RAPPORT Age of Cheap Empire

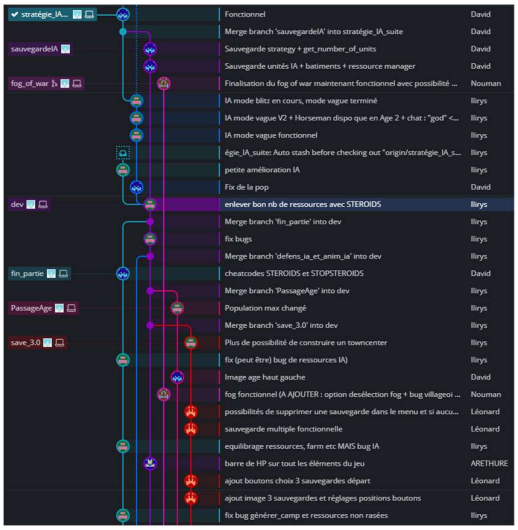
3A STI TD2 – Groupe 6

Nouman HAMDI
Hamza JAAIT
David LEGRAND
Arthur MANS
Lilian MARIÉ
Léonard MAURICE

INTRO

Voici notre rapport sur la copie du jeu Age of Empire. Ce dernier, sorti en 1997 par Microsoft, fait partie des jeux dits de stratégie en temps réel. Selon Wikipédia, « L'action du jeu se déroule dans un contexte historique, sur une période comprise entre 5000 av. J.-C. et 800 ap. J.-C. au cours de laquelle le joueur doit faire évoluer une civilisation antique de l'âge de la pierre à l'âge du fer pour débloquer de nouvelles technologies et unités lui permettant de bâtir un empire. »

PRESENTATION GENERALE

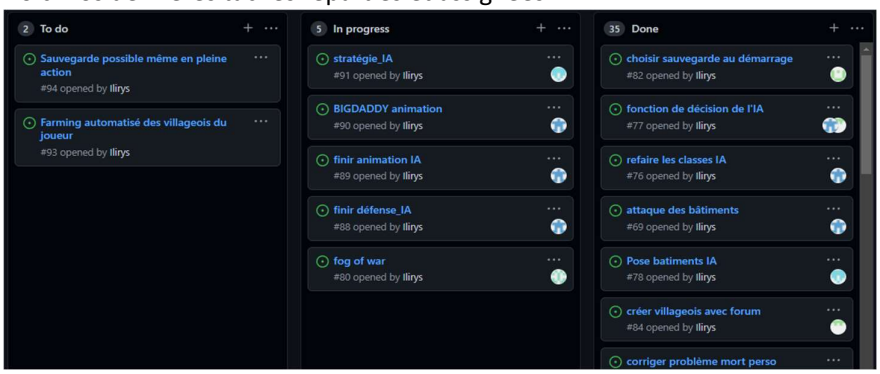


Nous avons décidé unanimement d'utiliser python 3.9 (3.10 plus tard) avec pygame. En effet, il s'agit actuellement de la librairie la plus documentée pour la création de jeux même si arcade se développe. Décidant de rester assez fidèle au jeu de base, les graphismes et les fonctionnalités ont pour but d'y ressembler fortement.

Les créneaux de cours ont été utilisés pour se concerter sur les actions à faire à l'aide d'issues sur GitHub et aussi pour merge nos branches et les tester. Nous codions uniquement sur notre temps libre en dehors de ces temps à l'école. De nombreux échanges sur Discord ont permis de s'aider mutuellement et d'expliquer une partie du code effectuée.

GitKraken a été utilisé pour GIT (voir ci-contre) ainsi que Visual Studio Code et PyCharm pour l'IDE.

Voici nos dernières tâches réparties et assignées.



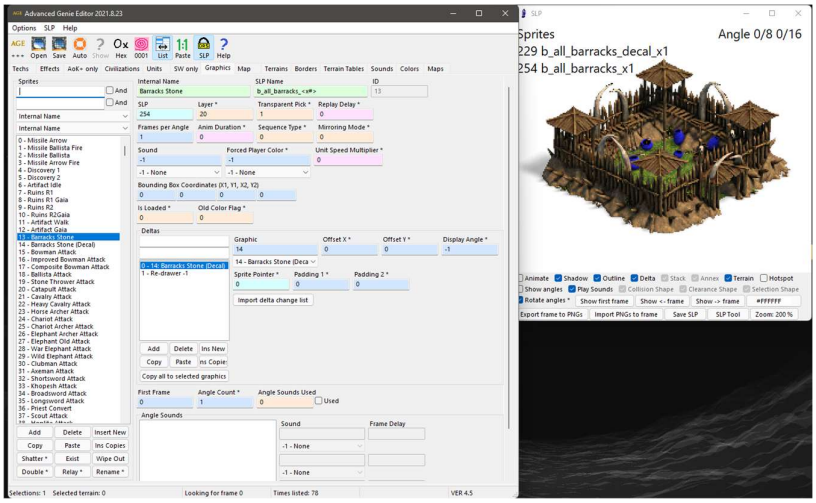
Ci-dessous se trouve les explications imagées du fonctionnement de notre jeu, puis la contribution de chacun avec le consensus final et enfin, la conclusion avec la liste de tous nos objectifs atteints.

I. STRUCTURE

1. Image / sprites

La majeure partie de nos images sont tirées directement du jeu mais ça n'a pas été une mince affaire. Au départ nous avions opté pour l'utilisation de SLP Editor.

Nous n'avions pas réussi à utiliser cet outil à cause d'une incompatibilité des fichiers en .slp issus du jeu car ceux-là étaient alors corrompus.



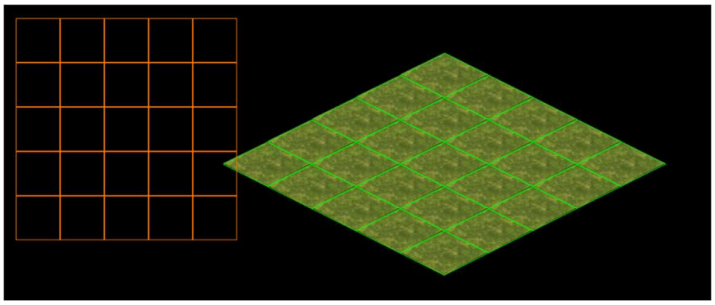
Nous devions dont trouver une autre solution et c'est pour cela que nous nous sommes servis de *AdvancedGenieEditor3* qui nous a permis d'avoir toutes les images du jeu. L'avantage de ce logiciel est qu'il va directement chercher les images dans les fichiers du jeu et donc nous simplifie la vie !

2. Map

La création de la map se fait en deux étapes, tout d'abord nous allons simplement afficher des rectangles à la suite les uns des autres pour faire une grille (en orange sur la capture). Cependant il nous est demandé une vue isométrique de la carte j'utilise donc la formule suivante pour transformer des coordonnées cartésiennes en isométrique.

```
isometricX = cartX - cartY;
isometricY = (cartX + cartY) / 2;
```

On remplit ensuite chaque rectangle isométrique par une image d'herbe. En orange la grille cartésienne, en vert la grille isométrique (ou 2,5D)



Une fois le côté visuel établi, nous élaborons l'aspect logique de la map. Une map se compose d'une d'un tableau d'objet "Tile" (case en français). Cet objet contiendra tous les attributs nécessaires d'une case, c'est-à-dire sa position en coordonnées (on prendra le centre de la case), si elle est vide ou non, le nombre de ressources que possède cette case ainsi que le type de ressource.

Pour se déplacer sur la map, nous avons créé une classe « caméra ». En réalité on ne se déplace pas à travers la map, *c'est la map qui se déplace.* Pour donner cette illusion, on détecte simplement la souris par rapport au bord de l'écran. Si la souris se trouve à une distance inférieure à 2% de la largeur de l'écran, on déplace toute la map à droite, et si elle trouve à plus de 98%, on la déplace à gauche. On procède de même pour la hauteur (mais on déplace la map verticalement). La capture ci-dessous illustre ce concept.

3. Ressources

Dans Age of Empire il existe quatre ressources, le bois, la nourriture, l'or et la pierre. Ces ressources permettent d'effectuer un certain nombre d'actions dans le jeu. Le bois et la pierre permettent de construire des bâtiments, là où la nourriture et l'or sont utilisés pour la création d'unité et de villageois. Ces ressources sont produites par diverses entités. Le bois est récupéré en coupant les arbres, la nourriture en récupérant des baies ou grâce à certains bâtiments, enfin, l'or et la pierre sont récupérés en minant des gisements.




Pour pouvoir gérer ces ressources, nous avions besoin d'un objet contenant tous les attributs nécessaires à l'exploitation de ressources. Un gestionnaire de ressources fut créé. Prenant en paramètre la team, il permet de gérer les ressources de départs, celles disponibles et contient le coût de chaque bâtiment / unité etc.

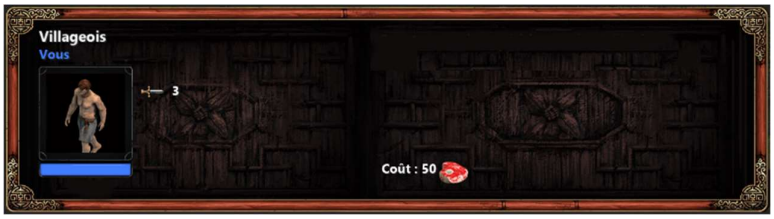
4. Bâtiments

Concernant les bâtiments, une classe avait été créée pour chaque bâtiment : towncenter, house, farm, barrack, storage. Au final, l'initialisation se comportait de la même façon pour tous, donc seule une superclasse a été gardée avec pour initialisation le nombre de pv, les images, la taille (1x1 ou 2x2), la team (joueur ou IA) et la position. Un dictionnaire contenant ces informations pour tous les bâtiments a été mis en place. Ainsi, l'initialisation se fait de la forme.

```
Batiment(render_pos2, "Towncenter", self.resource_manager, dicoBatiment["Towncenter"][2], team="red")
```

Ci-dessus la création du towncenter de l'IA en début de partie avec sa position (render_pos2, les informations contenues dans dicoBatiment et sa team (« red »). Il existe une fonction update au sein des bâtiments pour effectuer les actions spécifiques (par exemple, la ferme ajoute automatiquement 1 de nourriture à intervalle régulier).

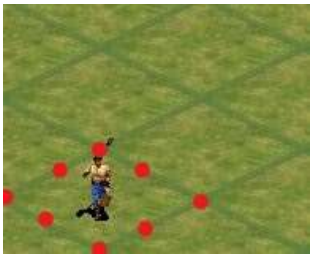
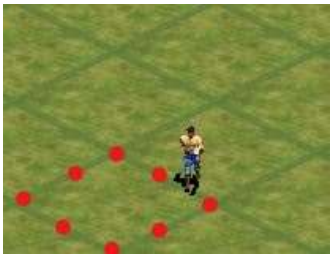
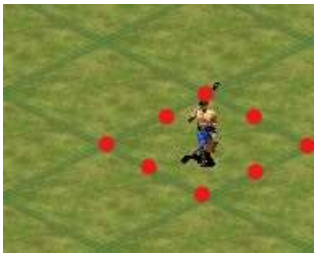
5. HUD



Le HUD est une part importante de n'importe quelle application et c'est encore plus vrai lorsque nous touchons au monde du jeu vidéo. Étant donné que nous ne sommes pas des designers en herbe et pour rester fidèle au jeu, nous avons récupéré les interfaces de ce dernier puis les avons traités séparément. Chaque image récupérée a subi une succession d'opérations. Nous avons dû dans un premier temps ne récupérer que la partie nécessaire que nous avions récupérée de la capture d'écran. Ensuite, nous avons retiré soigneusement les zones de texte ou image ne correspondant pas à nos attentes (nom de civilisation, nombre de points de vie...). Enfin, nous avons homogénéisé le tout pour que nous ne puissions pas voir les découpes que nous avons effectuées. Certaines images ont étés modifiées au pixel près.

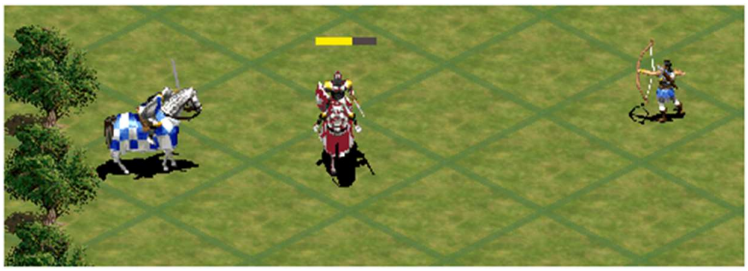
6. Unités

Pour des raisons esthétiques et de simplicité, une case peut contenir une seule unité. Cela permet notamment d'avoir des combats bien plus agréables à voir que si 3 unités pouvaient se battre sur la même case. Une unité n'a donc pas de coordonnées propres mais est simplement affichée aux coordonnées de la case où elle se trouve. Pour donner l'illusion que l'unité se déplace de case en case, sans se téléporter d'une case à l'autre, nous incrémentons sa position sur l'écran jusqu'à la prochaine case. Une fois arrivé sur la case de destination, sa case change.

La case de l'unité est considérée comme "occupée" dans le sens où aucun bâtiment ne peut être posé dessus et aucune unité ne peut se déplacer dessus.

En ce qui concerne les animations des unités, nous avons opté pour le fait d'implémenter une seule animation de déplacement par unité (quelle que soit sa direction), mais huit animations d'attaque (afin de couvrir toutes les directions possibles sachant que chaque unité a huit cases adjacentes) pour que le joueur puisse bien visualiser qui il attaque et qui l'attaque (chose moins importante lorsque le joueur ne fait que déplacer ses unités).



Nous avons également rassemblé toutes les méthodes d'animations en un seul fichier "animations.py", par souci de clarté d'optimisation.

Ces méthodes consistent à créer une liste, d'y ajouter les images nécessaires pour animer une certaine action, puis de retourner la liste. *Par exemple :*

```
def animation_horsemanIA_attack_uright(self):
    horsemanIA_attack_uright = []
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack031V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack032V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack033V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack034V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack035V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack036V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack037V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack038V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack039V2.png').convert_alpha())
    horsemanIA_attack_uright.append(pygame.image.load('assets/horsemanIA/Cavalierattack040V2.png').convert_alpha())
    return horsemanIA_attack_uright
```

Ensuite dans les classes des unités, nous avons une méthode update_sprites() qui est appelée dans la méthode update() des unités et qui, à l'aide de booléens (exemple booléens "attack" et "farms" vus précédemment), choisit l'animation adéquate à l'action effectuée par l'unité. *Par exemple :*

```
def update_sprite(self):
    if self.walkdown_animation == True:
        self.temp += 0.2
        self.image = self.animation[int(self.temp)]
        if self.temp + 0.2 >= len(self.animation):
            self.temp = 0
    elif self.attack_ani == True and self.attack == True:
        self.temp += 0.2
        if self.cible.tile["grid"][0] < self.tile["grid"][0] and self.cible.tile["grid"][1] < self.tile["grid"][1]:
            self.image = self.animation_attack_up[int(self.temp)]
```

Une fois l'animation choisie, la liste englobant les images de cette dernière est parcourue et l'attribut image de l'unité (qui correspond à son image actuelle) prend comme valeur les images de l'animation choisie, une par une.

7. La Classe mère “Worker” :

Lors de notre implémentation des unités, nous avons opté pour le fait de commencer avec une superclasse (que nous avons nommé “Worker” et dont toutes les autres classes d’unités hériteront) qui rassemblerait toutes les caractéristiques et fonctionnalités communes à toutes les classes d’unités, c’est à dire, principalement le déplacement et l’attaque, en plus des différents attributs et méthodes nécessaires à l’accomplissement de ces deux fonctionnalités.

Pour la fonctionnalité de déplacement, nous utilisons ce qu’on appelle en anglais, un “pathfinder”, plus précisément le pathfinder “AStarFinder” auquel nous faisons appel dans notre méthode create_path(x,y) qui prend en paramètres des coordonnées isométriques de “tiles” (cases en français) et qui, initialement, ne faisait que trouver le chemin du “Worker” à partir de sa propre case jusqu‘à la case de coordonnées (x,y) (nous reviendrons sur cette fonction quand nous parlerons de l’attaque ainsi que du “farm”).

Il y’a également une méthode change_tile, qui s’occupe de la gestion de collisions lors du passage d’une unité d’une case à une autre et qui fait également en sorte que des unités entrant en même temps sur une même case ne s’absorbent pas ou ne génèrent pas de bugs, de manière générale.

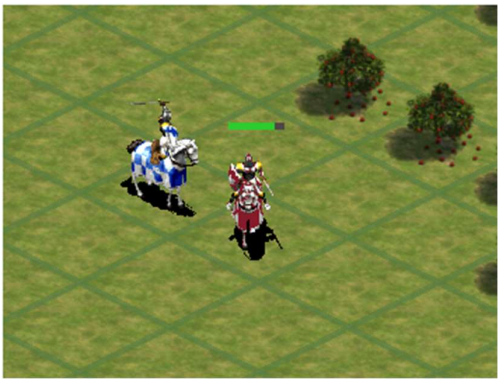
Quant à la fonctionnalité d’attaque, nous l’avons géré comme suit, si la méthode create_path(x,y) est appelée et qu’ il y a une unité appartenant au camp adverse sur la case de coordonnées (x,y) , une variable booléenne “attack” est mise à “vrai”, un chemin vers la case de l’unité ennemie est créé mais notre unité s’arrête une case avant la fin du chemin (donc sur une case adjacente à celle de l’unité que l’on souhaite attaquer) puis dans la méthode d’update (que nous verrons par la suite), nous vérifiions que notre le booléen “attack” est à vrai et s'il l’est et que notre unité est arrivée à sa case de destination (c’est à dire une case adjacente à l’unité ciblée), alors les points de vie de la cible sont décrémentés en fonction des dégâts d’attaque de l’attaquant.

Pour réaliser cela, nous avons commencé par établir un système se basant sur des listes, passées en attributs, et contenant les cases libres adjacentes de chaque unité (en itérant ces cases-là en continu), puis très rapidement, nous nous sommes rendu compte que cette implémentation était relativement couteuse (baisse de fps, …).

Nous avons donc finalement opté pour quelque chose de beaucoup moins couteux et même plus simple à implémenter, en nous intéressant de plus près à notre pathfinder, qui en fait, stock le chemin qu’il crée pour chaque unité dans une liste (passée en attribut).

Cette liste que nous avons nommée “path” contient donc dans l’ordre toutes les cases que l’unité que nous cherchons à déplacer doit parcourir. Donc pour les fonctionnalités nécessitant que notre unité soit à une distance d’une case de sa destination (comme la fonctionnalité de “farm” ou encore celle de l’attaque), il nous suffisait de sortir (avec la méthode pop()) le dernier élément de la liste “path”.

Enfin notre fonctionnalité d'attaque apporte une chose en plus c'est que notre attaquant suit automatiquement sa cible si celle-ci se déplace et continue à l'attaquer. Pour réaliser cela, à chaque fois qu'une unité est à l'arrêt nous stockons sa case actuelle dans un attribut "temp_tile_a" (pour dire case temporaire attaque) et dans la méthode d'update, si les conditions d'attaque, citées précédemment sont vérifiées, nous vérifions de manière continue si la case actuelle de la cible est celle stockée dans "temp_tile_a" et si ce n'est pas le cas nous appelons une nouvelle fois la méthode create_path (qui gère le déplacement, l'attaque et le "farm" comme vu précédemment) en lui passant en paramètres les coordonnées de la case de destination de la cible (qui est en déplacement).



8. La Classe Villager :

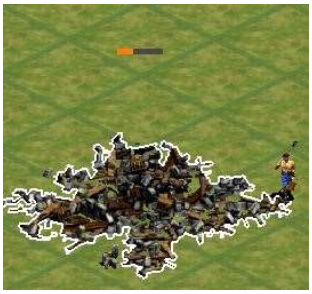
Comme nous l'avons dit dans la partie concernant la classe-mère "Worker", toutes les unités du jeu héritent de la classe "Worker", dont la classe Villager qui est une sorte de "Worker" mais avec les fonctionnalités de "Farm" et de construction de bâtiments en plus.

Pour la fonctionnalité de "Farm" nous avons procédé à la manière de la fonctionnalité d'attaque, c'est-à-dire que nous traitons cela, dans un premier temps, dans la méthode de déplacement create_path(x,y) en vérifiant si la case de coordonnées (x,y) contient, premièrement une collision, puis si elle contient des ressources.

Si c'est le cas, un booléen "Farm" passe à vrai, ensuite dans la méthode update(), nous vérifions si le booléen "Farm" est à vrai et si le villageois a atteint sa case de destination, c'est-à-dire l'avant-dernière case de son chemin vers la case où il doit "farmer" (case adjacente à la ressource), si c'est le cas alors le nombre de ressources portées par le villageois est incrémenté selon son attribut efficacité et le nombre de ressources de la case est, bien sûr, décrémenté de la même manière.

Si le nombre de ressources de la case "farmée" tombe à 0, alors la ressource disparait et le villageois si dirige vers la ressource d'à côté pour continuer sa tâche. Quand le villageois atteint son nombre maximal de ressources transportées, il se dirige vers un grenier, s'il y'en a de construits, sinon vers l'hôtel de ville (Towncenter) afin de déposer ses ressources et retourner "farmer".

Le villageois peut aussi construire un bâtiment, c'est-à-dire poser un bâtiment avec une image de ruine, ayant 0 point de vie, puis se diriger sur la case adjacente pour ensuite incrémenter les points de vie de ce dernier. Il arrêtera de construire s'il est interrompu ou si les PV du bâtiment sont à leur valeur maximale.




II. IA

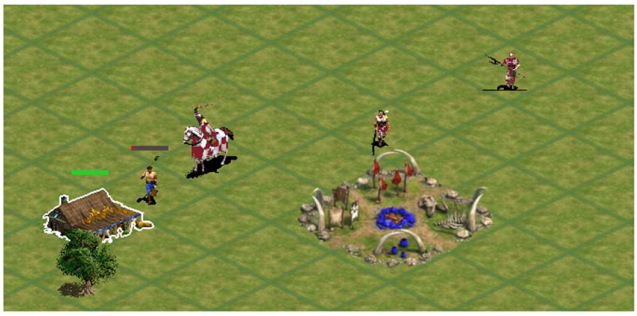
L'IA a été faite dans un 2nd temps, une fois le jeu étant fonctionnel pour un joueur avec les fonctionnalités de base. En effet, une fois l'IA en développement, il devient très compliqué de revenir sur des fichiers et fonctions du joueur sans devoir réadapter toute l'IA derrière. L'IA a plusieurs stratégies décrites plus bas, et agit comme un joueur le ferait, en ripostant quand on l'attaque, en récoltant des ressources quand nécessaire. Étant donné l'aspect temps réel du jeu, notre IA fait des calculs trop lourds pour qu'on se permette (en termes d'fps) qu'elle soit active tout le temps. C'est pourquoi grâce aux évènements pygame, l'IA prendra une décision chaque 2 secondes. Cela permet de ne pas avoir de pertes de performances, ou du moins qu'on ne puisse pas les ressentir.

1. Fonctions

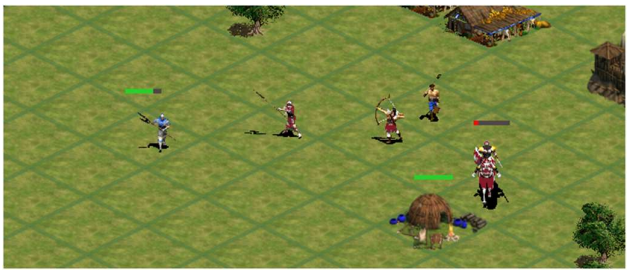
Nous avons réalisé notre intelligence artificielle, ou IA, en 2 parties, tout d'abord, les méthodes qui seront utilisées par cette dernière (par exemple méthode qui attaque des bâtiments du joueur ou encore les villageois, fonctions de farming…etc), ensuite la stratégie qui utilise ces méthodes-là de manière adéquate selon le type de mode de jeu choisi par le joueur en lançant la partie.

La stratégie de notre IA se base sur les méthodes suivantes :

Premièrement nous avons la méthode attack_villagers(), qui comme son nom l'indique fait en sorte que les unités recrutées par l'IA se mettent à attaquer les villageois du joueur l'empêchant ainsi de "farmer" et de se développer.



Puis nous avons la méthode attack_warriors() qui, elle, fait en sorte que les unités combattantes de l'IA attaquent les unités combattantes du joueur, le sabotant ainsi en l'empêchant de réunir ses forces pour attaquer.

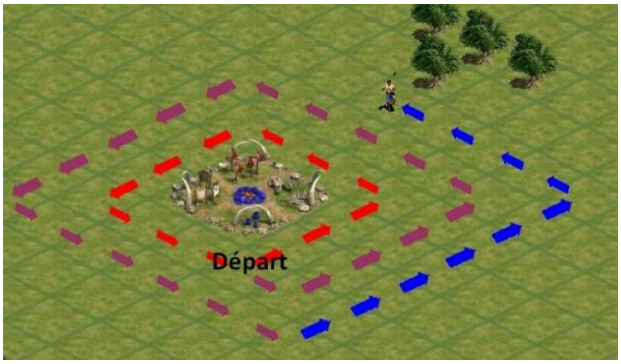


Ensuite nous avons les méthodes attack_batiment(nom_batiment) et attack_player_towncenter() qui, respectivement, font en sorte que les unités combattantes de l'IA attaquent les bâtiments ayant comme attribut "name" la variable nom_batiment passée en paramètres et attaquent l'hôtel de ville (Towncenter) du joueur.

Une autre fonction importante est la récolte de ressources ou « farm » de manière automatique. Plusieurs solutions s'offraient à nous, par exemple vérifier les cases autour du villageois pour savoir si elles contiennent des ressources souhaitées pendant qu'il se déplace dans une direction. Cependant lors de l'implémentation nous nous sommes vite rendu compte que l'aspect temps réel du jeu ne nous permettait pas ce genre de calculs gourmand, notamment avec plusieurs unités.

Nous avons donc trouvé une très bonne astuce : lors du chargement du jeu, nous allions concevoir un algorithme de recherche en spirale partant du Towncenter IA pour vérifier chaque case et dans le cas où elle contiendrait des ressources, cette case sera mise dans une liste (nous avons donc 4 listes pour 4 types de ressources). L'image ci-dessous illustre le début d'un parcours en spirale.

L'algorithme vérifie au tout début la case « Départ » et tourne toujours à gauche sauf si la case à gauche a été visitée, dans quel cas il continuera tout droit. Lorsqu'il rencontre une ressource sur une case, ici un arbre la case est ajoutée dans la liste correspondante. Lors de l'appel à la fonction farm, le villageois ira tout simplement sur la case de la liste de ressource correspondante pour farmer. Il n'y a donc pas de calculs lors du jeu, et les villageois récoltent les ressources les plus proches.



Comme dernière fonction cruciale nous avons celle pour poser les bâtiments. Son but est de trouver un endroit libre pour poser un bâtiment passé en paramètre. Nous avions initialement comme idée de réutiliser l'algorithme en spirale pour trouver un endroit libre autour des bâtiments, mais le résultat n'était pas esthétique, car les bâtiments étaient posés à la suite, collés au towncenter. C'est donc pour ces raisons que la fonction cherche à poser des bâtiments avec une forme de croix avec un peu d'aléatoire (voir capture ci-contre) c'est visuellement plus beau.

2. Stratégies

4 stratégies ont été imaginées : « Attaque » où l'IA se développe pour former une grande armée afin de raser d'un coup le joueur. « Blitz » où l'IA farm rapidement de la nourriture pour lever 5 soldats qui attaqueront le plus rapidement possible le forum du joueur. Enfin, « Vague », qui est un mode alternatif de résistance, où l'IA n'a plus les mêmes contraintes que le joueur. « Défense » est en préparation : l'IA se développe fortement, et garde ses soldats pour protéger le camp puis attaque de temps en temps le joueur. En effet, l'IA va envoyer de façon périodique un nombre de soldats croissants attaquant les soldats, puis les villageois et enfin, le forum.

```
if self.strategy == "defensive":
    match self.evolution:

        case 0:
            if self.ressource_manager.r
                self.spawn_unit_autour_
                self.number_of_building
            else: self.farm(self.food_l
            if self.number_of_buildings
                self.number_of_building
                self.evolution += 1

        case 1:
            if self.ressource_manager.r
                self.find_and_place_bui
            else: self.farm(self.wood_l
            self.compteur_construction_
            if (self.action_faite == 1)
                self.compteur_construct
                self.action_faite = 0
                self.number_of_building
                self.evolution += 1

        case 2:
            if self.ressource_manager.r
                self.find_and_place_bui
            else: self.farm(self.wood_l
            self.compteur_construction_
            if (self.action_faite == 1)
```

La stratégie de l'IA a été pensé pour être construite sous forme de « bloc ». Le système de match case (switch case dans d'autres langages) a été utile et a nécessité la mise à niveau vers python 3.10. Ainsi, lorsque l'IA a fini un bloc de commande, la stratégie va évoluée et donc passer au case suivant (voir image). L'IA va principalement construire selon une liste prédéfinie (caserne → ferme → maison pour la stratégie défensive par exemple) lorsque les ressources le permettront, et les villageois farment en conséquence. Des unités peuvent être déployées et attaquer le joueur lorsque l'IA est assez développée et que la nourriture le permet...

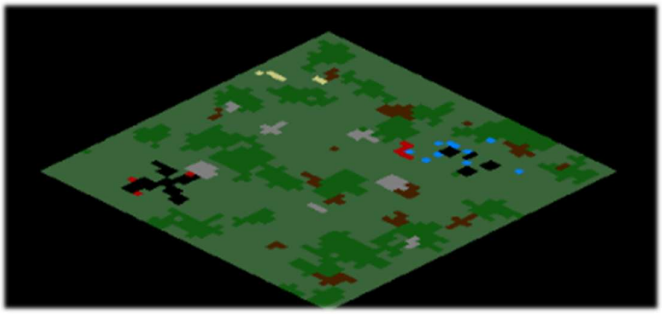
Villageois de l'IA cherchant du bois et construisant des maisons

Villageois farmant des buissons pour créer des soldats avec la nourriture récoltée

Il reste des bugs où l'IA devient soudainement inactive (les villageois ne farment plus), ou bien la population du joueur n'est pas considérée nulle alors que tous les soldats/villageois ont été tués.

3. AUTRES FONCTIONNALITÉS

4. Minimap



Comme dans Age of Empire, une minimap a été ajoutée. Pouvant être activée ou non avec une commande (« minimap ») dans le chat, elle montre toutes les ressources, bâtiments et unités présents sur la carte avec un code couleur établi (vert pour bois, marron pour nourriture, gris pour pierre, jaune pour or, noir pour les bâtiments et bleu / rouge pour les unités du joueur ou de l'IA). Cette minimap est un peu consommatrice de fps au vu du nombre d'éléments affichés. Une optimisation de ne pas afficher toutes les cases vides, mais simplement un arrière-plan global, les cases de couleurs sont ensuite imprimées par-dessus. Si sur 50*50 = 2500 cases, seuls 200 contiennent quelque chose, on affiche 201 éléments et non les 2500.

5. Passage Age

Le concept de passage d'Age est assez trivial, et son implémentation aussi. Lors de l'appui sur l'icône de passage d'Age sur le Towncenter, si les ressources sont suffisantes (nous avons défini le cout de 1500 or) les bâtiments ayant la même équipe que le joueur ayant appuyé sont remplacés par leur bâtiment d'âge supérieur correspondant. Les cavaliers sont débloqués à partir de l'Age de l'outil. De plus, les fermes ajoutent désormais 2 de nourriture (au lieu de 1).

6. Cheats

Nous avons implémenté le fait de pouvoir tricher grâce à une fenêtre de chat interactive. Pour cela il suffit de sélectionner la fenêtre de chat, et de taper l'un des cheatcodes suivants :

NINJALUI : permettant de gagner 10 000 ressources de chaque type

BIGDADDY : invoquant une unité surpuissante

STEROIDS : réduit considérablement le temps de construction, de farming,

STOPSTEROIDS : annule le cheatcode STEROIDS

minimap : affiche ou cache la minimap

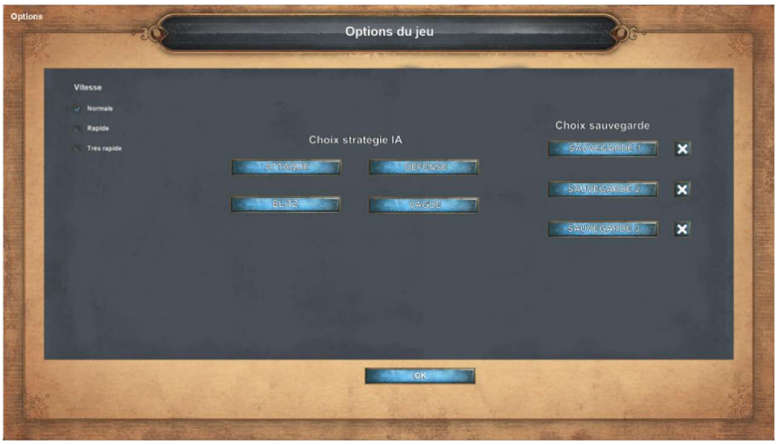
Et certains cheatcodes permettent de modifier la vitesse de jeu :

speed = normal, speed = fast, speed = veryfast, speed = god

7. Options : démarrage, pause

Nous avons implémenté plusieurs menus qui permettent à l'utilisateur d'interagir avec le jeu. Le premier menu et le menu d'accueil qui va apparaitre dès lors que le jeu va être lancé, nous avons alors plusieurs choix, soit on lance le jeu, soit on accède au menu des options soit on quitte le jeu.

Le menu des options nous permet de choisir la vitesse de déplacement des personnages, mais aussi de choisir la stratégie de l'IA. On a aussi le système de sauvegarde qui nous permet de choisir parmi 3 sauvegardes et même de supprimer le contenu d'une sauvegarde.



Le dernier menu est celui de pause qui va pouvoir être activé *in game* en appuyant sur la ECHAP qui nous permet soit de quitter le jeu ou soit de sauvegarder la partie en cours.

8. Sauvegarde

Pour sauvegarder l'état de notre jeu nous avons opté pour le module Pickle, qui sauvegarde sous forme de fichiers binaires. C'est simple d'utilisation mais ne peut sauvegarder les objets pygame. Même en essayant de séparer au mieux la logique de l'affichage, nous avions besoin de certains objets pygame pour la logique du jeu. La sauvegarde qui devait être simple est devenue une tache plus difficile.

C'est grâce à une technique de « Data Transfer Objects » ou DTO que nous avons réussi à sauvegarder l'état du jeu. Le principe est plutôt simple, nous allons créer des objets qui contiendront uniquement les attributs nécessaires à la sauvegarde, pour ensuite les recharger lors du chargement du jeu, et transférer les attributs sauvegardés ensuite.

L'IA est encore plus difficile à sauvegarder car il faut qu'elle retrouve ses actions en cours, et là où elle était avant dans son plan. Pour simplifier, nous allons sauvegarder des entiers « évolution » qui représentent différentes étapes dans sa stratégie, par exemple lors de l'évolution 4 l'IA à déjà construit 3 fermes et 2 maisons, elle sait donc reprendre son fonctionnement normal.

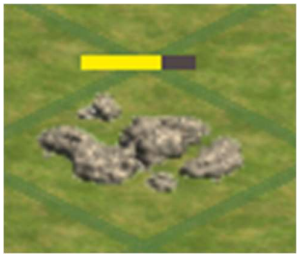
9. Barre de HP / construction

Une barre de points de vie apparaît lorsqu'un élément de la map n'a pas l'entièreté de ses points de vie, passant de la couleur verte, au jaune, puis à l'orange et enfin à la rouge.

Construction / destruction de bâtiments

Indicateur de points de vie des unités

Indicateur du nombre de ressources restantes sur différents éléments de farm



10. Fog of war

Le fog of war est une option qui permet de rajouter un peu de challenge au jeu. Lors d'une partie normale (sans fog of war) nous voyons absolument toute la carte et ce qui s'y trouve. En revanche, lorsque le fog of war est activé, nous ne voyons que le town center (bâtiment de base) et le villageois. Ce dernier doit se déplacer sur la carte afin de découvrir ce qui s'y trouve. Ainsi il ne peut pas récupérer de ressources avant d'avoir été assez proche de celle-ci pour la voir.

Pour des problèmes de compatibilité, le fog n'a pas pu être mergé sur la branche principale. Il pourra toujours être testé dans la branche où il a été créé. Enfin, l'IA ne prend pas en compte pour le moment le fog of war dans sa stratégie.

PARTICIPATION ET PONDERATION

Chacun a écrit les parties du rapport qui le concernaient directement.

Voici le travail apporté par chacun :

A. David:

- Architecture globale du jeu
  - Séparation affichage et update
  - Mise en place de definition.py pour tous les réglages
  - Création de la map isométrique
  - Affichage de la map et ressources (arbres, rochers..)
  - Implémentation de la caméra (Pour se déplacer avec la souris)
  - Implémentation de la matrice de collision
- Hud
  - Implémentation du build hud, resource hud, information hud
  - Affichage de l'hud correspondant en fonction de ce qui est sélectionné
  - Affichage des informations sur les unités (Vie, ressources transportées)
  - Icônes
- Unités
  - Architecture principales des unités (Classe mère dont ses fils dérivent)
  - Classe mère Worker
  - Mise en place du pathfinding
  - Implémentation d'une hitbox, collision
  - Implémentation du déplacement des unités case par case
  - Transition entre les case
  - Affichage des unités en prenant en compte la caméra, affichage selection box
  - Fonction de farm et transfert de ressources
  - Fonction de construction des bâtiments
- Sauvegarde complète
  - Système de Data transfer objects afin de séparer logique et affichage
- Bâtiments
  - Spawn unités à côté du town center
  - Spawn unités autour de la caserne
  - Implémentation du grenier et town center comme dépôt de ressources
  - HP des bâtiments, qui augmentent en construction
  - Les Greniers augmentent la population
- IA
  - Architecture principale de l'IA
  - Système de décision par intervalle de temps
  - Algorithme pour trouver où construire les bâtiments

    - Algorithme de farm pour les villageois IA
- Passage Age
    - Icônes des bâtiments + icône age
    - Passage de l'âge

- *Aide notable:*
    - *Implémentation de bouquet*
    - *Redimensionnement des images du HUD*
    - *Idée pour le fog of war*
    - *Organisation du groupe*
    - *Aide menu sauvegarde*
    - *Images age 2, sprites personnages, Partie Terminée*
    - *Organisation du GitHub (merge)*

B. Lilian:

- Création système écran de démarrage + option
- Bâtiments
    - Classes et tous les différentes constructions : Towncenter, farm, house, storage
    - Pose sur la map si non collision
    - Taille 1x1, 2x2
    - Refonte classe bâtiment avec une superclasse + dictionnaire
- Création Hud
    - Programme global : sections, affichage simple
    - Affichage coordonnées souris
- Génération de camp
    - Pose le forum joueur + adversaire aléatoirement
    - Rase les ressources aléatoirement
- Chat
    - Création cheat NINJALUI, "minimap" (affichage minimap), correction du changement de vitesse avec "speed = XXX"
- Création système de ressources
    - Ressources stockées, affichées dans le hud
    - Bâtiments constructible si assez de ressources
    - Arbres etc enlevés si ne contiennent plus de ressources
- Définition
    - Dictionnaire bâtiments + variables globales
- Minimap complète
- IA
    - Mise en place des systèmes de stratégies à étapes
    - "Intelligence": décisions en fonction des ressources / unités…
    - Les 4 stratégies : "Attaque", "Défense", "Vague" et "Blitz"
- Ecran déclenché de fin de partie
- Equilibrage globale du jeu (ressources, vitesse de farm, pv bâtiments, coût bâtiments / units)

- *Aide notable:*
    - *Organisation du GitHub (merge, issues)*
    - *Organisation du groupe*
    - *Barre de HP*
    - *Passage Age : ajout changement production ferme*

C. Hamza :

- UML
  - réalisation des diagrammes UML
- Unités
  - Architecture principales des unités, création des classes des unités
  - Classe mère Worker
  - Déplacement unités + farm + attaque unités (inutilisés mais étape principale)
  - Premier affichage et déplacement des unités
  - Transition entre les case
  - Fonction de farm (perfectionnée par david)
  - Déplacement unités
  - Fonctions d'attaques des unités
  - Toutes les animations des unités
- Bâtiments
  - Attaque bâtiments
  - Rendre bâtiments attaquables
  - Destruction bâtiments 1x1
  - Destruction bâtiments 2x2
- IA
  - Créations des classes d'unités de l'IA
  - Création de la classe IA
  - Travail sur le principe global de l'IA
  - Architecture principale de l'IA
  - Architecture principale des unités de l'IA
  - Toutes les fonctions d'attaque de l'IA
  - Fonctions de défense de l'IA
  - Toutes les animations unités de l'IA

- *Aide notable:*
  - *Organisation du groupe*
  - *BigDaddy*

D. Nouman:

- UML :
  - Diagramme de classe/ UC
- Classes de base : Ressources / cases / soldat...
- HUD :
  - Images/ mise en place bandeau ressources / âge, HUD bâtiment / unités…
- Fog of war
- Création programme changement couleur pour les Sprites

- *Aide notable:*
  - *Interface du jeu*
  - *Refonte de classe (bâtiment, tile, ressource...)*
  - *Organisation du groupe*

E. Léonard:

- Menu :
  - Menu de démarrage avec options
  - Menu d'options avec changement vitesse / choix entre 3 sauvegardes / 4 stratégies

    - Menu de pause in game avec choix de sauvegarder / quitter
- Chat :
    - Changement vitesse en jeu avec “speed = XXX”
- Images :
    - Ajout de la plupart des images du jeu (herbe, bâtiments, ressources et menus)

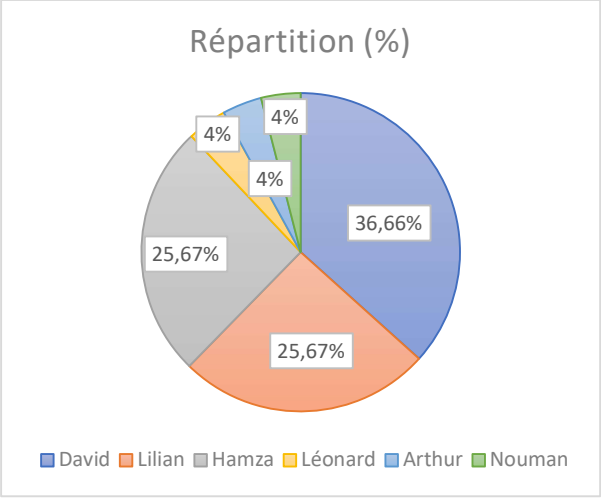
- *Aide notable:*
    - *Sprites des personnages*

F. Arthur:

- Map :
    - Ressources disposées aléatoirement
    - Fonction récursive de création aléatoire de forêts, de carrière de pierre…
- Chat :
    - Création du chat interactif
    - implémentation de cheat codes
- Barre de vie :
    - Création d’une barre de vie dès qu’un élément n’a plus la totalité de ses points de vie
    - Barre pendant la création de bâtiments

En découle cette pondération, obtenue – *très difficilement* – par consensus :

	Consensus (%)
David	36,66
Lilian	25,67
Hamza	25,67
Léonard	4
Arthur	4
Nouman	4



Un manque de communication et une méthode de travail non définie clairement dès le départ ont pu conduire à ce détachement. À noter que le cours d’introduction au génie logiciel a été effectué après coup.

CONCLUSION

Dans cette version de Age of Cheap Empire, de nombreuses possibilités sont offertes et respectent le cahier des charges. Voici un récapitulatif des actions accomplies, définies comme obligatoires et facultatives :

Obligatoire :

- Commencer avec des villageois et un forum
- 4 classes : archer / chevalier (débloqué en age 2) / soldat / villageois + Bigdaddy avec un nombre de HP et attaque différent
- Des ressources limitée, réparties aléatoirement au démarrage : bois (en grande quantité) / buissons avec baies, mine de pierre (non utile pour le moment en age 1 et 2), et or.
- Actions des villageois : cueillir, couper les arbres, miner de l'or / pierre, créer de nouveaux bâtiments
- Bâtiments : Town Center (création villageois, amélioration) / Barracks (entraînement soldats) / Farm (récoltant ponctuellement de la nourriture) / Maison (augmentant la population maximale autorisée)/
- Population maximale définie en fonction du nombre de maisons
- IA jouant comme un joueur avec stratégie : collecte de ressources / construction / technologie / entraînement / assauts

Options disponibles :

- Sauvegarde sans perte d'informations du côté du joueur et de l'IA
- Contrôler la quantité de ressources au départ en modifiant une valeur
- Cheats : NINJALUI, BIGDADDY, STEROIDS, + STOPSTEROIDS, minimap, speed = xxx

Facultatif :

- Génération de map aléatoire
- Vue isométrique (2.5D)
- Garder des graphismes proches de AoE
- RTS = loop main rapide, actions en temps réel
- Brouillard (fog of war) pour joueur (pas pour l'IA pour cause de manque de temps)

Options disponibles :

- Contrôle du temps : accélération possible des déplacements / récolte / construction
- 2 âges avec avantage : chevalier débloqué.

Ainsi, même si des bugs persistent, le jeu reste globalement fonctionnel du début à la fin avec de nombreuses possibilités, des parties complètement nouvelles grâce à l'aléatoire, plusieurs modes d'IA et des graphismes et animations attractifs. La plupart des demandes ont été respectées. Des améliorations pourraient être apportées pour une meilleure expérience de jeux avec plus de temps, tel plusieurs IA, l'IA contre IA et plusieurs civilisations.