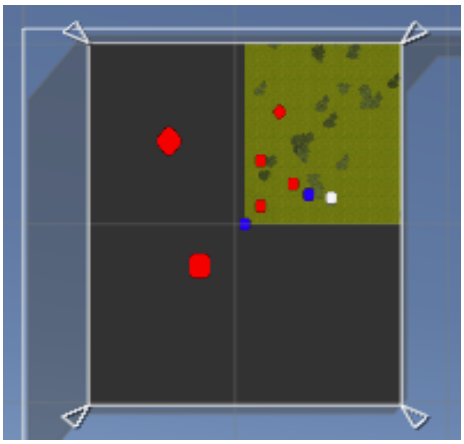

Fig. 53: Affichage de mini-carte sur le *Canvas* à l'aide de la caméra (mode édition avec terrain décentré)

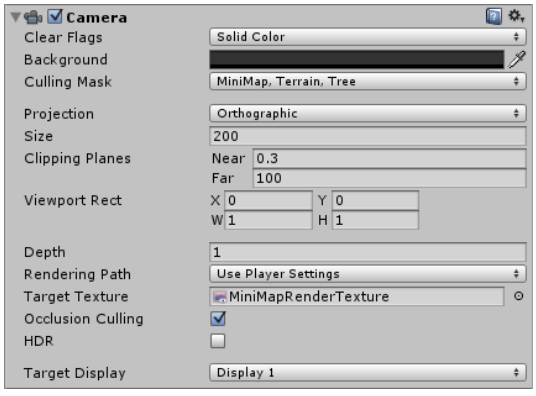

Fig. 52: Texture de rendu de la caméra « *MiniMapCamera* » pour transférer les images

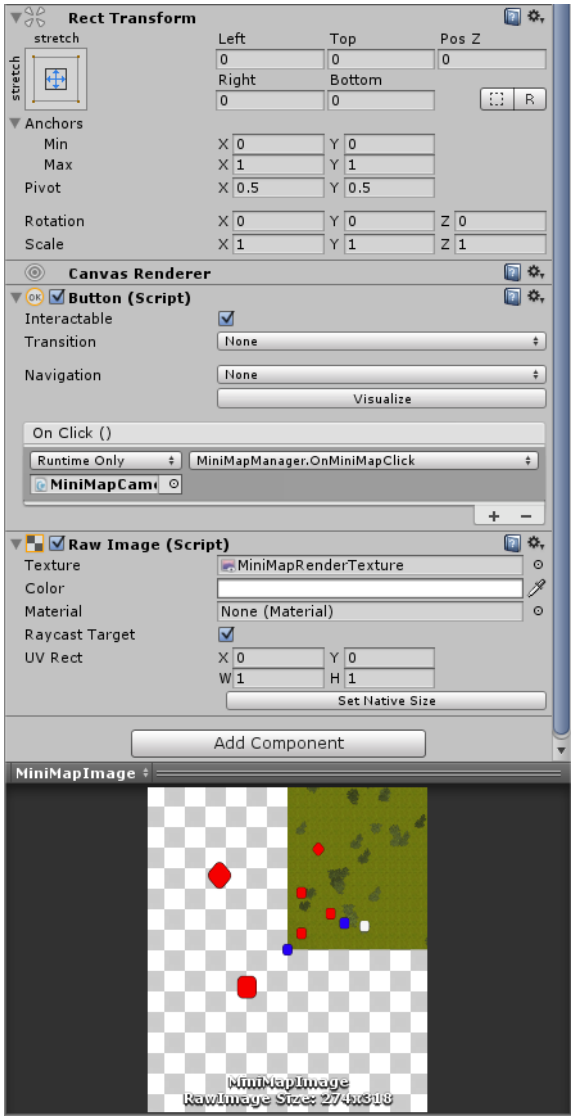

Fig. 54: Éléments du *Canvas* pour la mini-carte

Avec la mini-carte et sa caméra en place, il ne restait plus qu'à accomplir les déplacements selon la position des clics sur la carte. Pour ce faire, un élément *Button* est employé tel que montré à la Fig. 54. Celui-ci est lié à la fonction « *OnMiniMapClick* » située dans le script « *MiniMapManager.cs* » précédemment lié au *GameObject* contenant la caméra de la mini-carte. Lorsque l'évènement de clic de souris est détecté, la fonction est automatiquement appelée et celle-ci se charge de faire les déplacements sur le terrain. Afin d'obtenir la position correspondante entre le clic et le terrain, plusieurs ajustements sont nécessaires. Tout d'abord, puisque la mini-carte peut être redimensionnée à l'écran selon le ratio du *Canvas*, la position XY du clic est obtenue en pixels et est normalisée par les quatre coins de l'image de la mini-carte aussi en pixels. Puisque cette normalisation ramène les coordonnées entre [0,1], mais que le terrain est centré en (0,0) et possède des coordonnées tant positives que négatives, les coordonnées normalisées sont ramenées en dimensions de terrain en multipliant par la largeur/longueur moins la moitié de la largeur/longueur selon l'axe calculé. Finalement, puisque la caméra affiche le terrain sous un aspect carré pouvant dépasser ses réelles limites, les coordonnées de terrain obtenues doivent être encore une fois ajustées en fonction de son ratio largeur-longueur. Cette position finale est ensuite appliquée en XZ à la caméra *RTS* accomplissant l'affichage principal du jeu, mais sa valeur en Y est laissée intacte. Les formules suivantes présentent en détail la procédure décrite plus haut.

$$\begin{cases} MiniMapX_{norm} = \dfrac{MiniMapX - MiniMapX_{min}}{MiniMapX_{max} - MiniMapX_{min}} \\ MiniMapY_{norm} = \dfrac{MiniMapY - MiniMapY_{min}}{MiniMapY_{max} - MiniMapY_{min}} \end{cases} \quad (1)$$

$$\begin{cases} TerrainX_{caméra} = MiniMapX_{norm} \cdot TerrainW - TerrainW / 2 \\ TerrainZ_{caméra} = MiniMapY_{norm} \cdot TerrainH - TerrainH / 2 \end{cases} \quad (2)$$

$$Terrain_{ratio} = TerrainW / TerrainH \quad (3)$$

$$\begin{cases} TerrainX_{réel} = TerrainX_{caméra} / Terrain_{ratio} \\ TerrainZ_{réel} = TerrainZ_{caméra} \end{cases} \text{ si } Terrain_{ratio} < 1$$

$$\begin{cases} TerrainX_{réel} = TerrainX_{caméra} \\ TerrainZ_{réel} = TerrainZ_{caméra} \cdot Terrain_{ratio} \end{cases} \text{ si } Terrain_{ratio} > 1 \quad (4)$$

$$\begin{cases} TerrainX_{réel} = TerrainX_{caméra} \\ TerrainZ_{réel} = TerrainZ_{caméra} \end{cases} \text{ si } Terrain_{ratio} = 1$$

$$\begin{cases} CaméraRTS_X = TerrainX_{réel} \\ CaméraRTS_Z = TerrainZ_{réel} \end{cases} \quad (5)$$

La vidéo « *minimap-display-interaction.mp4* » démontre les principes introduits avec cet objectif selon diverses dimensions de terrain. La mini-carte de dimension dynamique selon le terrain et l'écran est présentée, avec le remplissage des bordures hors terrain, les icônes d'unités et de bâtiments ajoutés au terrain et le déplacement à la position spécifiée.

<u>Objectif :</u> Sélection par région de multiples unités

<u>Description :</u>
Cet objectif consiste à fournir une nouvelle méthode de sélection des unités en spécifiant une région de sélection rectangulaire, comme supporté dans un très vaste ensemble d'applications graphiques. Toutes les unités situées à l'intérieur de la région spécifiée seraient alors sélectionnées comme si elles avaient été manuellement sélectionnées une à une. L'effet attendu est présenté à la Fig. 55.

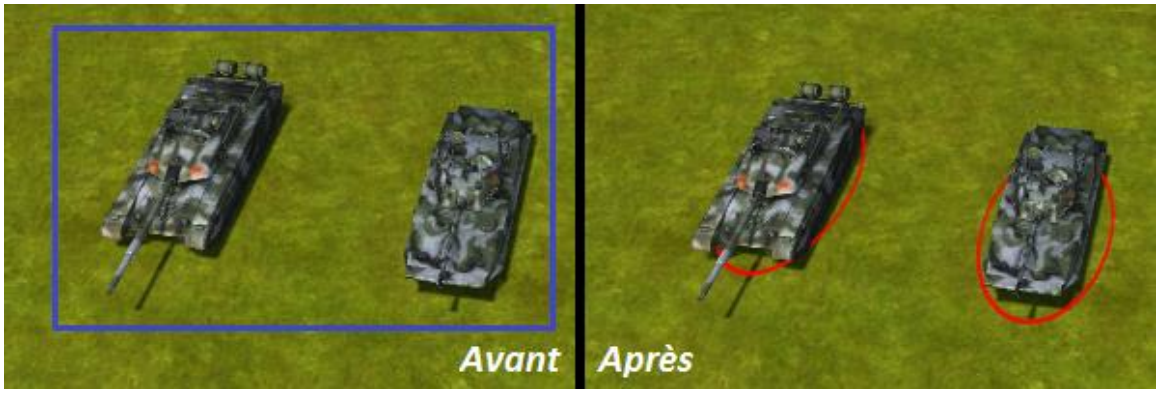

Fig. 55: Affichage de la région utilisée pour la sélection de plusieurs unités

<u>Réalisation :</u>
Afin d'accomplir l'affichage de la région de sélection, une image complètement blanche ayant des bordures opaques et un centre partiellement transparent, comme présenté à la Fig. 56, a été importée dans *Unity*. Cette image a été ajoutée au *Canvas* avec un nouveau *GameObject* appelé « *SelectionBox* » et qui contient un script *Image* tel que présenté à la Fig. 57. Les paramètres « *Sliced* » et « *Fill Center* » appliqués sont nécessaires afin de permettre un redimensionnement de cette image seulement en ajustant la surface transparente du centre sans toucher aux bordures, sans quoi celles-ci seraient également redimensionnées, ce qui donnerait un effet visuel indésirable.

Fig. 56: Image servant à l'affichage de la région de sélection sur le *Canvas*

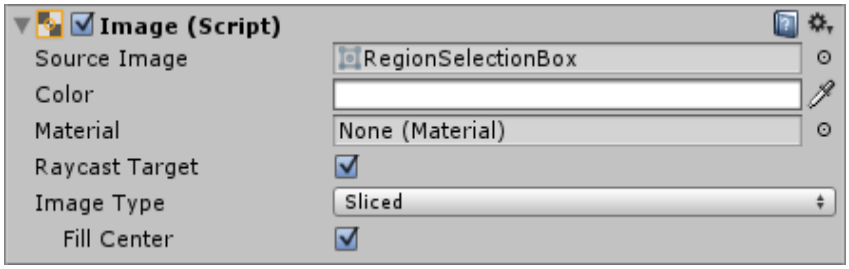

Fig. 57: Paramètres de l'image permettant le redimensionnement adéquat

Afin de spécifier quelle est la région du centre qui peut être redimensionnée par rapport aux bordures qui doivent demeurer fixes, l'éditeur de *Sprites* est employé pour déplacer les valeurs de bordures sur l'image importée, tel que présenté à la Fig. 58.

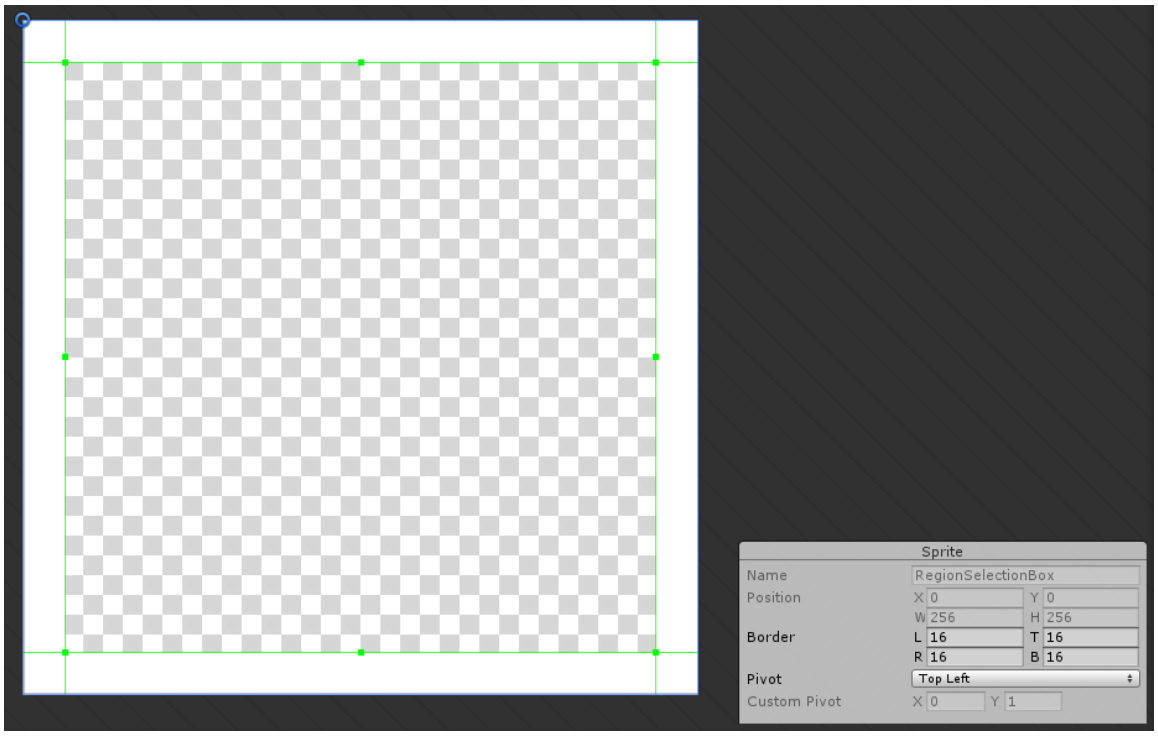

Fig. 58: Ajustement des bordures de l'image employée pour la région de sélection avec le *Sprite Editor*

Un dernier ajustement nécessaire par rapport à l'image était son point de pivot. Lorsque le pivot était laissé aux valeurs par défaut (0.5,0.5), la boîte de sélection ne réagissait pas tel que souhaité. Par contre, en plaçant le pivot à (0,0), tout fonctionne bien dorénavant.

Pour accomplir l'affichage de la région spécifiée par l'usager à l'écran, tant que le clic de souris effectuant la sélection d'unités (selon le paramètre d'entrée) est maintenu enfoncé et est capté par « *GetKey* », la fonction « *DrawSelectionRegion* » disponible dans le script « *SelectorManager.cs* » est appelée. Celle-ci emploie la position de la souris enregistrée temporairement lors du début du clic obtenu par « *GetKeyDown* » ainsi que la position actuelle afin de déterminer différence en X et Y (dimensions de la boîte de sélection). Ces dimensions sont ensuite ajustées indépendamment pour être amenées en valeurs positives dans le cas où elles seraient négatives, car la position actuelle de la souris est plus à gauche et/ou au-dessus de la position de départ. Dans un tel cas, la position de départ de la souris est adéquatement décalée des valeurs ajustées dans les deux axes pour obtenir le même résultat visuellement. Cet ajustement est nécessaire afin de pouvoir appliquer les dimensions au *RectTransform* de l'image qui n'accepte que des valeurs positives. De plus, les valeurs calculées sont redimensionnées en fonction de la grandeur d'échelle locale et actuelle du *Canvas* étant donné que celui-ci s'ajuste continuellement selon la dimension de la fenêtre du jeu. Sans ce dernier ajustement, le calcul de la dimension et position de la boîte de sélection paraîtrait adéquat tant que la fenêtre ne serait pas redimensionnée, mais son affichage produirait des résultats très erronés dès que la taille de l'espace d'affichage serait modifiée pendant que le jeu s'exécute. Le code employé pour accomplir cet affichage est en grande partie inspiré des instructions indiquées sur un forum de la communauté d'*Unity* qui permettait d'accomplir l'affichage recherché [7].

Pour réaliser le calcul des positions formant la région de sélection, une certaine partie de la méthodologie planifiée initialement a effectivement été employée, mais avec quelques ajustements. Ainsi, la position où le clic de souris est initialement enfoncé est effectivement captée avec « *GetKeyDown* » pour la mémoriser temporairement jusqu'à ce que la relâche du clic de souris soit captée par « *GetKeyUp* ». Par contre, toutes les méthodologies anticipées pour convertir ces positions n'ont pas été effectives. Ainsi, une autre approche a été effectuée en s'inspirant d'un concept de segmentation d'image vu dans un autre cours, et qui consiste à évaluer la bordure d'une région de forme quelconque.

Tout d'abord, les quatre coins formant la boîte de sélection dans l'espace 2D du *Canvas* sont convertis en positions 3D sur la surface du terrain, selon la même méthode employée à l'objectif « *Sélection d'unités et commande d'attaque* ». Lorsque convertis dans l'espace 3D, ces coins forment un quadrilatère quelconque en fonction de l'angle de la caméra et de la position de la boîte à l'écran (Fig. 59). Il suffit alors d'évaluer la position relative de chaque objet sélectionnable sur le terrain par rapport aux quatre segments formés par le quadrilatère. Si un objet se retrouve à droite des quatre segments testés individuellement dans le sens horaire en fonction des points adjacents qui les forment, l'objet est nécessairement à l'intérieur de la région. Cela est illustré à l'aide de la Fig. 60.

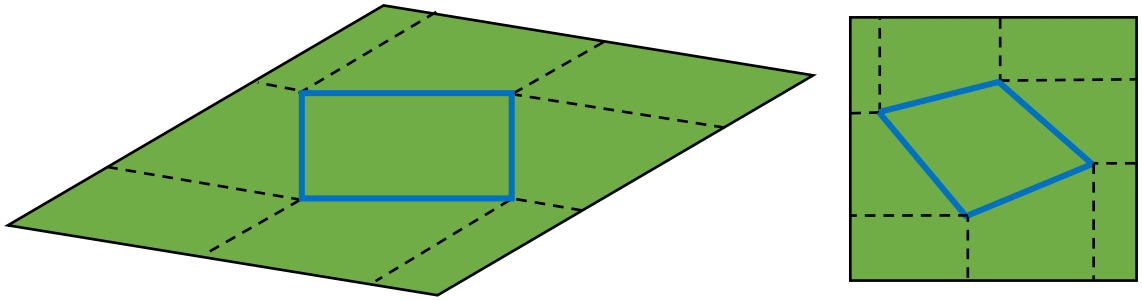

Fig. 59: Conversion de la boîte de sélection de l'espace 2D à 3D et visualisée à plat sur le terrain

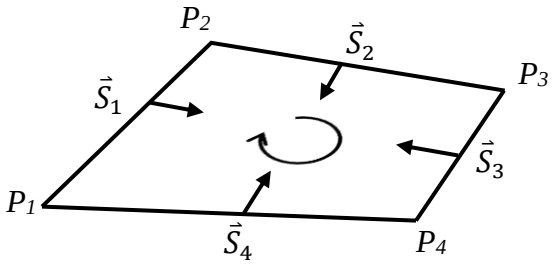

Fig. 60: Évaluation d'un objet contenu à l'intérieur des segments formant la zone de sélection 3D

Les objets sélectionnables sur le terrain sont simplement obtenus à l'aide de la fonction « *FindGameObjectsWithTag* » ainsi que des listes de *Tags* préalablement établies (Fig. 18). L'évaluation de la position relative des objets par rapport à chaque segment est réalisée avec la fonction « *PositionRelativeToLineSegment* » qui a été implémentée. Celle-ci retourne une direction d de -1 si l'objet évalué est à gauche d'un segment S tracé à l'aide de deux points P, +1 si l'objet est à sa droite et 0 si l'objet est directement sur le segment.

Pour évaluer la direction de l'objet par rapport au segment, la fonction effectue simplement les opérations suivantes :

$$\begin{aligned} \vec{S} &= P_{s,\,fin} - P_{s,début} \\ \vec{P} &= P_{objet} - P_{s,début} \\ \vec{V}_{\perp} &= \vec{S} \times \vec{P} \\ d &= \text{sign}\left(\vec{V}_{\perp} \cdot \vec{V}_{up}\right) \end{aligned} \tag{6}$$

Ainsi, tous les objets évalués qui retournent une valeur plus grande ou égale à zéro simultanément pour les quatre segments sont contenus dans la région et sont alors ajoutés à la liste d'objets actuellement sélectionnés. À partir de ce point, les procédures habituelles de sélection et d'affichage présentées à des objectifs précédents prennent la relève. Tout comme pour la sélection d'unités individuelle, la sélection par région permet de cumuler des unités sélectionnées par plusieurs traçages de régions de sélection en maintenant la touche *CTRL* enfoncée (valeur par défaut éditable).

La vidéo « *selection-box-precision-region-multiselect.mp4* » démontre l'application d'une sélection par région de plusieurs unités, suivie d'une sélection cumulative d'unités déjà sélectionnées en maintenant *CTRL* enfoncé. Puis, la vidéo présente la procédure d'affichage de la boîte de sélection demeurant opérationnelle, peu importe la direction de sélection effectuée en fonction des positions relatives de la souris à l'appui et à la relâche du bouton de sélection. Ensuite, la vidéo présente plusieurs sélections selon diverses rotations de caméra pour illustrer que le calcul de l'espace 2D vers le 3D s'accomplit adéquatement malgré les rotations et changements de zoom. Aussi, plusieurs sélections très rapprochées autour d'une unité sont effectuées stratégiquement pour démontrer que l'évaluation de la région bornée par le quadrilatère quelconque obtenu sur le terrain s'accomplit avec précision, encore une fois, pour illustrer que le calcul de l'espace 2D vers le 3D des coins formant les segments de la région est correctement effectué. La vidéo « *selection-box-resize-screen.mp4* » présente quant à elle diverses opérations de sélection par région appliquées suite à une réduction et à une augmentation de la taille de la fenêtre d'affichage. Dans les deux cas, l'affichage de la boîte et les calculs pour évaluer les régions s'adaptent tous deux automatiquement et immédiatement aux nouveaux ratios d'aspect, comme il l'est désiré.

Objectif : Affichage de la vie d'une unité

Description :
Lorsqu'une ou plusieurs unités sont sélectionnées, elles devront afficher une barre indiquant leur niveau de vie actuel légèrement au-dessus d'elles. Lorsque les unités seront endommagées, elles afficheront leur niveau de vie, peu importe si elles sont sélectionnées ou non. Le résultat recherché est d'avoir un affichage semblable à celui illustré à la Fig. 61 selon les conditions qui y sont mentionnées.

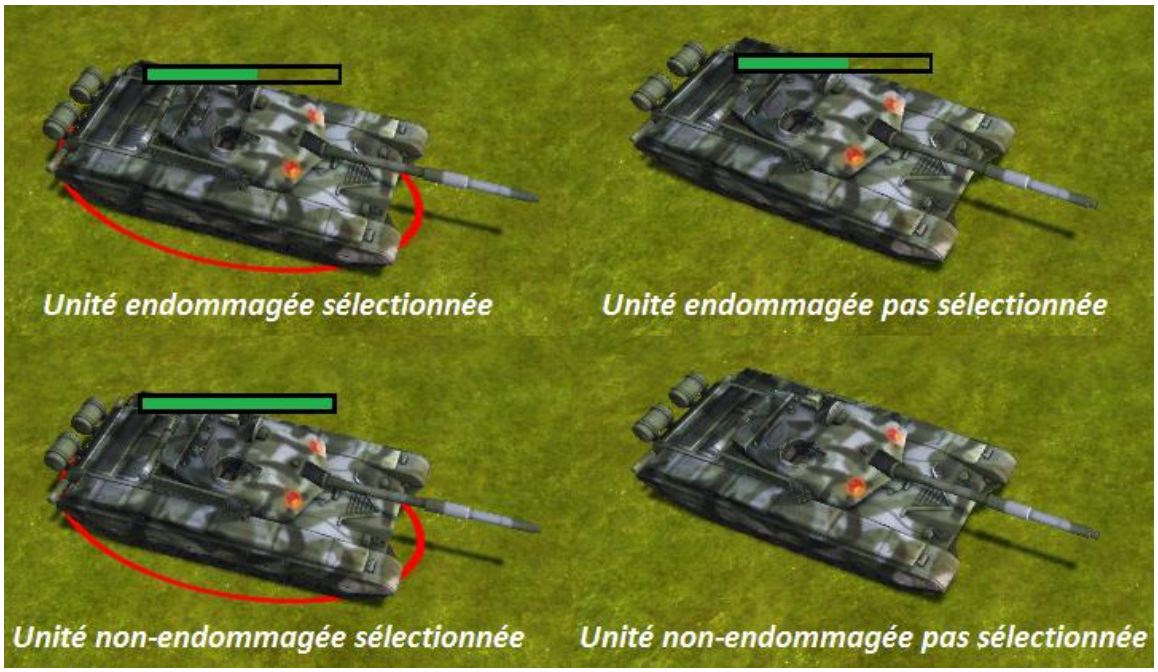

Fig. 61: Différents affichages attendus des niveaux de vie d'une unité

Réalisation :

Afin de réaliser cet objectif, l'*Asset* « *Progress Bar* » [34] disponible sur l'*Asset Store* a été employé tel que planifié. Celui-ci vient avec un *Shader* nommé « *AlphaProgress* » qui affiche une image selon un seuil de valeur alpha. L'idée est donc d'avoir une image représentant la barre de vie complète, telle qu'à la Fig. 62 qui est employée dans le cadre de ce projet, et d'avoir une seconde image sous forme de gradient de valeurs alpha employées comme masque pour afficher l'image en couleurs. À la Fig. 63, le masque est affiché en tons de gris pour des raisons d'affichage, mais les valeurs réelles sont en fait un gris de couleur constante ayant des niveaux de transparence allant de zéro à un linéairement de gauche à droite. Avec une valeur de seuil normalisée, il est possible de spécifier au *Shader* quelles valeurs de l'image couleur conserver en fonction des valeurs alpha contenues aux pixels correspondants du masque. L'affichage de la vie restante de l'unité avec ce *Shader* consiste donc à simplement appliquer le ratio entre une valeur de vie restante et la valeur totale qu'elle possède à sa création. Les paramètres de vie totale et de quantité de dommage effectué lors d'une attaque sont éditables avec l'interface éditeur du script « *UnitManager.cs* ». Lorsqu'une unité en attaque une autre, le dommage attribué est alors adéquatement déduit de la valeur restante de vie de l'unité attaquée avec le dommage de l'unité attaquante et le *Shader* applique l'affichage de la barre de vie automatiquement. L'utilisation de diverses valeurs de pointage de vie et de dommages est visible à travers plusieurs des vidéos précédemment présentés.

Fig. 62: Image de la barre de vie complète

Fig. 63: Image du masque de valeurs alpha

Pour effectuer un affichage visuel plus agréable et efficace de la barre de vie, une autre image est employée comme « conteneur » pour entourer la barre par une bordure noire, et ainsi mieux illustrer la quantité restante de vie par rapport à la valeur totale qui n'est pas directement visible étant donné que la barre devient progressivement transparente.

De plus, l'affichage de la barre est maintenu à dimension constante alors que la caméra est ajustée selon le zoom avant-arrière. Cela est un ajustement supplémentaire qui n'avait pas été prévu, mais qui rehausse grandement la qualité d'affichage de la barre de vie. En effet, initialement, les barres n'étaient pas maintenues à dimension constante, et le zoom éloigné du terrain les rendait très difficiles à percevoir. À l'inverse, les barres devenaient immenses à l'écran et commençaient à l'encombrer lorsque la caméra était très rapprochée des unités. Ainsi, un script nommé « *HealthBarManager.cs* » a été créé et est appliqué aux *GameObjects* qui contiennent la barre de vie appliquée à toutes les unités. Celui-ci ajuste la dimension d'une barre lorsqu'elle est visible à l'écran afin qu'elle préserve les mêmes hauteur et largeur, peu importe le zoom appliqué. Cela est accompli avec la procédure suivante tirée de [11] qui applique essentiellement le ratio de redimensionnement requis pour annuler l'ajustement de dimensionnement automatique à l'écran, ce qui préserve les dimensions constantes. Selon la valeur désirée de « *SizeOnScreen* », il est possible d'ajuster la dimension constante appliquée aux barres de vies.

```
private void UpdateLockedSizeOnScreen()
{
   var a = Camera.main.WorldToScreenPoint(transform.position);
   var b = new Vector3(a.x, a.y + SizeOnScreen, a.z);
   var aa = Camera.main.ScreenToWorldPoint(a);
   var bb = Camera.main.ScreenToWorldPoint(b);
   transform.localScale = Vector3.one * (aa - bb).magnitude;
}
```

En plus du redimensionnement le script s'assure de toujours déplacer la barre de vie à la position courante de l'unité associée, mais avec un décalage vers le haut ajustable par une constante au script. La barre est également tournée pour faire face à la caméra du joueur en tout temps grâce à l'appel « *Quaternion.LookRotation(Camera.main.tranform.forward)* ». Le script « *HealthBarManager.cs* » effectue aussi la conversion de la valeur de seuil alpha en un paramètre envoyé au *Shader* selon le ratio de vie qui lui est spécifié par le script « *UnitManager.cs* ». De plus, ce script accomplit automatiquement la gestion de la visibilité d'affichage de la barre à l'écran en fonction de la valeur de vie restante comme illustré avec les conditions de la Fig. 61. Lorsqu'une unité est maintenue sélectionnée, un paramètre supplémentaire présent dans ce script permet de surcharger l'état d'affichage en fonction de la vie restante pour imposer un affichage visible de la barre à l'écran.

L'affichage de la barre en fonction des modes de sélection ainsi que de la quantité de dommage reçu est illustré à travers plusieurs des vidéos disponibles. Une vidéo supplémentaire nommée « *health-bar-constant-size.mp4* » présente plus en profondeur le redimensionnement constant qui semble parfois trompeur à l'œil étant donné que tous les autres objets de la scène sont redimensionnés, ce qui peut laisser croire que la barre le fait aussi. Cette vidéo utilise une image de référence fixe appliquée sur le *Canvas* pour la comparer avec la largeur de la barre de vie en fonction du zoom appliqué.

Le résultat final obtenus pour l'affichage de la barre de vie d'une unité est présenté à l'aide des Fig. 64 et Fig. 65.

Fig. 64: Barre de vie pleine d'une unité visible seulement lorsque sélectionnée

Fig. 65: Barre de vie avec dommages d'une unité non sélectionné conservée toujours affichée

Objectif : Effet de poussière d'unité en mouvement

Description :
Cet objectif consiste à faire apparaître de la poussière ou de la fumée autour des chenilles ou des roues d'une unité en mouvement afin d'ajouter un peu plus de réalisme durant ses déplacements. L'effet pourra être appliqué légèrement autour et en arrière de l'unité selon ce qui semble visuellement mieux. La Fig. 66 offre une représentation approximative de l'effet recherché.

Fig. 66: Approximation de l'effet de poussière attendu d'une unité en mouvement

Réalisation :
Pour accomplir cet objectif, multiples *ParticleSystems* sont employés. Ceux-ci sont basés sur le tutoriel d'*Unity* pour simuler la fumée sortant d'un tuyau d'échappement d'un véhicule [31] ainsi que d'*Assets* disponibles comme point de départ pour la conception. Chaque ensemble de *ParticleSystem* nommé « *TaillingDust* » contient un effet de poussière linéaire pour chacun des deux côtés de l'unité là où les chenilles des tanks ou les roues de l'unité de construction sont situées. Ceux-ci constituent la fumée immédiatement visible sur les côtés des unités et clairement illustrés à la Fig. 67. Également, un nuage de fumée supplémentaire est généré presque au centre de l'unité (point plus éclairé à la Fig. 67) pour émettre la fumée visible à l'arrière de l'unité. Finalement, une autre émission de particules génère de petits nuages de fumée de manière aléatoire, qui eux, s'élèvent plus en hauteur que les autres fumées très concentrées localement, afin de donner l'impression que la fumée générée par les unités en mouvement se dissipe progressivement dans l'atmosphère.

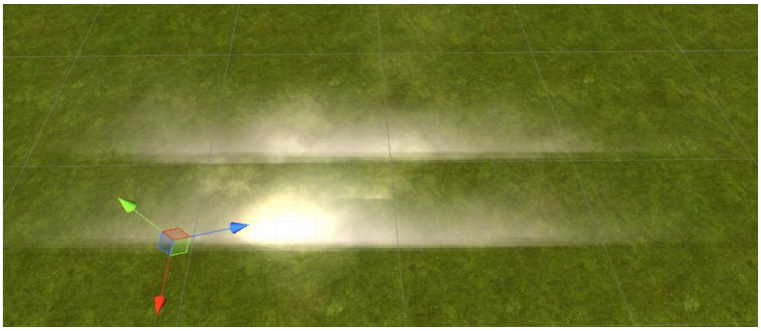

Fig. 67: Ensemble de *ParticleSystems* simulant l'effet de poussière

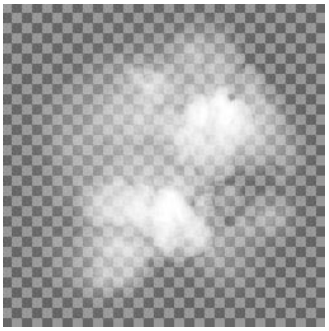

Fig. 68: *Sprite* de fumée employé

L'affichage des particules de fumées est accompli avec le même *Sprite* importé dans *Unity* et présenté à la Fig. 68, mais selon différents paramètres d'émission et de concentration. Pour gérer lorsque la fumée est affichée, cela est tout simplement effectué avec le script « *EffectManager.cs* » précédemment introduit à l'objectif « *Projectile et effet d'impact* ». Ainsi, lorsqu'une unité commence à se déplacer, une *Coroutine* accomplissant l'émission des particules en boucle continue est lancée et persiste jusqu'à ce que l'unité s'arrête ou qu'elle change de direction suite à une rotation. À ce moment, les *ParticleSystems* ne sont pas immédiatement désactivés sans quoi la fumée disparaîtrait d'un seul coup plutôt qu'en transition jusqu'à ce qu'elle termine de se dissiper, ce qui semblerait étrange. Donc, lorsque le mouvement linéaire se termine, l'émission de nouvelles particules est arrêtée, mais l'objet reste actif pour accomplir la fin de l'animation de la fumée déjà émise. Ainsi, le script « *EffectManager.cs* » se charge d'automatiquement déterminer lequel des systèmes de particule nécessite le plus de temps pour compléter son animation et continue à l'exécuter jusqu'à ce délai soit atteint. Seulement à ce moment, l'objet contenant l'ensemble de *ParticleSystems* est désactivé et rendu invisible.

De plus, comme on pouvait le voir avec la Fig. 47 qui présente tous les systèmes de particules instanciés pour une unique unité, cinq exemplaires de l'ensemble « *TraillingDust* » sont générés. Cela est accompli afin de pouvoir cycler entre les diverses instances créées de sorte à ce qu'il soit possible d'en afficher plus d'une simultanément. En effet, lorsque l'unité effectue une rotation et recommence immédiatement un mouvement dans une autre direction, les nuages de fumée générés lors de mouvements précédents n'ont souvent pas terminé de se dissiper. Ainsi, les mouvements d'unités enchaînés à petits intervalles faisaient que, avec un seul groupe de *ParticleSystems*, les nuages étaient tournés soudainement pour s'aligner avec l'unité. Pour résoudre ce problème, on utilise une duplication de « *TraillingDust* » dont les références sont maintenues dans une liste pour permettre de cycler à travers ceux-ci en maintenant un indice de l'instance actuellement active. Chaque nouveau mouvement nécessitant un nouvel affichage de fumée emploie l'instance active en boucle. L'indice est incrémenté lorsque des conditions de rotation sont détectées pour indiquer qu'un nouveau mouvement sera potentiellement effectué avec de la fumée pointant dans une autre direction. Puisque les animations sont réalisées avec « *EffectManager.cs* » qui se charge de les désactiver lorsqu'elles terminent, les effets de fumés peuvent être appelé en chaîne sans se soucier d'affecter les animations précédemment lancées. De plus, les *ParticleSystems* ont été ajustés de sorte à émettre la fumée dans l'espace *World* plutôt que local au *GameObject*

qui les contient afin de s'assurer que la fumée émise ne soit pas tournée si l'unité venait à tourner légèrement et que l'indice n'était pas adéquatement ajusté (à cause de l'angle de rotation permissif de l'objectif « *Déplacements de base des unités* »). Évidemment, si l'on dépasse plus de cinq mouvements différents par intervalles très rapprochés, le cycle ferait que le dernier effet actif serait rapidement réemployé malgré qu'il n'ait pas terminé son animation, mais, avec autant de mouvements consécutifs, l'unité n'a habituellement pas le temps de faire suffisamment de distance pour accomplir une grande émission de fumée. Ainsi, la limite de cinq instances est viable et rarement problématique visuellement.

L'affichage erroné des particules de fumée en employant une seule instance est présenté dans la vidéo « *unit-smoke-effect-single-problem.mp4* », alors que la bonne mise en application peut être visualisée à la vidéo « *unit-smoke-effet-multiple.mp4* ». Cette dernière présente également l'animation de chacun des *ParticleSystems* de fumée indépendamment.

Objectif : Destruction d'une unité

Description :
Lorsqu'une unité atteint un niveau de vie de zéro, une animation de destruction devra se produire. Plus spécifiquement, un petit effet d'explosion sera généré à partir de la position de l'unité afin d'illustrer sa destruction. Puis, après un petit délai, l'instance de l'unité devra disparaître pour qu'elle ne soit plus disponible. L'unité pourrait également être assombrie afin de renforcer l'effet d'explosion comme si elle se brûlait.

Réalisation :
La vérification de la valeur de zéro point de vie restant est effectuée dans la boucle de mise à jour du script « *UnitManager.cs* ». Lorsque cette condition survient, une *Coroutine* nommée « *DestroyUnit* » est appelée pour accomplir toute la séquence nécessaire à la destruction de l'unité. Toutes les unités qui l'attaquaient arrêtent automatiquement de tirer sur celle-ci étant donné que la condition de vie supérieure à zéro est également vérifiée avant de lancer les projectiles. L'unité est aussi désélectionnée automatiquement selon la même vérification accomplie dans « *SelectorManager.cs* ».

La fonction « *DestroyUnit* » débute par fixer l'unité à sa position courante dans le cas où elle était en mouvement. Elle supprime ensuite toutes les références pouvant pointer vers une unité à attaquer. Puis, la couleur de tous les matériaux appliqués à l'unité est assombrie en appliquant une procédure similaire à celle employée pour teindre le bâtiment à l'objectif « *Rotation et positionnement de bâtiments* », mais cette fois-ci en employant la valeur multiplicative $(0.9,0.9,0.9)_{RGB}$, afin de produire l'impression d'unité brûlée par les flammes. Les effets d'explosion, de feu, de flammèches, de fumée et plusieurs autres sont ensuite affichés avec un ensemble de *ParticleSystems* nommé « *ExplosionAndFire* ». L'ensemble des *Sprites* impliqués dans la réalisation de toute la séquence d'animation de ces effets de particules est présenté à la Fig. 69.

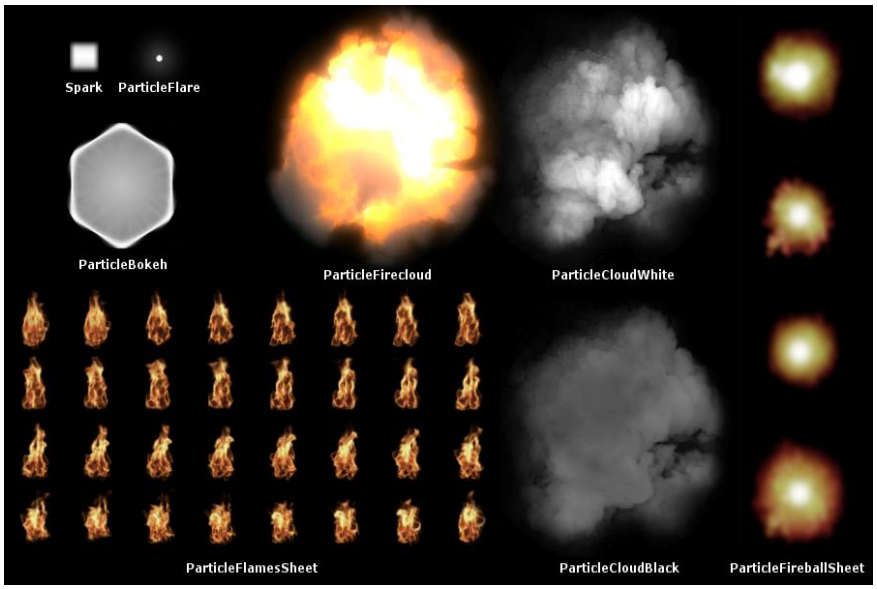

Fig. 69: Regroupement de tous les *Sprites* utilisés pour l'effet de destruction

Fig. 70: Liste exhaustive des *ParticleSystems* employés dans l'effet « *ExplosionAndFire* »

Parmi les *Sprites* présentés à la Fig. 69, certains sont employés selon une séquence d'animation telle que les « feuilles » de flammes, alors que d'autres consistent à une réémission de plusieurs duplications des mêmes particules en boucle comme dans le cas de la fumée et des *Sparks*. Aussi, certains des *Sprites* sont réutilisés à travers plusieurs *ParticleSystem*s distincts pour réaliser différents effets le long de la séquence, comme afficher la fumée de la flamme suite à l'explosion ainsi qu'utiliser la même fumée lors de l'explosion, mais selon une forme d'émission différente. L'ensemble des *ParticleSystems* employés et spécifiés comme enfants à l'animation « *ExplosionAndFire* » sont présentés à la Fig. 70. Ceux-ci sont encore une fois gérés avec le script « *EffectManager.cs* » afin que tous les *ParticleSystems* aient le temps de compléter leur animation intégralement avant de poursuivre la séquence de commandes pour la destruction de l'unité.

Lorsque la séquence d'animation des explosions et flammes est complétée, la fonction « *DestroyUnit* » déplace l'unité progressivement à travers le terrain avec une translation « *MoveToward* » le long de l'axe Y afin de faire disparaître l'unité. Cette translation offre un effet visuel plus intéressant que de simplement faire disparaître l'unité soudainement. De plus, cet effet a souvent été observé dans plusieurs jeux *RTS*, donc il a été répliqué dans ce projet selon ses sources d'inspiration. Lorsque la translation de l'unité sous le terrain atteint un certain décalage spécifié en paramètre d'entré au script « *UnitManager.cs* », la grille de *Nodes* est remise à l'état inoccupé là où l'unité était positionnée afin de libérer cet espace pour le déplacement des autres unités. Ensuite, les instances des *ParticleSystems*, projectile ou tout autre élément chargé en mémoire pour cette unité sont supprimés. Finalement, l'instance de l'unité même est supprimée.

La vidéo intitulée « *unit-destruction-effet-sequence-instance-cleanup.mp4* » présente toute la séquence présentée ainsi que le menu hiérarchique des instances de l'unité qui sont supprimées à la fin de la séquence. Seulement deux unités sont disponibles sur le terrain dans cette vidéo afin de mieux visualiser le nettoyage des instances libérées.

3. Discussions

3.1. Limitations de l’application

Première limitation

Le modèle du terrain est limité à un unique exemplaire parce que plusieurs terrains nécessitent autant de re-designs que de terrains différents désirés. Donc, le besoin de modélisation de nouvelles formes, l'application de textures diverses et positionnées adéquatement, ainsi que l'application des transformations aux sous-éléments compris dans le graphe de scène du terrain sont toutes des tâches manuelles demandant beaucoup de travail et de temps. Ainsi, l'édition des objets présents dans la scène, de sorte à produire un résultat suffisamment réaliste, requiert beaucoup d'ajustements minutieux. Cela est un exemple de limitation majeure par rapport à la diversification des éléments pouvant être intégrés dans toute application d’infographie. Malgré que certains aspects pourraient être générés synthétiquement à l'aide de scripts avancés, comme avec le générateur de terrain proposé dans [22], cela nécessiterait un niveau beaucoup plus poussé d'automatisation qui implique plusieurs d'ajustements graphiques avancés. Aussi, des méthodes automatisées pour générer des terrains demanderaient beaucoup de tests opérationnels afin de s'assurer que toutes les composantes impliquées dans le graphe de scène du terrain interagissent correctement entre elles et qu’elles produisent des résultats suffisamment réalistes.

Deuxième limitation

En infographie, le positionnement de la caméra est un aspect majeur à considérer pour bien visualiser les objets modélisés selon les effets recherchés. Cela est vrai si la caméra est fixe, mais l'est d'autant plus si l'on permet un déplacement de la caméra par l'utilisateur. Les limitations très rigides de déplacements de la caméra dans l'application réalisée sont un excellent exemple de limitations volontaires et primordiales à son bon fonctionnement. Sans le contrôle de sa position dans le monde en fonction de la hauteur et des bordures de la carte, la caméra pourrait être déplacée de diverses manières qui seraient très contraignantes parce que l'espace de jeux pourrait ne plus être visible. Aussi, la limitation de la rotation selon certains axes est requise afin de s'assurer que la caméra n'effectue pas d'enroulement sur elle-même, ce qui rendrait sont contrôle peu aisé. Finalement, le fait que la caméra ne soit aucunement limitée à traverser des objets en infographie peut poser beaucoup de problèmes. Il suffit de penser à l'effet de « vide » observé lorsque l'on traverse un mur ou que l'on entre à l'intérieur d'un objet dans un jeu, ce qui détruit l'immersion et le réalisme du jeu. Ce problème est une grande limitation par rapport au contrôle de la caméra. Dans le cas de l'application réalisée, lorsque la caméra est déplacée pour produire le zoom, les limitations volontaires en hauteur utilisées permettent d'éviter à notre avantage ces problèmes parce que l'on empêche simplement d'atteindre les objets pouvant être traversés. Par contre, les limitations de rotation de la caméra par rapport aux axes X et Z du terrain font qu’il est beaucoup plus difficile d’observer des éléments éloignés sur le terrain, car la vue à 45° vers le bas impose une région très restreinte de vision. Cette dernière limitation pourrait donc être moins appréciée par certains utilisateurs.

3.2. Extensions possibles

Première extension

Le jeu ne fait actuellement pas usage d'un éclairage très avancé. Grâce aux calculs d'éclairage accomplis automatiquement par *Unity* par rapport à la source directionnelle qui a temporairement été positionnée en angle au-dessus du terrain, il est possible d'obtenir des ombrages de base à partir des unités et des bâtiments. Par contre, il serait possible d'obtenir encore plus de réalisme si ces ombres s'affichaient dynamiquement selon un mouvement de la source pour simuler le déplacement naturel du soleil à travers la journée. De plus, il serait possible d'intégrer des cycles de jour et nuit pour ajouter de la diversification sur le terrain, ce qui nécessiterait aussi un contrôle d'éclairage plus avancé. Par contre, ajouter de l'ombrage dynamique nécessiterait un grand nombre d'améliorations procédurales étant donné que, même pour des ombres fixes et simples lorsque les arbres ont initialement été appliqués sur le terrain, il a rapidement été observé à quel point l'application avait de la difficulté à rafraîchir le tout considérant tous les calculs d'éclairage impliqués par chacun des arbres, branches et feuilles.

Afin de réaliser des procédures pouvant simuler le mouvement du soleil et ajuster dynamiquement l'éclairage et les ombrages, plusieurs tutoriels pourraient être consultés comme « *Mini Tutorial - Day And Night Cycle - Beginner Tutorial* » [33], « *Working with physically-based shading: a practical approach* » [15], ainsi que plusieurs autres disponibles à travers la vaste documentation d'*Unity*.

Deuxième extension

Actuellement, le jeu est directement chargé avec le terrain existant et les quelques unités pré-positionnées sur celui-ci. Cela limite grandement la diversité du jeu et sa convivialité d'utilisation pour générer des scénarios de jeu particuliers. Ainsi, dans le contexte d'un jeu *RTS* complet, une interface de menus serait requise afin d'accueillir le joueur dans le jeu et lui offrir la chance d'éditer les paramètres de configuration et de contrôle de souris et de clavier. De plus, une bonne interface de menus devrait permettre des options standard tels la sauvegarde de partie, le chargement d'une partie sauvegardée, et finalement, la mise en pause et la reprise d'une partie courante. Alors, une extension évidente et primordiale pour cette application serait de supporter l'ensemble de ces fonctionnalités et de les rendre accessibles à l'utilisateur avec des menus et boutons d'interface pour naviguer d'une page à l'autre.

Afin de réaliser les interfaces de menus dans *Unity*, une ou plusieurs scènes indépendantes à celle du terrain et contenant des *Canvas* correspondants à chacun des menus désirés seraient à réaliser. Ces scènes contiendraient uniquement les *Canvas* et l'ensemble d'éléments 2D requis comme des boutons pour capter les commandes de navigation entre les menus, des *Sprites* pour l'arrière-plan du menu, du texte pour afficher certaines informations et des boîtes d'entrées (*InputBox*) pour capter les configurations indiquées par le joueur. Les scènes pourraient alors être chargées avec la fonction « *LoadScene* » lorsque nécessaire. En séparant le tout en scènes distinctes, il deviendrait possible de charger seulement les menus requis, lorsqu'opportuns, tout en évitant d'avoir plusieurs

duplications des *Canvas* de menus dans le cas où différents types de terrains deviendraient disponibles.

La grande majorité des scripts réalisés dans ce projet qui nécessitent une interaction humaine pour contrôler la caméra, capter la sélection d'objets ou envoyer des commandes aux unités emploient des paramètres éditables en entrées aux scripts à partir de variables publiques visibles dans l'éditeur d'*Unity*. Ainsi, il ne serait pas extrêmement difficile de rediriger la configuration des paramètres de contrôle spécifiés par l'usager via un menu vers chacun des scripts correspondants. Cela permettrait au joueur de profiter d'une expérience interactive personnalisée. Par contre, plusieurs tests seraient nécessaires de sorte à valider que toutes les fonctionnalités demeurent opérationnelles selon les valeurs de configuration indiquées. Ces paramètres d'entrée devraient également être très bien contrôlés afin que seulement des valeurs réalistes soient appliquées, par exemple, en empêchant une duplication de clés de clavier pour des commandes différentes.

Finalement, le chargement et la sauvegarde de scènes pour différentes parties pourraient aussi être gérés à partir de menus d'interface. Ceux-ci devraient être capables d'effectuer des lectures de fichier sur le disque pour charger ou sauvegarder adéquatement les parties. Des boutons et des listes déroulantes seraient requis pour capter les opérations à effectuer et afficher les fichiers sur lesquels les appliquer. Les concepts reliés aux menus seraient ici similaires à ce qui a été mentionné plus haut selon des scènes et *Canvas* indépendants, tandis que la gestion de données pour charger ou sauvegarder la partie nécessiterait davantage de recherche pour sérialiser les éléments de la scène pouvant changer d'une partie à l'autre, soit les unités et bâtiments instanciés.

4. Références

[1] "3dregenerator". Bulldozer B10 [Online]. Available: http://tf3dm.com/3d-model/bulldozer-b10-60330.html

[2] "3dregenerator". General Dynamics M1A2 TUSK [Online]. Available: http://tf3dm.com/3d-model/puo-34209.html

[3] "3dregenerator". Norinco 15545 Sp Tempered 3d model [Online]. Available: http://tf3dm.com/3d-model/puo-58697.html

[4] "3dregenerator". United Defense M2 Bradley Statesman IFV [Online]. Available: http://tf3dm.com/3d-model/puo-58657.html

[5] "azlyirnizam". Abrams Tank 3d model [Online]. Available: http://tf3dm.com/3d-model/abrams-tank-17774.html

[6] "bobobobo". (2013, Accessed: 2016-06-26). *In A star, how does the heuristic help determine your path?* Available: http://gamedev.stackexchange.com/questions/61850/in-a-star-how-does-the-heuristic-help-determine-your-path

[7] "Korindian". (Accessed: 2013-06-25). *RTS Style Drag Selection Box*. Available: http://forum.unity3d.com/threads/rts-style-drag-selection-box.265739/

[8] "north". Dragon Lance («Type-99»/WZ-213 LT) 3d model [Online]. Available: http://tf3dm.com/3d-model/dragon-lance-type-99wz-213-lt-82195.html

[9] "north". Iron Dove AA (PGZ-95 AA) 3d model [Online]. Available: http://tf3dm.com/3d-model/iron-dove-aa-pgz-95-aa-30877.html

[10] "north". Iron Mountain (Type-99 MBT) 3d model [Online]. Available: http://tf3dm.com/3d-model/iron-mountain-heavy-tank-type-99-84308.html

[11] T. Bigham. (2015, Accessed: 2016-07-05). *With a perspective camera: Distance independent size gameobject, With a perspective camera: Distance independent size gameobject*. Available: http://answers.unity3d.com/questions/268611/with-a-perspective-camera-distance-independent-siz.html

[12] A. J. N. Champandard, Alex. (2016, Accessed: 2016-07-09). *Lazy Theta*: Faster Any-Angle Path Planning*. Available: http://aigamedev.com/open/tutorial/lazy-theta-star/

[13] GodlessReason. (2015, Accessed: 2016-05-18). *Unity Tutorial: Building Placement (RTS) for Mobiles*. Available: https://www.youtube.com/watch?v=v7o7cjFqvrY, https://www.assetstore.unity3d.com/en/#!/content/45166

[14] Holistic3d. (2016, Accessed: 2016-06-13). *A Simple GUI Inventory, Object Pickup and Respawn in Unity 5*, [Video]. Available: https://www.youtube.com/watch?v=CHUOprBocoY

[15] E. Körner. (2015, Accessed: 2016-05-18). *Working with physically-based shading: a practical approach, Working with physically-based shading: a practical approach*. Available: http://blogs.unity3d.com/2015/02/18/working-with-physically-based-shading-a-practical-approach/

[16] S. Lague. (2015, Accessed: 2016-05-18). *A* Pathfinding Tutorial*, [Video]. Available: https://www.youtube.com/playlist?list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW

[17] S. Lague. (2013, Accessed: 2016-05-21). *Unity Tutorial: Building Placement*, [Video]. Available: https://www.youtube.com/watch?v=OuqThz4Zc9c

[18] Metalstache. (2012). *Unity RTS Binary: Selecting Individual Units*, [Video]. Available: https://m.youtube.com/watch?v=69OlSjOCwYY

[19] A. Nash. (2010, Accessed: 2016-07-08). *Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments, Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments*. Available: http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/

[20] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-Angle Path Planning on Grids," in *Proceedings of the national conference on artificial intelligence*, 2007, p. 1177.

[21] A. Nash, S. Koenig, and C. Tovey, "Lazy Theta*: Any-angle path planning and path length analysis in 3D," in *Third Annual Symposium on Combinatorial Search*, 2010.

[22] D. Pahunov. MapMagic World Generator [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/56762

[23] Rakshi Games. Realistic Tree 9 [Rainbow Tree] [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/54622

[24] M. Shalabai. Modular Abandoned Slaughterhouse: Lite [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/58082

[25] D. Sylkin. RTS camera (RTS camera), Version 1.0, [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/43321.

[26] Unity Answers. (Accessed: 2016-05-23). *Calculating ball trajectory in full 3d world*, [Online Forum]. Available: http://answers.unity3d.com/questions/248788/calculating-ball-trajectory-in-full-3d-world.html

[27] Unity Answers. (Accessed: 2016-05-23). *Changing an object's transparency during runtime*, [Online Forum]. Available: http://forum.unity3d.com/threads/changing-an-objects-transparency-during-runtime.46134/

[28] Unity Answers. (Accessed: 2016-05-23). *How do I attach a Camera to a GUI (for a Mini map)*, [Online Forum]. Available: http://answers.unity3d.com/questions/14253/how-do-i-attach-a-camera-to-a-gui-for-a-mini-map.html#answer-14261

[29] Unity Answers. (Accessed: 2016-05-23). *How to make enemy Cannonball fall on moving target position*, [Online Forum]. Available: http://answers.unity3d.com/questions/145972/how-to-make-enemy-canon-ball-fall-on-mooving-targe.html

[30] Unity Answers. (Accessed: 2016-05-23). *Shooting a cannonball*, [Online Forum]. Available: http://answers.unity3d.com/questions/148399/shooting-a-cannonball.html

[31] Unity Technologies. (2016, Accessed: 2016-05-23). *Exhaust Smoke from a Vehicle, Unity Documentation - Particle System*. Available: http://docs.unity3d.com/Manual/PartSysExhaust.html

[32] Unity Technologies. (2016, Accessed: 2016-05-23). *A Simple Explosion, Unity Documentation - Particle System*. Available: http://docs.unity3d.com/Manual/PartSysExplosion.html

[33] J. Vegas. (2015, Accessed: 2016-05-18). *Unity 5 Mini Tutorial - Day And Night Cycle - Beginner Tutorial*, [Video]. Available: https://www.youtube.com/watch?v=K_F9aYkEBtc

[34] C. Wang. Progress Bar (Progress Bar), Version 1.0, [Online]. Available: https://www.assetstore.unity3d.com/#!/content/34493.

[35] Zombiegons. Modern : Concrete Building Exterior 01 [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/2088

[36] 박정의. (2015, Accessed: 2016-06-26). *High school math level computer game AI pathfinding*. Available: https://hermit1004computer.blogspot.ca/2015/01/blog-post_89.html