

Projet de programmation

-

Le jeu de la vie

MATHIEU Thibaut
POINTEAU Gabrielle
RAVET-LECOURT Florian

POMPIDOR Pierre



Rapport de notre projet de programmation
3ème année de licence d'informatique
Département informatique
Université des Sciences

9 mai 2022

Table des matières

1	Introduction	4
1.1	Notre groupe	4
1.2	Présentation du projet	4
1.3	Interêt dans le monde de l'informatique	4
1.4	Choix de la bibliothèque graphique	5
1.5	Déroulé du projet	7
2	Création de la carte (Map)	8
2.1	Prototype de base	8
2.1.1	Différents terrains	8
2.1.2	Pourcentage des terrains	9
2.1.3	Zones de terrain	9
2.2	Tanière	10
2.3	Dynamique de la carte	11
3	Création des créatures	12
3.1	Besoins biologiques	12
3.1.1	Hydratation	12
3.1.2	Satiété	12
3.1.3	Energie	12
3.2	Critères de survie	13
3.2.1	Perception	13
3.2.2	Mobilité	13
3.2.3	Force	13
4	Fonctionnement du jeu	14
4.1	Paramètres	14
4.1.1	Paramètres Initiaux	14
4.1.2	Sliders	15
4.2	Actions	15
4.2.1	Apport énergétique	16
4.2.2	Apport nutritif et hydrique	16
4.2.3	Reproduction	17
4.3	Fin de partie	17
5	Ajout des prédateurs	18
5.1	Prédateurs	18
5.2	Interaction avec la créature	18

6 Conclusion	20
6.1 Difficultés	20
6.2 Bénéfices	20
6.3 Améliorations possibles	21
Glossaire	22
Acronymes	23
7 Bibliographie	24
7.1 Photos	24
7.2 Définitions	24
7.3 Aides pour le code en JavaScript	24
7.4 Aides en LaTeX	24

Chapitre 1

Introduction

1.1 Notre groupe

Notre groupe est composé de 3 membres :

- MATHIEU Thibaut
- POINTEAU Gabrielle
- RAVET-LECOURT Florian

Notre encadrant est Monsieur POMPIDOR Pierre.

1.2 Présentation du projet

Notre projet s'intitule : Le Jeu de la vie

Portant le même nom que le jeu inventé par John CONWAY, il est en réalité plus proche d'une simulation de vie d'une ou de plusieurs créatures.

Le but de ce projet est donc d'élaborer un jeu de la vie qui génère des créatures pour chaque joueur (max 4 joueurs) sur une carte avec différents terrains. Ces créatures vont alors vivre sur la carte en satisfaisant leurs besoins vitaux. Elles devront donc boire, manger et se reproduire pour pouvoir survivre. La particularité de ce jeu est que les joueurs n'interviennent que sur le paramétrage général des créatures, et n'interagissent pas avec elles, durant la partie.

Il s'agit là de notre première version du jeu, et nous pourrons ensuite y ajouter des prédateurs pour bouleverser la vie de ces créatures herbivores, ainsi qu'éventuellement implémenter un apprentissage des créatures pour se méfier des prédateurs.

Ainsi, on vient d'exposer le projet dans son ensemble. Nous allons par la suite nous intéresser aux intérêts informatiques de ce projet et nous allons aussi le décrire avec plus de détails.

1.3 Interêt dans le monde de l'informatique

Ce projet rentre dans le cadre des jeux Intra-actifs. Il a été créé uniquement dans un but ludique, afin de pouvoir mieux aborder nos notions de Javascript. Ce projet nous a semblé être le plus attractif et le plus intéressant parce qu'il aborde un langage de programmation qu'on maîtrise. De plus, ce projet réunit deux domaines qui nous apprécions : les jeux vidéos et la programmation web.

Comme le jeu se joue seul après le paramétrage des créatures, le joueur a pour but principal de trouver les meilleures données au départ pour que son espèce de créature soit la plus performante. Les paramètres entrés au départ seront affichés au joueur au cours de la partie pour qu'il puisse en déterminer les meilleurs.

1.4 Choix de la bibliothèque graphique

Pour notre projet, nous utilisons le langage de programmation imposé qui est le Javascript. Nous devons donc choisir une bibliothèque à utiliser entre D3.js et Canvas. Ce sont deux paradigmes qui répondent à des besoins différents. Nous avons finalement choisi la bibliothèque D3.js.

C'est-à-dire que si j'ai besoin de souvent modifier mon affichage, il vaudrait mieux utiliser D3.js qui permet d'associer une donnée à un élément SVG, puis de pouvoir la manipuler sans difficulté à l'aide des outils mis à disposition.

On utilisera plutôt Canvas pour les affichages statiques (qui ne bougeront que très peu de fois ou pas du tout), car ce dernier ne peut pas lier des données à des formes : il comprend seulement la gestion des pixels. C'est donc plus difficile de manipuler les éléments de notre affichage.

De plus, l'interactivité avec l'utilisateur, comme par exemple la manipulation avec le clic de la souris est également plus difficile. En effet, comme Canvas n'interagit qu'avec les pixels, il faut déterminer la zone de pixels demandée pour actionner un événement. On doit alors tester pour le pixel sélectionné s'il appartient à cette zone.

Avec D3.js, l'interactivité avec la souris est plutôt facile, c'est-à-dire que la Balise SVG représentant la forme voulue va directement être sélectionnée sans passer par l'analyse des pixels. Comme la Balise SVG est associée à une donnée représentant une forme, lors de la sélection par la souris on va sélectionner directement la balise par l'intermédiaire de l'affichage de la forme voulue, pour actionner un événement.

Pour finir sur les différences entre Canvas et les balises SVG de la bibliothèque D3.js, Canvas demeure plus performant quand il y a énormément de noeuds à manipuler lors de certaines animations. En effet, le rendu visuel sera plus lisse avec Canvas qu'avec D3.js, où son affichage va paraître plus saccadé.

En résumé, si on doit utiliser beaucoup de données en même temps et que celles-ci ont peu d'interactivité avec l'utilisateur, il vaut mieux utiliser Canvas. En revanche, si on a besoin de plus d'interactivité avec l'utilisateur mais de moins de données, il vaut mieux utiliser la bibliothèque D3.js.

Or dans notre projet, nous utilisons un quadrillage qui représente une carte, c'est la raison pour laquelle nous avons choisi d'utiliser la bibliothèque D3.js, car nous devons modifier assez souvent le type de case (leurs couleurs) ce qui est facilité par cette bibliothèque. De plus, cela nous permet d'utiliser des événements plus facilement sur ces cases lorsque les créatures de notre jeu sont en action.

De plus la balise SVG est très utile car elle permet d'écrire directement dans le DOM c'est-à-dire qu'on écrit à l'aide de d3.js la base du code pour définir de nouveaux objets qui vont être créés directement dans le DOM.

On peut voir sur la figure 1.1 à la page 6 que l'on a créé la carte et la balise SVG.

Puis sur la figure 1.2 à la page 6, on a créé les différentes cases de la grille pour former notre carte en leur donnant des attributs qui correspondent à leurs caractéristiques graphiques dans la balise SVG.

Ainsi ces deux créations ont été écrites dans le DOM comme montré sur la figure 1.3 à la page 6.

```
// Création de la balise svg
let svg = d3.select("#carte")           //selectionne l'id de la carte
    .append("svg")                     //ajoute la balise svg
    .attr("width", nbColonnes*taille)   //taille en largeur du contour ici de la carte
    .attr("height", nbLignes*taille);  //taille en hauteur du contour ici de la carte
```

FIGURE 1.1 – Création de la carte et de la balise svg pour créer des formes

```
d3.select("svg")           //selectionne la balise svg
    .append("rect")        //ajoute un rectangle
    .attr("width", taille) //on donne la largeur de la case donnée en paramètre
    .attr("height", taille) //de même pour la longueur
    .attr("x", x)          //coordonnée x
    .attr("y", y)          //coordonnée y
    .attr("stroke", "black") //contour de la case est noir
    .attr("fill", function(){
```

FIGURE 1.2 – Création des rectangles qui forment la carte dans la balise svg

```
▼ <div id="carte">
  ▼ <svg width="300" height="300"> [débordement]
    <rect id="0_0" width="10" height="10" x="0" y="0" stroke="black" fill="green" pousse="6"></rect> [event]
    <rect id="0_1" width="10" height="10" x="0" y="10" stroke="black" fill="dimgrey"></rect> [event]
    <rect id="0_2" width="10" height="10" x="0" y="20" stroke="black" fill="darkkhaki"></rect> [event]
    <rect id="0_3" width="10" height="10" x="0" y="30" stroke="black" fill="green" pousse="6"></rect> [event]
    <rect id="0_4" width="10" height="10" x="0" y="40" stroke="black" fill="green" pousse="6"></rect> [event]
    <rect id="0_5" width="10" height="10" x="0" y="50" stroke="black" fill="green" pousse="6"></rect> [event]
```

FIGURE 1.3 – Création de la carte de la balise svg et des rectangle qui forme la carte dans le DOM

1.5 D roul  du projet

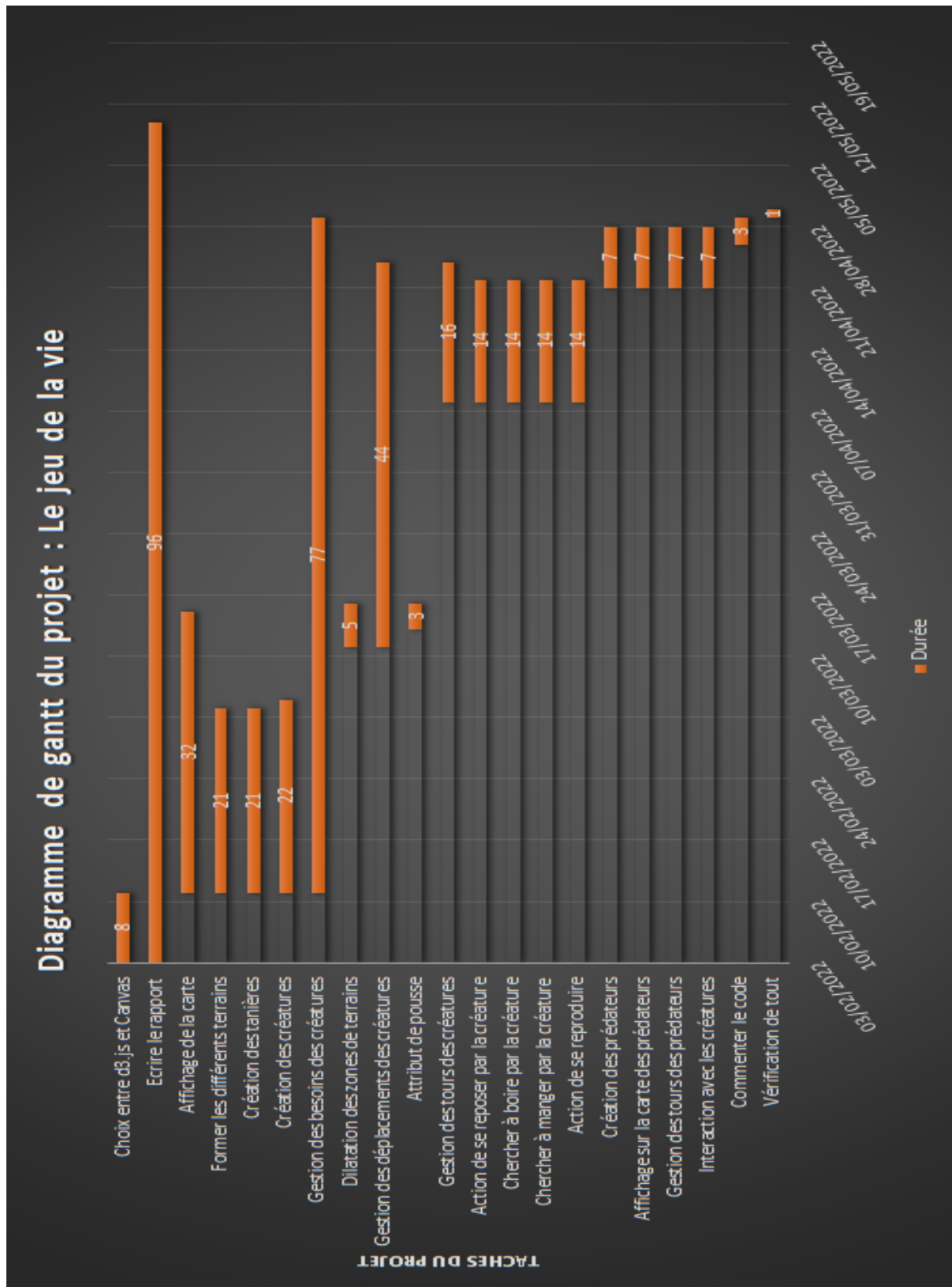


FIGURE 1.4 – Diagramme de gantt de notre projet

Chapitre 2

Création de la carte (Map)

2.1 Prototype de base

La génération de la carte se fait automatiquement après la sélection du nombre de joueurs dans un formulaire HTML. Après validation, à l'aide d'un bouton de type "submit", une fonction JavaScript se déclenche, elle fait disparaître le formulaire avant de mettre en place la carte.

2.1.1 Différents terrains

Nous avons plusieurs terrains possibles que nous allons disposer sur notre carte. Chaque case est associée à un type de terrain pour la totalité du jeu réalisé :

- Eau
- Terre/Herbe
- Sable
- Roche

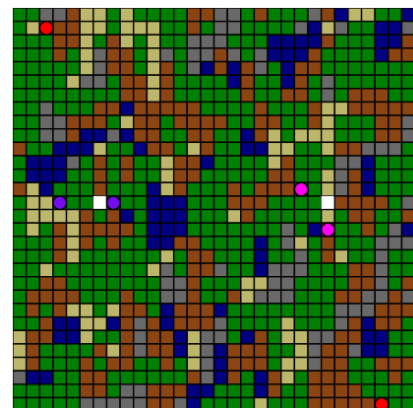


FIGURE 2.1 – Carte des différents terrains

Chaque type de terrain a une couleur attribuée pour bien les différencier. Ils occupent une fonction qui peut être vitale ou un obstacle pour les créatures :

- Le sable ne possède aucune particularité : les créatures n'y performent aucune action. Le sable est représenté par la couleur marron très clair que l'on peut voir sur l'image 2.1 à gauche.
- L'eau permet aux créatures de s'hydrater, pour cela elles doivent se trouver sur une case adjacente, il leur est impossible de se tenir sur une case eau. Les cases eaux correspondent aux cases bleu foncé de l'image à droite.

- La terre et l’herbe ne forment qu’un seul type complémentaire, une créature se nourrit sur une case herbée qui laisse place à une case terre une fois l’herbe consommée. Avec le temps, la case terre laissera revenir de la verdure. Sur l’image on voit en marron les cases de terres et en vert foncé les cases d’herbe, il existe également une couleur verte claire qui correspond à l’herbe qui vient de repousser ou l’herbe qui a été mangée par une créature.
- Enfin la roche constitue un obstacle pour nos créatures, les cases roches demandent plus d’efforts de la part de la créature pour être franchies. Elles sont représentées par la couleur grise sur la carte.

Pour colorer la carte de cette façon on utilise un argument SVG "fill" qui va représenter le type de case.

2.1.2 Pourcentage des terrains

Chaque case du terrain est générée aléatoirement, cependant, pour conserver une certaine cohérence et un environnement jouable, des restrictions doivent être mises en place.

La génération d’une case dépend d’un nombre aléatoire qui détermine son type, ainsi, en jonglant avec le pourcentage de chance pour qu’un type de case soit choisie, on peut manipuler leurs quantités respectives.

```
.attr("fill", function(){
    //couleur en fonction des valeurs aléatoire
    //60% d'herbe/terre
    if (rand < 60) {
        if (rand2 < 60) return "green";           //si herbe
        else return "saddlebrown";              //si terre
    }
    //15% d'eau
    if (rand < 75) return "navy";                 //si mer
    //12% de sable
    if (rand < 87) return "darkkhaki";            //si sable
    //12%de sable
    else return "dimgrey";})                      //si rocher
```

FIGURE 2.2 – Code pour donner une couleur à chaque case

2.1.3 Zones de terrain

Après avoir réparti de façon aléatoire les terrains sur les cases de la carte, nous nous sommes retrouvés avec un amas de couleur sans cohérence. Pour remédier à cela, nous avons mis en place une fonction de lissage. Celle-ci examine une à une les cases de notre carte, ainsi que ses voisines. Après analyse et étude des types voisins de notre case, ainsi que de sa nature première, la fonction décide ou non de modifier le terrain actuel.

Ainsi, après application notre carte comporte de vraies zones d’eau, de roche ou d’herbe pouvant s’apparenter à des lacs, des montagnes ou des rivières.

D’après le code sur la figure 2.3 à la page 10, on a fait le choix de modifier tous les voisins de la case courante, si une seule des cases voisines est de la même couleur que la case

courante. Ainsi on a utiliser un tableau qui nous permet à chaque case de lui re-attribuer une couleur si besoin.

```

////////////////////
// Parcours des voisins //
////////////////////

for (let u = -1; u<=1; u++){ //on parcourt les voisins en largeur
  for (let v = -1; v<=1; v++){ //on parcourt les voisins en hauteur

    //CELLULES VOISINES
    cellVoisin = $('#'+(i+u)+'_'+(j+v)); //on stocke une cellule voisine
    let couleurVoisin = cellVoisin.attr("fill"); //on stocke la couleur d'une cellule voisine
    //console.dir(couleurVoisin); //on veut éventuellement afficher la couleur du voisin

    //LIMITE DE LA GRILLE
    if ((i + u) < 0 || (i + u) >= nbColonnes || (j + v) < 0 || (j + v) >= nbLignes){
      continue;
    }

    //COMPARAISON DES COULEURS DES CELLULE COURANTE ET VOISINES
    if (couleurVoisin == couleurCourante){
      //si deux case de même couleur sont a coté
      couleur = couleur || true; //on dit que oui il y a deux case de la meme couleur a coté
    }

    //si il y a deux case cote à cote de même couleur
    if (couleur){
      //On ajoute les cellules voisines avec la couleur courante dans le tableau des modifications
      modif[cellVoisin.attr("id")] = couleurCourante;
    }
  }
} //fin du parcours des voisins

//SI UN VOISIN PAREIL QUE LA CELLULE COURANTE
if (couleur){
  modif[cellVoisin.attr("id")] = couleurCourante; //on ajoute au tableau des modifications les cellules voisines de la cellule courante pour les metre à la meme couleur
}

```

FIGURE 2.3 – Code du parcours des voisins

Une fois qu'on a attribué à toutes les cases une nouvelle couleur on peut donc modifier les cases à l'aide de notre tableau sont en modifiant le "fill" des différentes cases, comme on peut le voir sur la figure 2.4 à la page 10.

```

////////////////////
// Modification de la grille //
////////////////////

for(let c in modif){ //on parcourt le tableau modif
  //console.dir("indice : "+c); //on peut afficher l'indice de la cellule a modifier

  let cellCouranteModifie = $('#'+c); //on stocke la cellule courante à modifier
  //console.dir("couleur avant : "+cellCouranteModifie.attr("fill")); //on peut afficher la couleur avant modification dans la console

  //on verifie que la cellule à modifier n'est pas une tanière
  if (cellCouranteModifie.Taniere == undefined){
    //on donne la couleur associé à la case
    cellCouranteModifie.attr("fill", modif[c]);
  }
  //console.dir("couleur apres : "+cellCouranteModifie.attr("fill")); //on peut afficher la couleur après modification
} //fin parcours de modif

```

FIGURE 2.4 – Code du parcours des voisins

2.2 Tanière

On cherche maintenant à placer les tanières des 4 joueurs. Les tanières sont des cases spéciales, uniques pour chaque joueur. C'est là que les créatures pourront se reproduire après avoir répondu à leurs besoins primaires et c'est le seul lieu où elles seront intouchables.

On les répartit donc de façon à ce que les tanières soient à peu près à égale distance l'une de l'autre et du centre de la carte. S'il y a un seul joueur la tanière est au centre de la carte. S'il

Il y a deux joueurs des tanières sont placés en ligne, espacés d'environ un tiers de la carte. S'il y a 3 joueurs, les tanières sont réparties en triangle à peu près équilatéral. Et s'il y a 4 joueurs, les tanières sont réparties en losange par rapport au centre.

Sur la figure 2.1 à la page 8 elles sont représentées en blanc selon leur disposition par rapport au nombre de joueurs.

2.3 Dynamique de la carte

L'herbe est le seul type de case dynamique de notre carte. Contrairement aux autres cases elle possède un attribut supplémentaire de pousse. Cet attribut peut prendre des valeurs allant de zéro à six qui décrivent trois stades de repousse différents.

De zéro à deux, il n'y a pas d'herbe, la case est considérée comme une case de terre comme sur la figure 2.5 à la page 11. De trois à cinq l'herbe commence à repousser et se distingue par sa couleur vert clair, les créatures peuvent se nourrir dessus comme sur la figure 2.6 à la page 11. Enfin pour un attribut de pousse de valeur six, la case est une vraie case herbe, les créatures peuvent également se nourrir dessus comme sur la figure 2.7 à la page 11.

Chaque fois qu'une créature se nourrit, elle fait passer la case à son stade inférieur. De plus, à chaque tour, et pour toutes les cases concernées, l'attribut de pousse est incrémenté de un.

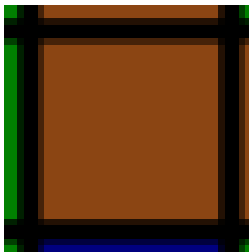


FIGURE 2.5 – Case de terre

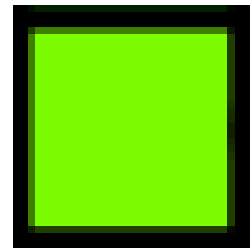


FIGURE 2.6 – Case d'herbe coupé

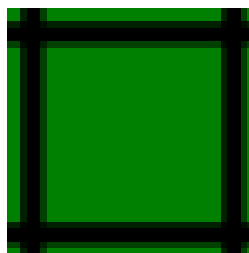


FIGURE 2.7 – Case d'herbe repoussé

Chapitre 3

Création des créatures

3.1 Besoins biologiques

La représentation biologique des créatures se fait au travers de trois jauges différentes.

- Valeur d'hydratation
- Valeur de satiété
- Valeur d'énergie

Toutes les trois sont initialisées à 50 à la naissance d'une créature. Ces jauges vont varier tout au long du jeu entre 0 et 100 pour simuler les besoins que les créatures vont devoir satisfaire pour survivre. De ce fait, plus la valeur d'une jauge diminue plus la créature est en danger de mort.

3.1.1 Hydratation

L'hydratation représente le besoin en eau de la créature. La créature ressent le besoin de boire à partir d'une valeur inférieure à 75, elle se mettra alors en quête d'une case eau pour répondre à ce besoin.

La valeur d'hydratation diminue après chaque tour de jeu d'une créature. Si cette valeur atteint 0 la créature mourra de soif.

3.1.2 Satiété

La satiété représente le besoin en nourriture de la créature. La créature ressent le besoin de manger si elle n'a pas soif et si la valeur de satiété est inférieure à 75, elle se mettra alors en quête d'une case herbe pour répondre à ce besoin.

La valeur de satiété diminue après chaque tour de jeu d'une créature. Si cette valeur atteint 0 la créature mourra de faim.

3.1.3 Energie

L'énergie représente la capacité physique actuelle de la créature. À chaque déplacement la créature va consommer de l'énergie, a contrario elle pourra se reposer pour en gagner. Plus les déplacements sont longs plus l'énergie à dépenser sera importante. Une créature prend la

décision de se reposer si ses deux autres besoins sont satisfaits et si la valeur d'énergie est inférieure à 75.

Lorsque la valeur d'énergie atteint 0 la créature rentre dans un état amorphe, elle devra obligatoirement se reposer au prochain tour.

3.2 Critères de survie

Pour répondre à leurs besoins biologiques chaque créature possède des critères de survie lui permettant d'évoluer dans son environnement. Ces critères doivent être paramétrés de manière différente pour chaque joueur en début de chaque partie. En effet, chaque joueur se verra attribuer un nombre de points qu'il devra répartir entre les différents critères, leurs valeurs variants de 1 à 5. Le but sera donc de trouver le meilleur paramétrage pour que nos créatures puissent survivre le plus longtemps.

Les créatures possèdent quatre critères de survie :

- Perception
- Mobilité
- Force
- Taux de reproductivité

3.2.1 Perception

La perception représente la capacité de la créature à analyser son environnement. Lors de son tour de jeu la créature pour répondre à son besoin le plus urgent va scruter les environs. Ainsi pour chaque point de perception qui lui a été attribué elle va pouvoir voir à une distance d'une case supplémentaire. C'est alors que la créature pourra déterminer la cible de son prochain déplacement.

3.2.2 Mobilité

La mobilité représente la capacité de la créature à se mouvoir dans son environnement. Lors de son tour de jeu après avoir décidé où se rendre la créature va pouvoir se déplacer. Ainsi pour chaque point de mobilité qui lui a été attribué elle va pouvoir se déplacer à une distance d'une case supplémentaire vers son objectif.

3.2.3 Force

La force représente la capacité de la créature à se défendre face aux prédateurs. Lors de son existence il se peut qu'une créature se fasse attaquer par l'un des deux prédateurs présents sur la carte. Ainsi chaque point de force qui lui a été attribué augmente ses chances de survie face à cet agresseur.

Chapitre 4

Fonctionnement du jeu

4.1 Paramètres

Au début de la partie, un joueur doit choisir le nombre de tours total de la partie et aussi calibrer les paramètres précédemment décrits. Une fois ceux-ci configurés, il ne sera plus possible de les modifier au cours de la partie. Ces configurations vont servir à la fin de la partie pour indiquer celle qui a été la meilleure. Commençons donc par les énoncer :

4.1.1 Paramètres Initiaux

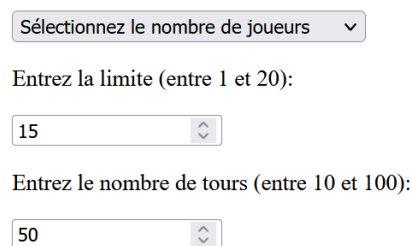
Dans un premier temps un joueur renseignera le nombre total de participants à la partie (entre 1 et 4 joueurs).

Une fois ce nombre choisi, il doit ensuite choisir un total de points maximum attribuable à une créature dans cette partie. Le total des points sur les différents paramètres ne devra pas dépasser ce maximum.

Par exemple, si le joueur principal choisit 12 comme maximum, chaque combinaison doit avoir un total en-dessous ou égal à 12 (ex : 2, 2, 4 et 4 ou 3, 2, 3 et 4 sont des combinaisons valides).

Si jamais la somme des sliders, que nous allons expliquer plus bas, sont plus grandes que le maximum, un message d'avertissement s'affichera pour indiquer qu'une redistribution des points est nécessaire.

De plus, il devra renseigner le nombre de tours que les créatures pourront faire, une fois ce nombre dépasser alors le jeu s'arrête.



Sélectionnez le nombre de joueurs ▼

Entrez la limite (entre 1 et 20):

15

Entrez le nombre de tours (entre 10 et 100):

50

FIGURE 4.1 – Affichage au démarrage du jeu

4.1.2 Sliders

Une fois que le joueur principal a rempli les paramètres généraux, des sliders pour chaque joueur s'affichent sur l'écran comme sur la figure 4.2 à la page 15. Ainsi, chaque joueur doit sélectionner sur ses sliders les taux des différents critères de survie de ses créatures.

On doit alors remplir le taux de reproductivité, de perception, de mobilité et de force. Ces différents critères de survie ont été décrits un peu plus haut dans la partie « Critère de survie ».

Une contrainte permet que deux joueurs n'aient pas la même configuration de jeu (les mêmes taux sur chaque critère). Un avertissement du système sera également affiché si ce point n'est pas respecté.

Après avoir rempli tous les paramètres on peut donc passer à la partie jeu et l'explication des actions possibles de la créature à chaque tour, comme sur un jeu de plateau, mais qui sont appliquées automatiquement par le programme.



FIGURE 4.2 – Exemple de sliders d'un joueur

4.2 Actions

Nous venons d'exposer la base du jeu avec son interface et les critères des créatures, nous allons maintenant expliquer les différentes actions possibles dans un tour de jeu. Les créatures ont en début de tour quatre possibilités de jeu classées ici par ordre de priorité :

- Se reposer si elle est dans un état amorphe
- Chercher à boire
- Chercher à manger
- Se reproduire

Plus précisément, lorsque c'est son tour, la créature analysera ses besoins pour savoir duquel elle dépend en priorité, puis observera le terrain pour trouver une case correspondante et enfin se déplacera dans sa direction.

Lors d'une observation ou d'un déplacement ce sont les critères de survie qui influent sur leur efficacité. Pour n points attribués la créature pourra observer/se déplacer dans un rayon de n cases, soit jusqu'à n cases en ligne droite et $n/2$ cases en diagonale. Sur la figure 4.3 à la page 16 on peut ainsi voir pour différents points de mobilité les possibilités de déplacement de la créature.

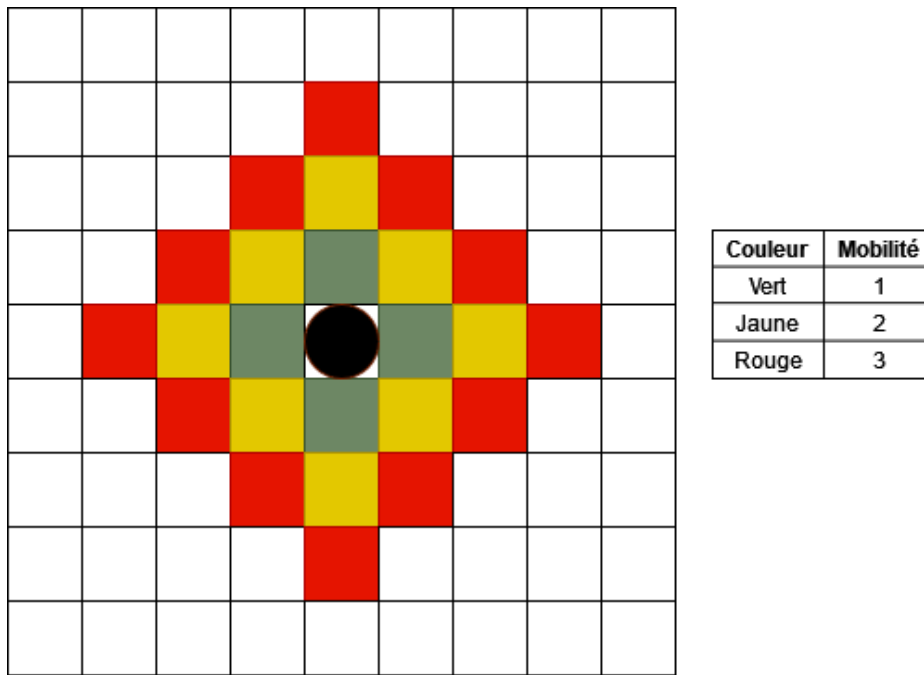


FIGURE 4.3 – Possibilité de déplacement en fonction de la mobilité

La recherche d'un chemin entre la case cible et la case actuelle de la créature s'effectue à l'aide d'un algorithme `plusCourtChemin`. Chaque case est composée de trois attributs : sa coordonnée en x, sa coordonnée en y ainsi que le parent de la case (pour la case actuelle, son parent est initialisé à null). Il existe un chemin entre deux cases c1 et c2 si dans les parents de la case c1 se trouve la case c2.

Tant qu'il reste des cases à explorer ou tant que la cible n'a pas été trouvée, on cherche les enfants de la première case dans le tableau des mouvements disponibles et, si les cases trouvées sont valides et non explorées, on les ajoute dans le tableau de mouvements possibles.

Si l'algorithme trouve un chemin entre ces deux cases alors il renvoie un tableau contenant ce chemin, sinon il renvoie le chemin entre la case actuelle et la case la plus proche de la case cible. Si la case cible n'existe pas ou qu'il s'agit d'une case eau alors il renvoie null.

4.2.1 Apport énergétique

Pour remonter sa jauge d'énergie, la créature peut prendre la décision de se reposer, ainsi cette valeur augmente de 25 et son tour prend fin. Pour chaque déplacement la créature perd de l'énergie à hauteur de 3 énergies par case.

4.2.2 Apport nutritif et hydrique

Si la créature a besoin de manger ou de boire elle cherchera donc une case d'eau ou d'herbe. Pour cela, il y a plusieurs cas de figure :

- Si dans son champ de vision se trouve une case d'eau ou d'herbe, alors elle va s'en approcher grâce à sa mobilité. Si cette dernière est plus grande ou égale à la distance entre les deux cases alors la créature va s'y déplacer.

- Sinon, elle va simplement s'en approcher le plus près possible.
- S'il n'y a rien dans son champ de vision elle va se déplacer sur une case au hasard parmi les cases où elle peut se déplacer.

Après s'être déplacée, la créature mange ou boit. Cela augmente le taux d'hydratation ou de satiété de la créature de 25.

4.2.3 Reproduction

Si la créature n'a aucun besoin, donc que ces derniers sont assouvis, elle retourne à sa maison.

Une fois arrivé, s'il y a déjà une autre créature de la même espèce (du même joueur) et de genre différent dans la tanière, alors il peut y avoir reproduction. Les créatures concernées perdent alors 45 d'énergie. Si la reproduction est un succès cela augmentera de 1 la population de créatures de la même espèce et augmentera les chances de gagner du joueur.

4.3 Fin de partie

La partie se termine après la fin du nombre de tours maximum, indiqué en début de partie. Le joueur possédant le plus grand nombre de créatures en vie sera désigné comme le gagnant de la partie. Ainsi, ce n'est pas forcément le joueur avec le plus de créatures totales qui a le plus de chances de s'approcher de la victoire.

Chapitre 5

Ajout des prédateurs

Nous avons maintenant une version du jeu fonctionnelle, les créatures peuvent survivre, se reproduire et s'épanouir de manière autonome. Pour ajouter une dimension de réalisme supplémentaire nous allons ajouter des prédateurs dans notre environnement.

5.1 Prédateurs

Les prédateurs sont des créatures qui ne sont pas générées par les joueurs. Au nombre de deux ils apparaîtront dans les coins haut-gauche et bas-droit de la carte, ils ne sont soumis à aucune jauge biologique et ne peuvent en conséquence pas mourir.

Aussi ils ont la possibilité de se déplacer sur les cases eau. D'ailleurs, leurs déplacements sont un peu particuliers. En effet, à chaque tour, ils peuvent se déplacer d'une case vers le haut ou le bas et d'une case vers la gauche ou la droite.

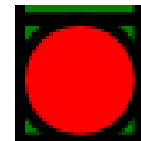


FIGURE 5.1 – Illustration d'un prédateur sur la carte

Sur la carte, les prédateurs sont de couleur rouge comme sur la figure 5.1 à la page 18.

5.2 Interaction avec la créature

```
/**
 * Fonction qui fait attaquer le prédateur sur la créature
 * @param cible cible du prédateur qui représente une créature
 */
attaque(cible){
  console.log(this.force);      //affichage sur la console de la force du prédateur
  console.log(cible.force);     //affichage sur la console de la force de la créature

  //Si la force du prédateur + un nombre aléatoire est plus grand que la force de la créature + un nombre aléatoire
  if(this.force + getRandomInt(6) > cible.force + getRandomInt(6)){
    cible.mort();               //la créature meurt
    console.log("miam");        //affichage dans la console que la créature est mangée
  }
  else{console.log("snif");}    //si ce n'est pas le cas, affichage de l'échec de l'attaque du prédateur
}
```

FIGURE 5.2 – Code de la fonction attaque des prédateurs

Le tour des prédateurs s'effectue après celui de chaque créature de la partie. Si un prédateur finit son tour sur la même case qu'une créature alors il l'attaque. Le combat s'engage et chaque opposant se voit attribuer une valeur de combat pour celui-ci, cette valeur est déterminée par l'addition de la force et d'un nombre aléatoire variant de 0 à 6. Si le résultat du prédateur est le plus important alors il mange la créature qui meurt, dans le cas contraire la créature survit et n'est plus inquiétée pour ce tour.

On peut donc voir sur la figure 5.2 à la page 18 le code de la fonction attaque qui compare les forces de la créature et du prédateur pour savoir si la créature reste en vie ou se fait dévorer par le prédateur. On met en place une chance de survie de la créature à l'aide d'un nombre aléatoire pour que la créature puisse, si jamais elle est moins forte que le prédateur, réussir à s'en sortir quand même.

Chapitre 6

Conclusion

6.1 Difficultés

- Implémentation de l'algorithme PlusCourtChemin
Une des difficultés importantes a été l'implémentation d'un algorithme répertoriant les mouvements possibles de la créature. D'abord effectués de manière heuristique, nous avons dû nous résoudre à implémenter l'algorithme PlusCourtChemin, ce qui a eu pour conséquence de retravailler la fonction `tour()` de nos créatures.
- Gestion du tour de la créature
Une autre difficulté a été la mise en place du tour de la créature avec la hiérarchisation de ses besoins, le moyen de trouver la case la plus proche du besoin actuel et la perte d'énergie lors du déplacement.
- Utilisation de `setInterval`
L'implémentation et l'utilisation de `setInterval` nous ont demandé un temps d'adaptation avant de trouver les bons paramètres pour un rendu optimal.

6.2 Bénéfices

- Acquis
La mise en place des compétences théoriques que nous avons acquises tout au long de notre cursus en JavaScript a été une opportunité pour nous d'appliquer et d'améliorer nos connaissances. D'autant plus que nous avons pu découvrir la bibliothèque graphique `d3.js` essentielle pour notre projet.
- Gestions/Organisation
Au cours du projet, nous avons appris à rebondir sur le travail accompli. L'utilisation de la méthode agile nous a permis de nous adapter aux divers imprévus. Notre gestion du temps est retranscrite dans le diagramme de gantt sur la figure 1.4 à la page 7.
- Mise en commun du travail
Ce projet nous a aussi incité à mettre notre travail en commun. Chaque étape devait être compréhensible par tous les membres du groupe, ainsi chaque opération était accessible en temps réel par l'utilisation d'outils collaboratifs tels que `git` que nous avons appris à maîtriser.

6.3 Améliorations possibles

- Apprentissage
Une idée d'implémentation serait l'ajout d'une intelligence artificielle qui permettrait aux créatures d'éviter les prédateurs (exemple : effet de groupe, transmission des connaissances).
- Connexion en réseau
Une autre idée serait d'utiliser les sockets à l'aide de Node.js pour créer un jeu disponible en réseau.
- Modèle de créature moins simpliste
Une remarque que l'on pourrait faire sur nos créatures serait qu'on ne peut différencier les créatures des prédateurs que par leurs couleurs. Ainsi, une autre idée serait de changer le modèle de la créature pour pouvoir bien les différencier.
- Amélioration du code
Le code devrait être retravaillé pour le rendre plus lisible et plus optimal, des calculs sont parfois effectués alors qu'ils ne sont pas nécessaires.

Glossaire

API Les Application Programming Interfaces (API) sont des interfaces disponibles dans les langages de programmation qui permettent aux développeurs de créer plus facilement des fonctionnalités.. 22

Balise SVG Le Scalable Vector Graphics (SVG) pour les programmes écrits en HTML, est un format d'image XML qui permet de créer des formes simples et complexes comme des rectangles, triangles etc... à insérer dans les documents HTML sans passer par un Logiciel de PAO. En JavaScript, la bibliothèque D3.js permet d'utiliser les balises SVG de la même façon pour pouvoir ensuite directement les utiliser en les sélectionnant par les fonctions données par la bibliothèque D3. De plus, les balises SVG permettent d'écrire directement dans le DOM. . 5

DOM Le Document Object Model (DOM) est l'interface de votre page Web. Il s'agit d'une API qui permet aux programmes de lire et de manipuler le contenu, la structure et les styles des pages.. 22

Intra-actifs Jeu dont la base se joue sans interaction avec les joueurs pendant la partie. Les joueurs remplissent les paramètres du début et laisse l'ordinateur prendre l'initiative des actions des créatures. Celles-ci sont réalisées en fonction des besoins des créatures.. 4

Logiciel de PAO Un logiciel de Publication assistée par ordinateur (PAO) est un logiciel de création d'éléments de communication visuel, c'est-à-dire qu'il va créer des brochures, des cartes de visite, et tout autre élément pour faire du marketing.. 22

Acronymes

API Application Programming Interfaces. 22

DOM Document Object Model. 22

PAO Publication assistée par ordinateur. 22

SVG Scalable Vector Graphics. 22

Chapitre 7

Bibliographie

7.1 Photos

Logo université montpellier

7.2 Définitions

Définition de la balise SVG

Autre définition de la balise SVG

Définition de PAO

Définition du DOM

Définition d'une API

7.3 Aides pour le code en JavaScript

Tuto JavaScript sur w3school

Informations sur le JavaScript sur la mdn web docs

Utilisation des couleurs pour les terrains

Premier pas en d3.js

7.4 Aides en LaTeX

Accents en LaTeX

Modèle de rapport

Entête et bas de page

Ajouter un glossaire