Rapport final de projet

JEU WUMPUS WORLD

Réalisé par :

- Mohammed TOUHAMI ELWAZANI
- Laila EL HAJJAMY
- Youssef OUAJDI
- Benacer CHERFAOUI

Table des matières

Introduction

1. Définition du projet
   - Enoncé
   - Plan
   - Présentation Générale
2. Cahier des charges

Analyse et Conception

1. Analyse du travail réalisé
2. Diagramme UML

Réalisation

1. Outils et langages
   - Bibliothèques graphiques
   - Langage
   - Plateforme de développement

2.Exemples du code

   - Création d'une partie
   - Perceptions
   - Déplacement du joueur
   - Actions importantes
   - L'ennemi est géré automatiquement et intelligemment
   - Scores

3.Résultat final

Conclusion

Introduction

1) Définition du projet

1-1) Enoncé

Le projet dont nous sommes en charge consiste à développer en java un jeu qui simule la recherche de l'or dans le monde du wumpus .Il doit répondre d'un coté ,aux contraintes du sujet (environnements, perceptions,…etc) , et d'un autre coté aux exigences du cahier de charger proposé .

Pour bien répondre au problème, nous allons mettre en pratique nos connaissances en programmation en java , et les connaissances acquises en suivant le cours de l'intelligence artificielle .

1-2) Plan

Par la suite nous présenterons la démarche que nous avons suivi pour tenter de mener à bien ce projet, en décrivant les deux grands axes de développement, c'est-à dire la conception et l'analyse du projet avec diagrammes UML à l'appui, la réalisation technique (choix des outils, portions de code, modèle utilisé), notre avis sur les résultats obtenus et les objectifs initiaux, puis une conclusion personnelle sur ce projet.

1-3) Présentation générale

Le jeu commence par une interface simple dans laquelle le joueur est invité à saisir un nombre qui sera l'ordre de la matrice (la taille de la caverne) . Il peut aussi afficher une fenêtre d'aide , expliquant chaque action et la touche clavier responsable de la déclencher , ainsi que les scores qu'il va perdre ou gagner selon l'état du jeu et ses actions .

Une deuxième fenêtre principale est la fenêtre du jeu , ou apparait la caverne , cette dernière se modifie au fur et à mesure que le joueur est en train de jouer .La taille de la caverne est évidement personnalisée selon le nombre saisi à l'entrée .

Nous nous étions impressionnés par des jeux wumpus bien sophistiqués et par fascination et curiosité nous voulions ajouter un autre joueur adversaire

manipulé automatiquement par le programme, et dont le but est d'augmenter le sens du défi dans le jeu .

2. Cahier des charges

Notre programme doit offrir ces fonctionnalités :

2.1. Caverne et agent :

Une caverne est générée aléatoirement comme une matrice carrée dont le nombre de lignes égale au nombre de colonnes.

La caverne doit être constituée au moins de 4x4 cases. Chaque case peut :

- Etre vide ;
- Contenir un Puits profond ;
- Contenir le Wumpus ;
- Contenir l'Or.

Dans la caverne du Wumpus il existe :

- Une seule case contenant le Wumpus ;
- Une seule case contenant de l'Or ;
- Au moins une case contenant un Puits.

La caverne peut contenir au plus n Puits tel que n est l'ordre (taille) de la matrice. Si l'agent arrive sur une case avec un Puits, il meurt et la partie s'arrête.

Si l'agent arrive sur une case avec le Wumpus, il meurt en étant mangé par le Wumpus et la partie s'arrête.

Si l'agent est sur la case où se trouve l'Or et qu'il le saisit, vous gagnez la partie.

L'agent commence à partir de la case [1,1] orienté vers l'EST.

2.2. Mesure de performance :

L'agent ramasse ou perd des points en fonction des actions qu'il exécute :

- +1000 pts s'il trouve l'Or ;

- -1000 pts s'il tombe dans un Puits ;
- -1 pt pour chaque action exécutée ;
- -10 pts s'il tire la flèche

2. 3. Perceptions et actions :

L'agent possède un ensemble de capteurs qu'ils lui permettent de percevoir plusieurs types de données :

- L'agent perçoit une Puanteur (Stench) sur la case du Wumpus et sur les cases adjacentes ;
- L'agent perçoit une Brise (Breeze) sur les cases adjacentes à un Puits ;
- L'agent perçoit une Lueur (Glitter) s'il est sur la case de l'Or ;
- Si l'agent avance dans un mur, il va percevoir un Choc (Bump);
- Lorsque le Wumpus meurt, l'agent va percevoir un Cri (Screem) ;
- Au cas où, l'agent ne perçoit aucune perception, cela veut dire que toutes les cases adjacentes sont sans danger (Ok). L'agent est capable d'effectuer les actions suivantes :
- L'agent peut avancer tout droit, tourner à gauche et tourner à droite ;
- L'agent meurt s'il tombe dans un Puits ou arrive sur la même case que le Wumpus
- Avancer n'a aucun effet s'il y a un mur devant lui ;
- L'action Ramasser permet de saisir l'Or ;
- L'action Tirer permet de tirer une flèche en avant si l'agent en possède une.
- L'action Grimper sert à sortir de la caverne à partir de la case [1,1]

2. 4 Spécifications :

Au lancement de l'application, les tâches suivantes doivent être réalisées :

- Générer automatiquement une caverne. Le nombre de lignes et de colonnes de la caverne doit être indiqué par l'utilisateur.
- Remplir aléatoirement les cases de la caverne ;
- Positionner l'agent dans la case [1,1] orienté vers l'EST. Ensuite, l'utilisateur peut commencer le jeu. Son objectif est de trouver l'Or dans la caverne du Wumpus. C'est le seul

qui décide des déplacements et des actions de l'agent. Il n'a aucune idée préalable de la position des Puits, de l'Or et bien sûr du Wumpus. Au niveau de chaque étape du jeu, il faut afficher :

- Les coordonnées actuelles de l'agent : ce sont les coordonnées de la case ;
- L'orientation de l'agent : EST, NORD, OUEST ou SUD ;
- Les percepts reçus au niveau de la position actuelle de l'agent ;
- Rappel des différentes actions que l'agent peut exécuter : Tourner à Gauche, Tourner à Droite, Avancer, Tirer, Ramasser, Grimper, Quitter (pour quitter la partie).

A la fin de chaque partie du jeu, il faut :

- Afficher le résultat du jeu: SUCCES, si l'utilisateur arrive à trouver l'OR, sinon, ECHEC

. • Afficher le score (total des points accumulés) de l'utilisateur ;

- Inviter l'utilisateur à jouer une nouvelle partie.

La fin d'une partie du jeu s'annonce dans l'un des cas suivants :

- L'agent ramasse l'Or ;
- L'agent tombe dans un puits ;
- L'agent arrive sur la même case que le Wumpus ;
- L'agent Grimpe et sort de la caverne à partir de la case [1,1]
- L'utilisateur Quitte la partie en exécutant la commande « Quitter ».

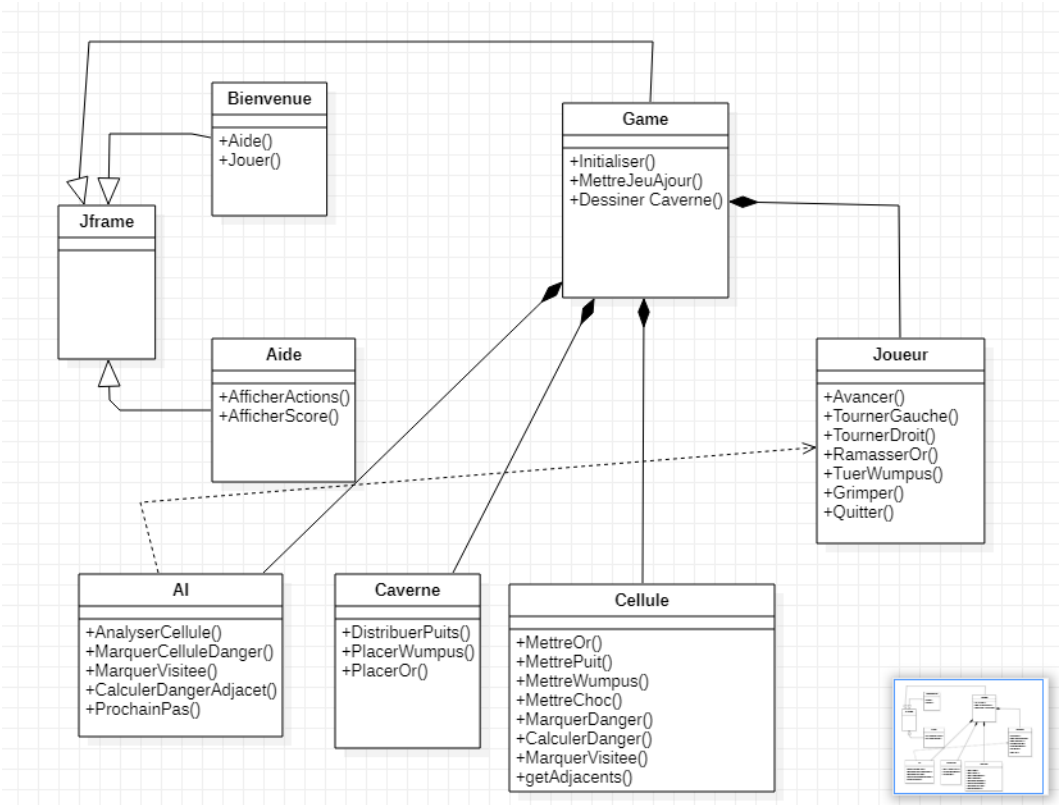
Analyse et conception

1. Analyse du travail réalisé

Un des objectifs principaux de ce projet a été de nous habituer au travail en groupe et aux notions de projet. Organiser son temps, établir les tâches ou encore nous obliger à respecter notre cahier de charge. Nous avons dans un premier temps travaillé ensemble sur les premières classes à établir (notre premier diagramme de classe en UML) afin d'établir un socle commun au sein de notre groupe en codant en dure.

2. Diagrammes UML

Après plusieurs premiers tests et après avoir imaginé de nombreux cas de figures, nous sommes arrivés à une configuration du jeu à travers le schémas suivant :



Réalisation

1. Outils et Langage

1. Bibliothèques Graphique

AWT vs SWING ? Nous allons, dans cette section, nous intéresser à la mise en œuvre d'interfaces graphiques en Java. En effet Java nous permet de créer toutes sortes d'interfaces, et d'y inclure tous types de composants des boutons, des cases à cocher, des zones de saisie de texte, des images ...etc., avec lesquelles on peut faire pas mal de choses à savoir un jeu. Mais avant d'aller plus loin il nous faut être capable de faire un choix. En effet, deux API de mise en œuvre d'interfaces graphiques sont proposées par l'environnement Java SE (Standard Edition) : l'AWT (Abstract Window Toolkit) et SWING.

- Le principal avantage à utiliser l'AWT réside dans les performances en termes d'affichage.

- L'AWT ne contient que des composants graphiques qui existent sur tous les OS. Du coup, on peut affirmer que cette librairie est relativement pauvre.

- Dans AWT c'est que Java fait appel au système d'exploitation sous-jacent pour afficher les composants graphiques.

- Dans SWING ce n'est plus l'OS qui va pixéliser les différents éléments graphiques mais bien l'API Java Swing. Du coup, si un composant graphique n'est pas proposé par un système d'exploitation, il n'y a pas de soucis.

- Une application codée en SWING consommera plus de ressources (mémoire et CPU) qu'une application codée via l'AWT.

AWT est adapté à nos besoin (performance du jeu élevé grâce à l'affichage rapide des composantes graphiques).

2. Langage

Nous sommes invités à réaliser ce projet en java donc nous sommes obligés de réaliser le projet en java, qui offre malgré tout un certain nombre d'avantages :

- Portable : exécutable sur n'importe quel système, à condition d'avoir installé une JVM).

- API : Java offre un nombre d'API qui facilite la programmation et qui correspond à notre besoin

. • Multiplateforme : Une application développer en java fonctionnera sous un système Unix, ou Windows ou n'importe quelle autre OS.

Pour développer des jeux vidéo en tant que professionnel il faut opter pour le C++ comme langage de programmation.

3. Plateforme de développement

NetBeans vs Eclipse ?

NetBeans est l'IDE supporté officiellement par Sun au contraire d'Eclipse. Donc quels sont les comparatifs entre les deux IDE :

- Eclipse est beaucoup plus utilisé (surtout dans les entreprises, ce qui est important). • Eclipse propose plus de plugins (car il est plus utilisé).

- Eclipse a une meilleure gestion des raccourcie (pour gagner du temps). • NetBeans est plus léger.

- NetBeans est plus rapide.

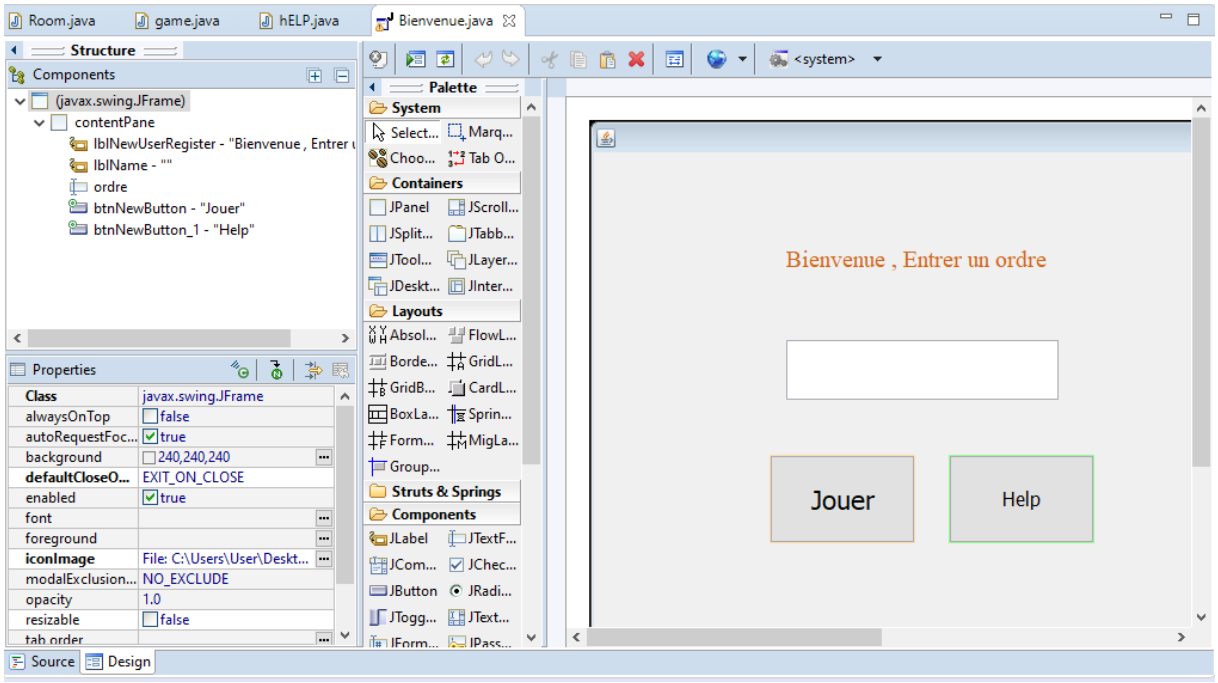
- NetBeans est plus agréable à l'utilisation.

- NetBeans a plus de plugins préinstallé (de fonction ou il faut un plugin sous Eclipse).

Donc c'est une question de choix. Nous avons l'habitude de travailler avec Eclipse, donc pour l'IDE on a choisi Eclipse.

Java WindowBuilder

WindowBuilder est construit en tant que plug-in pour Eclipse et les différents IDE basés sur Eclipse (RAD, RSA, MyEclipse, JBuilder, etc.). Le plug-in crée une arborescence de syntaxe abstraite (AST) pour naviguer dans le code source et utilise GEF pour afficher et gérer la présentation visuelle.



2 .Exemple du code

Nous n'avons pas jugé utile d'inclure l'intégralité de notre code dans ce rapport, mais nous décrirons seulement quelques extraits qui nous paraissent intéressants.

a. Création d'une partie

Une partie est créé en saisissant la taille de la caverne qui doit être entre 4 et 10, L'interface offre une zone de saisie , deux boutons pour lancer le jeu , et afficher les instructions ;

```
ordre = new JTextField();
ordre.setFont(new Font("Tahoma", Font.PLAIN, 32));
ordre.setBounds(161, 157, 228, 50);

ordre.setColumns(10);
contentPane.add(ordre);
btnNewButton = new JButton("Jouer");
btnNewButton.setBackground(new Color(255, 222, 173));
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String ordreStr = ordre.getText().toString();
         int i = Integer.parseInt(ordreStr);


        if (i>10 || i<4) {
            JOptionPane.showMessageDialog(btnNewButton, "Entrer un nombre entre 4 et 10");
        }
        if (i>3 && i<11) {
        try {
        game.NewScreen(i);

        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }}
});
```

Le nombre saisi dans le TextField , est d'abord examiné dans la condition (4<i<10) puis envoyé en paramètre dans la méthode *NewScreen()* de la classe **game** , il va être ensuite affecté au nombre de lignes et colonnes , comme suit :

```
    public static void NewScreen(int ordre) {
        //
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                game Game = new game(ordre); //
            }
        });
    }
}
```

```
public game(int ordre) {
    ROWS = ordre;
    COLS = ordre;
    CANVAS_WIDTH = CELL_SIZE * COLS;
    CANVAS_HEIGHT = CELL_SIZE * ROWS;
    squares =new DrawRoom[ROWS][COLS];
    board = new Board();
    Room startRoom = board.getRoom(0,0);
    startRoom.setHints();
    h1 = new Player(startRoom);
    h2 = new AI(startRoom);

    canvas = new DrawCanvas();
    canvas.setFocusable(true);
    canvas.setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
    canvas.setBorder(BorderFactory.createEmptyBorder(2, 5, 4, 5));
```

ROWS= ordre

COLS = ordre

La méthode *DrawCanvas( )* dessine la caverne , détermine les couleurs d'arrière plan et des frontières .

```
    public DrawRoom() {

        imageH1[0] = changeImageSize("src\\wumpus\\hunter10.png");
        imageH1[1] = changeImageSize("src\\wumpus\\hunter11.png");
        imageH1[2] = changeImageSize("src\\wumpus\\hunter12.png");
        imageH1[3] = changeImageSize("src\\wumpus\\hunter13.png");
        imageH2[0] = changeImageSize("src\\wumpus\\hunter20.png");
        imageH2[1] = changeImageSize("src\\wumpus\\hunter21.png");
        imageH2[2] = changeImageSize("src\\wumpus\\hunter22.png");
        imageH2[3] = changeImageSize("src\\wumpus\\hunter23.png");
        imageH1[4] = changeImageSize("src\\wumpus\\quitte.jpg");

        setLayout(new GridLayout(3,4));

        pics[0][0] = new JLabel(changeImageSize("src\\wumpus\\breeze.png"));
        pics[0][1] = new JLabel(changeImageSize("src\\wumpus\\stench.png"));
        pics[0][2] = new JLabel(changeImageSize("src\\wumpus\\glitter.jpg"));
        pics[0][3] = new JLabel(changeImageSize("src\\wumpus\\wall.png"));

        pics[1][0] = new JLabel(changeImageSize("src\\wumpus\\pitc.png"));
        pics[1][1] = new JLabel(changeImageSize("src\\wumpus\\roar.png"));
        pics[1][2] = new JLabel(changeImageSize("src\\wumpus\\gold.png"));
        pics[1][3] = new JLabel(changeImageSize("src\\wumpus\\ss.jpg"));

        pics[2][0] = new JLabel(imageH1[currentImageH1]);
        pics[2][1] = new JLabel(changeImageSize("src\\wumpus\\wumpusc.gif"));
        pics[2][2] = new JLabel(imageH2[currentImageH2]);
        pics[2][3] = new JLabel(changeImageSize("src\\wumpus\\rr.jpg"));
```

La méthode *DrawRoom( )*affecte les images qui s'affichent comme perceptions , détermine leurs positions dans la cellule , (on utilise GridLayout pour l'organisation des images)

```
    Random randomGenerator = new Random();
    randomOrdering = new int[ROWS*COLS - 1];
    for(int i = 0; i< randomOrdering.length; i++)
        randomOrdering[i] = i + 1; //
    for(int i = 0; i<randomOrdering.length; i++){
        int t = randomOrdering[i];
        int n = randomGenerator.nextInt(randomOrdering.length);
        randomOrdering[i] = randomOrdering[n];
        randomOrdering[n] = t;
    }


    int numPits = ROWS*COLS*PITP/100;
    for(int i = 0; i < numPits; i++){
        randomRoom(i).makePit();
    }
    randomRoom(numPits).spawnDragon();

    randomRoom(numPits + 2).depositGold();

}
```

mettre des puits , le wampus et l'or arbitrairement ,pour maintenir un niveau de jeu moyen , on a choisi que 20 pourcents des cases contiendront un puit

PITP = 20, donc vingt pourcents des cases contiendront un puit ,une seule case contiendra le wumpus et une autre contiendra l'or.

b-Perceptions

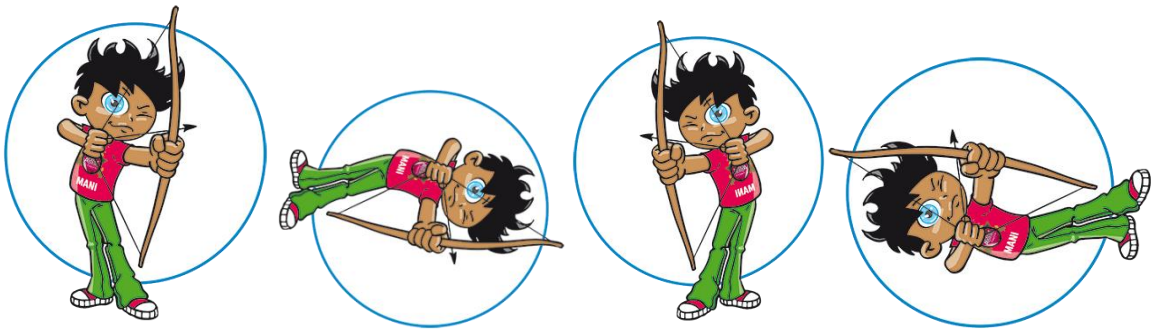
```
public void showPics(String s){
    String[] symbols = s.split("[ ]");
    for (int i = 0; i < symbols.length; i++){

        switch (symbols[i]){
        case "B": pics[0][0].setVisible(true); break;
        case "S": pics[0][1].setVisible(true); break;
        case "G": pics[0][2].setVisible(true); break;
        case "P": pics[1][0].setVisible(true); break;
        case "R": pics[1][1].setVisible(true); break;
        case "D": pics[1][2].setVisible(true); break;
        case "C": pics[0][3].setVisible(true); break;
        case "H1": pics[2][0].setIcon(imageH1[currentImageH1]);
        pics[2][0].setVisible(true); break;
        case "W": pics[2][1].setVisible(true); break;
        case "H2": pics[2][2].setIcon(imageH2[currentImageH2]);
        pics[2][2].setVisible(true); break;
        }
    }
}
```

Les images(de perceptions) sont affichées selon les perceptions détectées dans le paramètre de la méthode *showpics( ),* .

La portion du code ci-dessus , illustre qu'à chaque perception correspond une image .

Les indices *currentImageH2* ,et *currentImageH1* ,prennent les valeurs (0,1 , 2, 3, 4) , ils modifient ainsi l'orientation du joueur comme suit



La cinquième image , s'affiche lorsqu'il grimpe , elle porte la même couleur de la cellule

Exemples d’images utilisées respectivement pour les perceptions ;Glitter,Breeze ,Scream ,Stench

```
public ArrayList<Character> perceptions(){
    ArrayList<Character> perceptions = new ArrayList<Character>();
    hints = "";
    boolean pitAir = false;
    boolean stenchAir = false;

    for (int i = 0; i < neighbors.length; i++){
        if(neighbors[i]!=null){
            if(neighbors[i].hasDragon())
                stenchAir = true;
            if(neighbors[i].isPit()){
                pitAir = true;
            }
        }
    }

    if (chocAir)
        //
        perceptions.add('C');

    if (pitAir)
//
        perceptions.add('B');
    if (stenchAir)
//
        perceptions.add('S');
    if (hasDragon())
//
        perceptions.add('W');
    if (isPit())
//
```

Selon l’état des cellules adjacentes (Neighbors) , une liste de caractère (perceptions est retournée) par la méthode perception .

On avait mentionnée que ces perceptions sont passées en paramètres de la fonction *showPics*( )(classe **game**) , pour afficher les images correspondantes.

c-Déplacement du joueur

```
// le joueur ne peut pas avancer , il recoit une perception bump ,
public boolean avancer(){

 Room next = currentRoom.getNeighbors()[dir];
 if(next==null){

   currentRoom.setBump();
  return false;

 }
 else{
  currentRoom = next;
  currentRoom.setHints();
  if (currentRoom.hasDragon() || currentRoom.isPit())
   alive = false;
 }
 return true;
}

public void tournerGauche(){
 dir = dir==0?3:dir-1;
}

public void tournerDroit(){
 dir = dir==3?0:dir+1;
}

public Room getCurrentRoom(){
 return currentRoom;
}
```

si la cellule suivante (en face du joueur) est null le joueur ne peut pas avancer , il doit recevoir une perception Bump ,(Mur)

la méthode avancer retourne (true) dans le cas ou le joueur n'avance pas dans un mur , sa cellule actuelle devient la cellule en face de lui.

Sinon , la méthode retourne (false) et on met un Bump dans la cellule , à l'aide de la méthode *setBump*( ) de la classe Room

```
public void setBump(){

    this.chocAir = true;
}
```

Tourner à gauche ou à droite

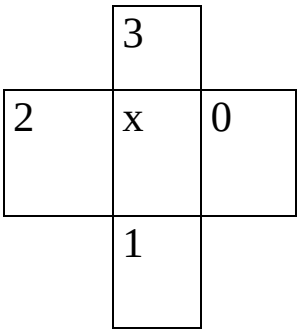
```
public void tournerGauche(){
 dir = dir==0?3:dir-1;
}

public void tournerDroit(){
 dir = dir==3?0:dir+1;
}
```

Ici dir est le numéro de la cellule en face du joueur

Si la direction en face du joueur est 0, tourner à gauche est équivalent à la direction 3 , sinon elle est équivalente à la direction actuelle +1

Contempler le schéma suivant pour mieux comprendre



d-Actions Importantes

Grimper :

```
case 'T':
        if(squares[currentPlayerLocation/ROWS][currentPlayerLocation%COLS]==squares[0][0]) //show
        {
                if(currentPlayer == h1){

                    currentImageH1 = 4;


                currentPlayer.grimper();

            }
        }

break;
```

Le joueur ne peux grimper qu'à partir de la case 1x1 , marqué par squares[0][0]

```
// sortir de la case 1x1
public void grimper(){
   grimpant = true;
}
```

```
}
else if (grimpe()) {
    currentState = GameState.H1_GRIMP;


}
```

L'action met l'état du jeu à H1_GRIMP , empêchant par la suite toute action autre que réinitialiser le jeu .

Ramasser l'or (grabGold)

```
public boolean grabGold(){
 gold = currentRoom.grabGold();
 return gold;
}
//s'il a tué le wumpus on renvoit true  false sinon
```

L'action met l'état du jeu à H1_WON ,(succes)

Tirer

```
public void shoot(){
 if(fleche==0)
  return;
 fleche --;
 Room r = currentRoom;
 while(r!=null){
  if(r.hasDragon()){
   r.killDragon();
   dragonKiller = true;
   return;
  }
  r = r.getNeighbors()[dir];
 }
}
```

L'action tirer ne peux se faire que si le nombre de flèche est 1 ou 2, à chaque fois le joueur utilise une flèche , son nombre se décrémente

e-L'ennemi est géré automatiquement et intelligemment

La classe AI (Artificial Intelligence) hérite de la classe Player , et se spécifie en plus par un ensemble de méthodes capables de déterminer la meilleure direction ,la cellule adjacente au moindre risque …etc , et par conséquent , le joueur ennemi sera une instance de cette classe et se comportera rationnellement pour défier le joueur (utilisateur). Son but est de ramasser l'or .

```java
public void analyzeRoom() {
    currentRoom.setVisited();
    currentRoom.setSafe();
    ArrayList<Character> perceptions = currentRoom.perceptions();

    if (perceptions.isEmpty()) {
        markNeighorborsSafe();
    } else {
        for (char ch: perceptions) {
            switch (ch) {
            case 'S':
                markUnSafe(ch);
                break;
            case 'B':
                markUnSafe(ch);
                break;
            case 'G':
                this.gold = true;
                break;
            }
        }
    }
}
```

on marque la cellule comme visitée et sûr si la liste des perceptions est vide , on marque les cellules adjacentes sûr. Si la liste des perceptions contient ,S(stench) ou B (breeze) on marque les cellules comme unsafe et si la liste contient un G , on met à jour le champ gold à (true),

```java
private int calculateRiskValue (Room room) {
    int riskReturn = 0;
for (Room rm: room.getNeighbors()) {
        if (rm != null && rm.getVisited()) {
            for (char danger: rm.perceptions()) {
                if (danger != ' ') {
                    riskReturn ++ ;
                }
            }
        }
    }
    return riskReturn;
}
```

le risque est 0 pour une cellule sûr, si une cellule est visitée , elle a une liste de perception soit vide soit non vide si la liste des perceptions n'est pas vide, il y

aura peut être un danger(sauf si la perception est G pour gold) on augmente alors le risque de 1 , pour chaque perception

```
private void getUnStuck() {
for (Room rm1: currentRoom.getNeighbors()) {
        if (rm1 != null) {
            rm1.unSetVisited();
            for (Room rm2: rm1.getNeighbors()) {
                if (rm2 != null) {
                    rm2.unSetVisited();
                }
            }
        }
    }
}
```

pour ne pas être bloqué dans une cellule il faut mettre l'état des cellules adjacentes comme UNVISITED , ainsi que leurs cellules adjacentes en effet , si le joueur se déplace vers une cellule adjacente cette dernière doit avoir des cellule adjacentes unvisited , pour que le joueur puisse sortir et se déplacer vers d'autres cellules si non , il reste bloqué.

```
private Room lookForSafeRoom() {
for (Room rm: currentRoom.getNeighbors()) {
        if (rm != null && rm.getDangers().isEmpty()) {
        return rm;
        }
    }
    return null;
}
```

si une cellule dans la liste des cellules adjacentes n'a pas de dangers, elle est sûr

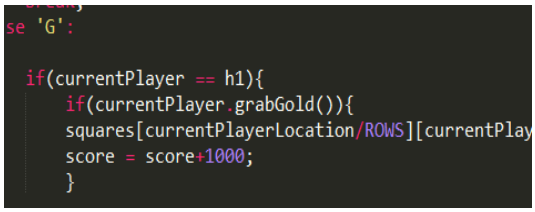
f-Scores

```
} else if (isDraw()) {
    currentState = GameState.DRAW;

    score = score-1000;
```

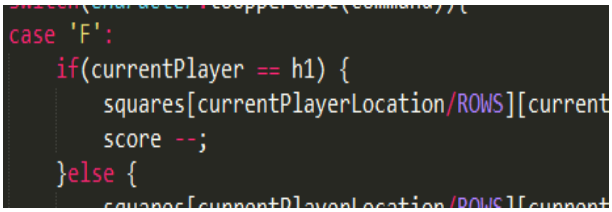
```
case 'S':

    currentPlayer.shoot();
    if((currentPlayer == h1) && (fleche < 2) )
    {   score = score-10;
        fleche ++;
```



```
se 'G':

  if(currentPlayer == h1){
      if(currentPlayer.grabGold()){
      squares[currentPlayerLocation/ROWS][currentPlay
      score = score+1000;
      }
```



```
case 'F':
    if(currentPlayer == h1) {
        squares[currentPlayerLocation/ROWS][current
        score --;
    }else {
```

Tirer une flèche paye le joueur 10 points

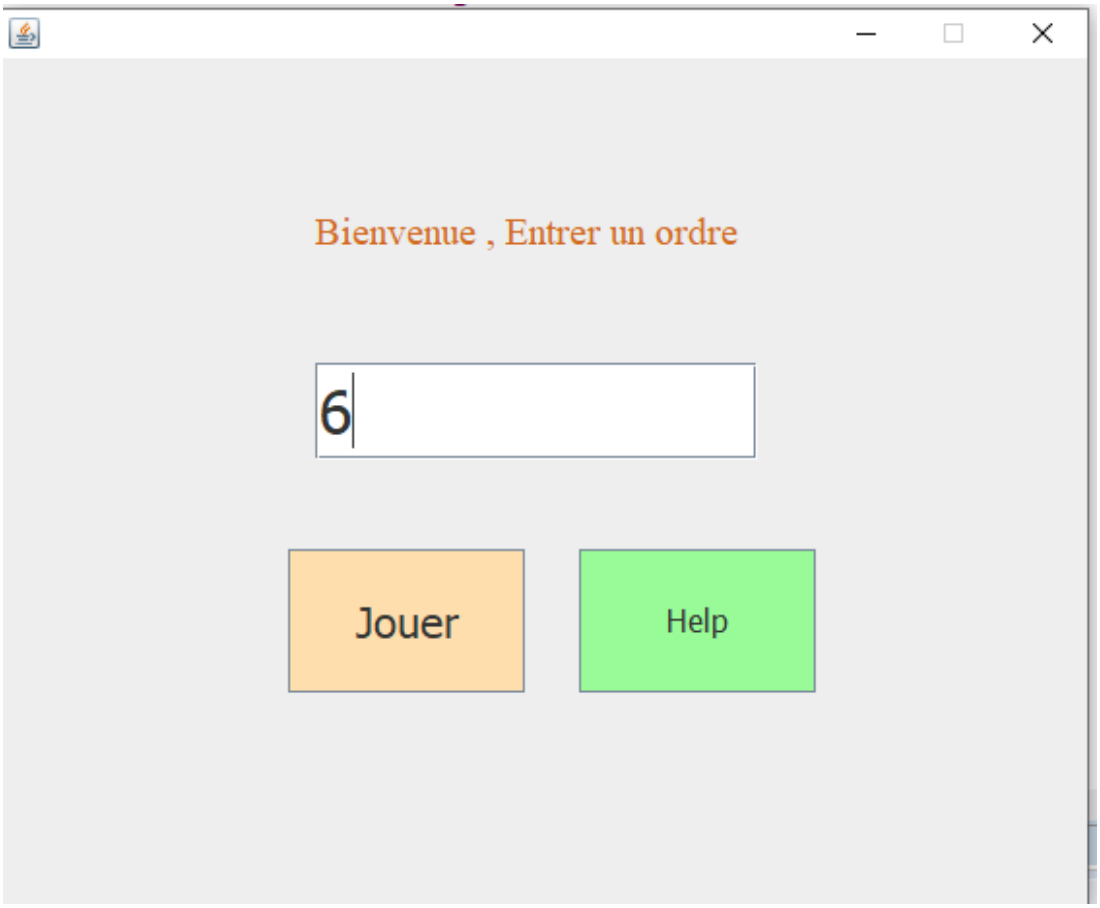
Ramasser l'or, lui donne +1000 points

Tomber dans un puit ou dans la même cellule que le wumpus , lui paye 1000 points , et sa vie ☹

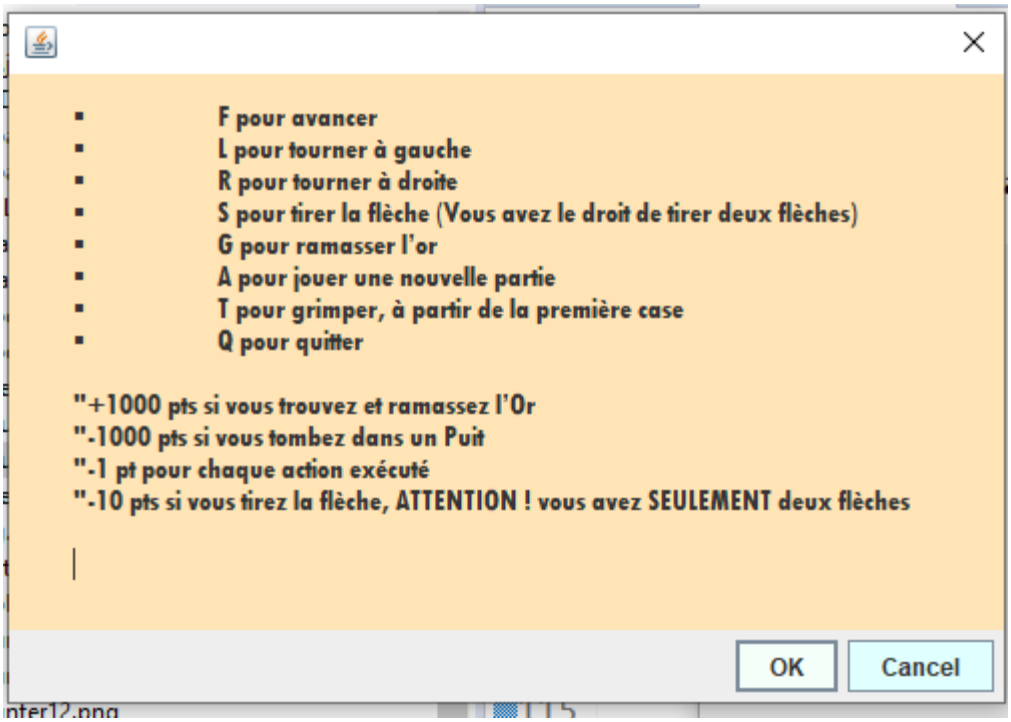
Chaque action , lui paye 1 point ,(avancer, tourner à gauche, tourner à droite)

Résultat Final

Fenêtre de bienvenue



Si on clique sur Help, la fenêtre suivante s'affiche, portant toutes les instructions du jeu.

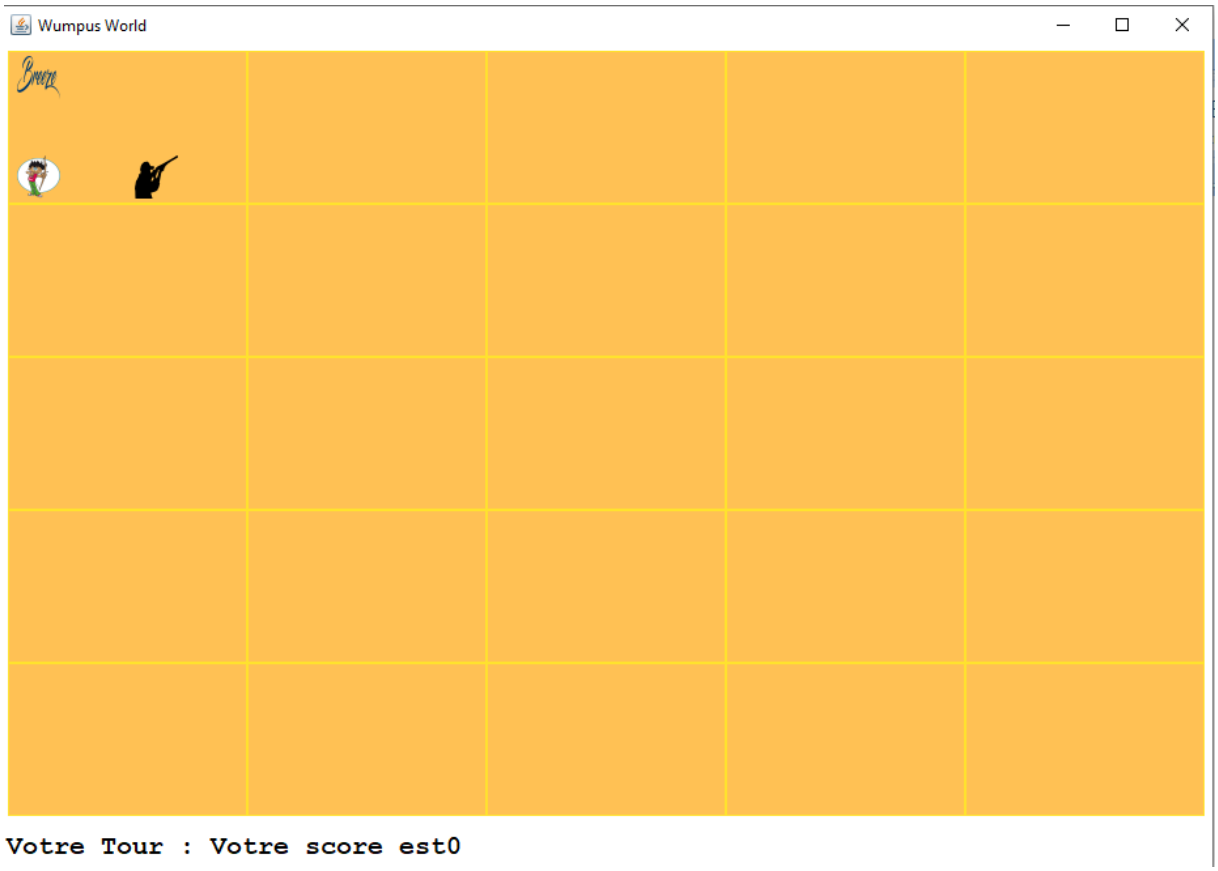

F pour avancer
L pour tourner à gauche
R pour tourner à droite
S pour tirer la flèche (Vous avez le droit de tirer deux flèches)
G pour ramasser l'or
A pour jouer une nouvelle partie
T pour grimper, à partir de la première case
Q pour quitter
"+1000 pts si vous trouvez et ramassez l'Or
"-1000 pts si vous tombez dans un Puit
"-1 pt pour chaque action exécuté
"-10 pts si vous tirez la flèche, ATTENTION ! vous avez SEULEMENT deux flèches
OK
Cancel


Bienvenue , Entrer un ordre
11
Message
Entrer un nombre entre 4 et 10
OK
Help

Si l’utilisateur fait entrer un nombre inférieur à 4 ou supérieur à 10 , on lui affiche un message d’alerte .

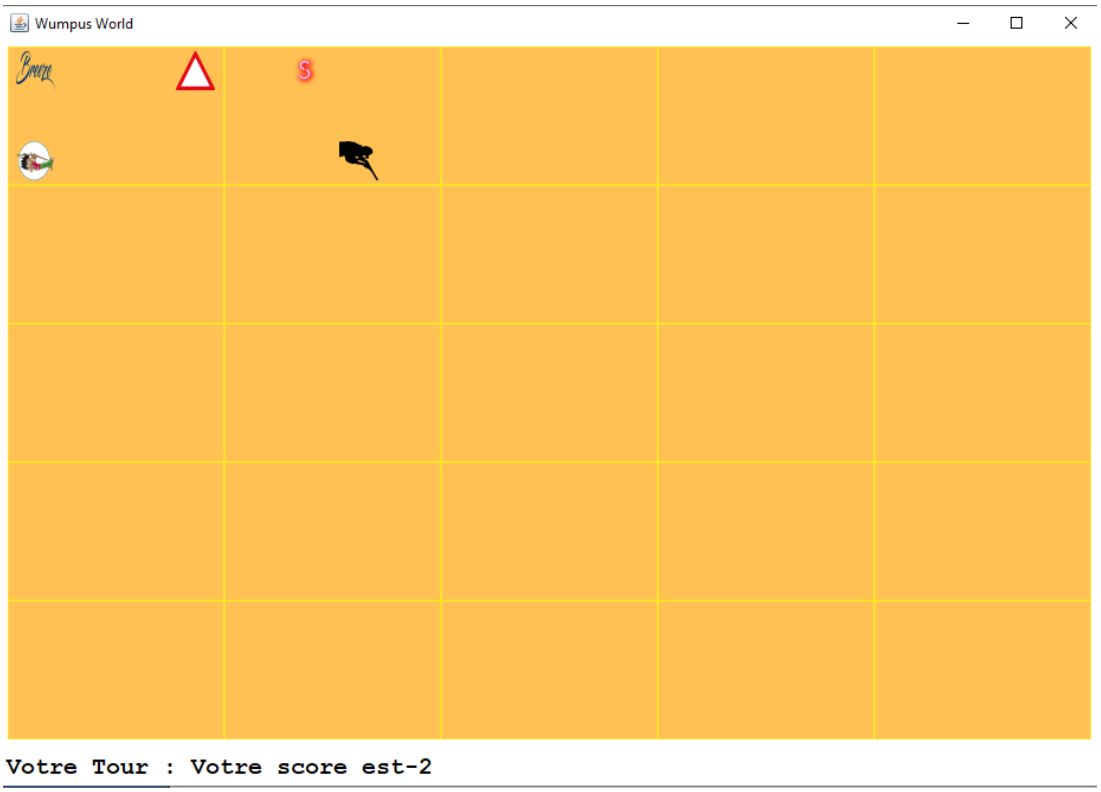
```
Votre Tour : Votre score est0
```

Exemple d’une caverne de taille 10

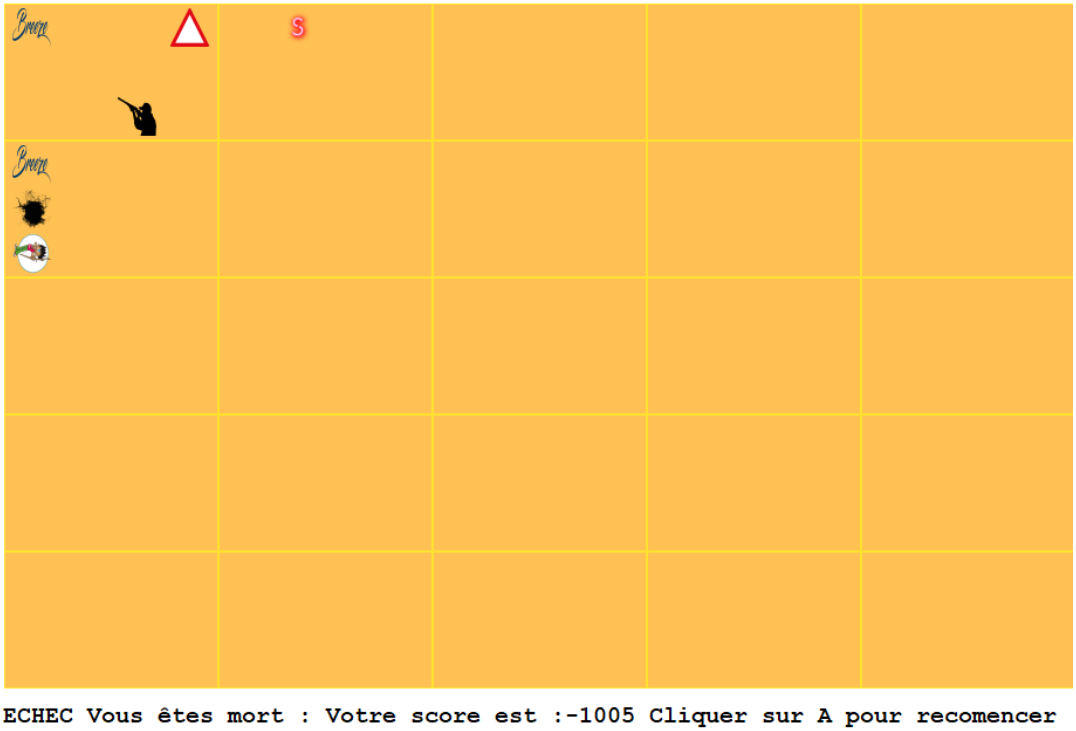


Au début du jeu ,le score est initialisé à 0 , le joueur est dans la case 1x1 , orienté vers l'est

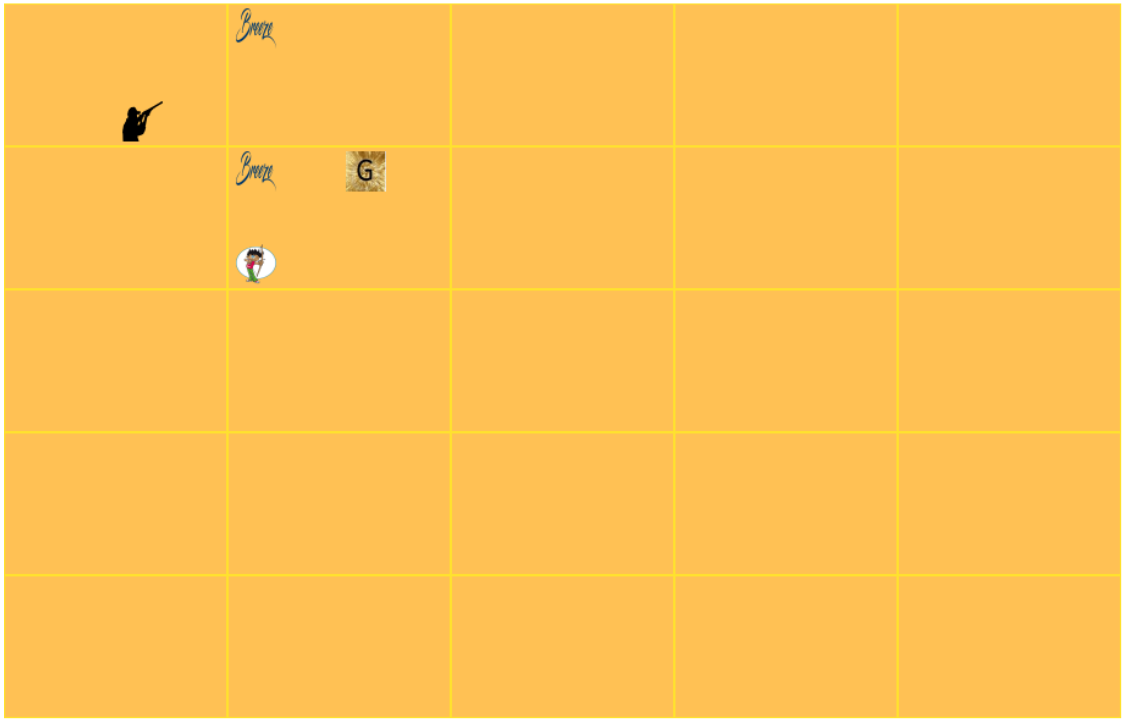
S’il grimpe, il ne peut plus faire aucune autre action , mais il peut recommencer le jeu



Le joueur ici a tourné à gauche , puis avancé dans le mur , son score est décrémenté de deux points (deux actions) , et un Bump (choc) apparait

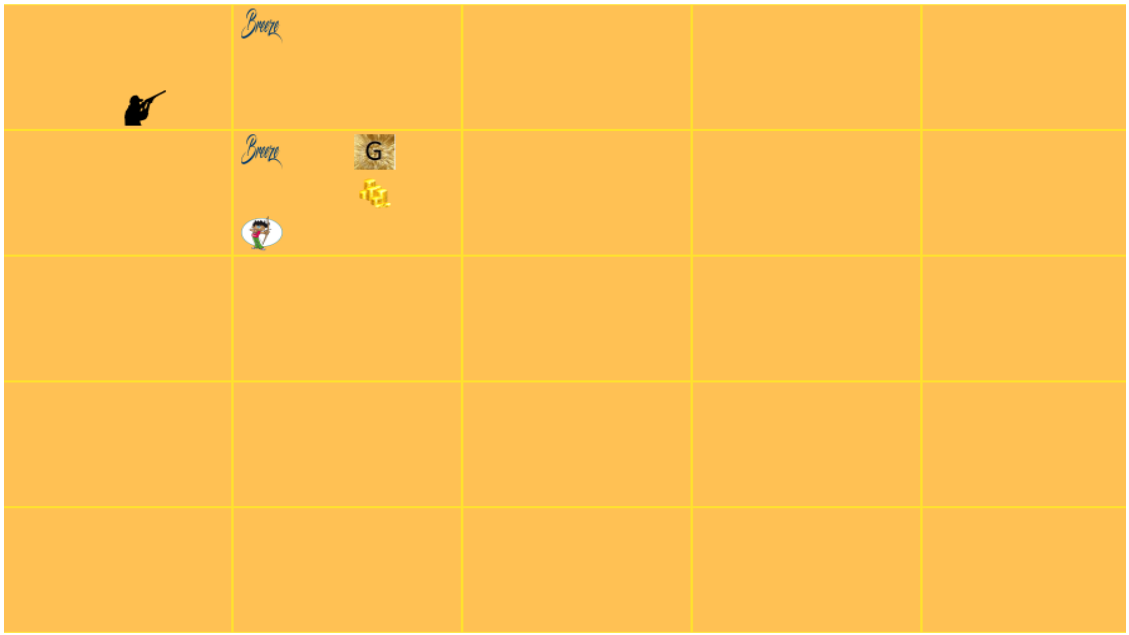


Dans ce cas le joueur est tombé dans un puit , son score se décrémente de 1000 points



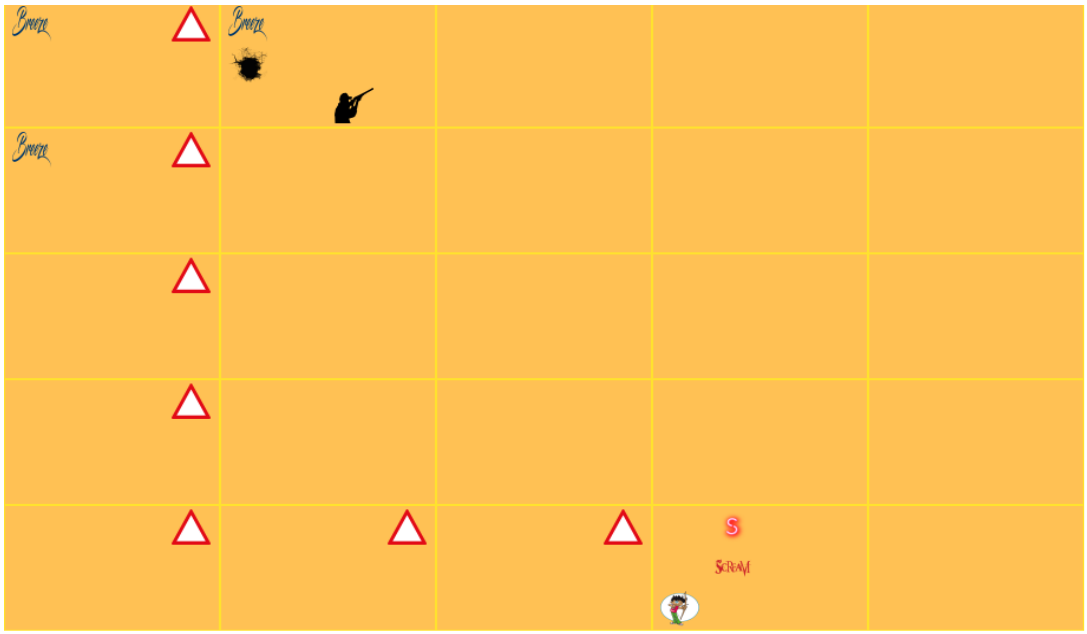

```
Votre Tour : Votre score est-6
```

La cellule actuelle du joueur contient l'or , le joueur reçoit un Glitter

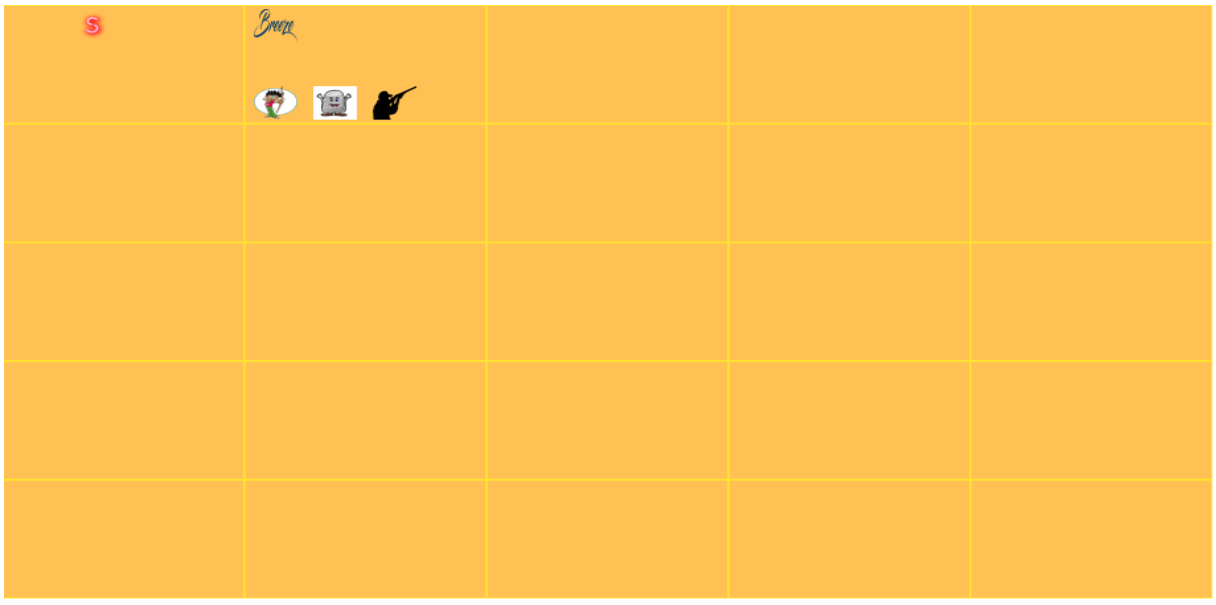



```
SUCCES Félicitations! : Votre score est :994 Cliquer sur A pour recomencer
```

Le joueur a ramassé l'or, il gagne la partie, l'état du jeu est SUCCES ,son score est augmenté de 1000 ponts, et il est invité à jouer une nouvelle partie .

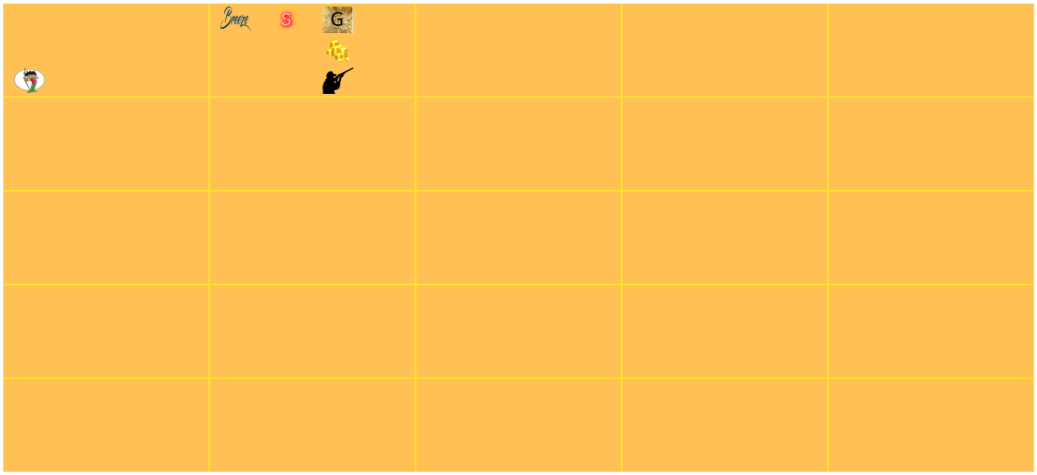


Si l'utilisateur avance dans un mur , il reçoit un CHOC , et ceci est pour toutes les cellules aux frontières de la caverne



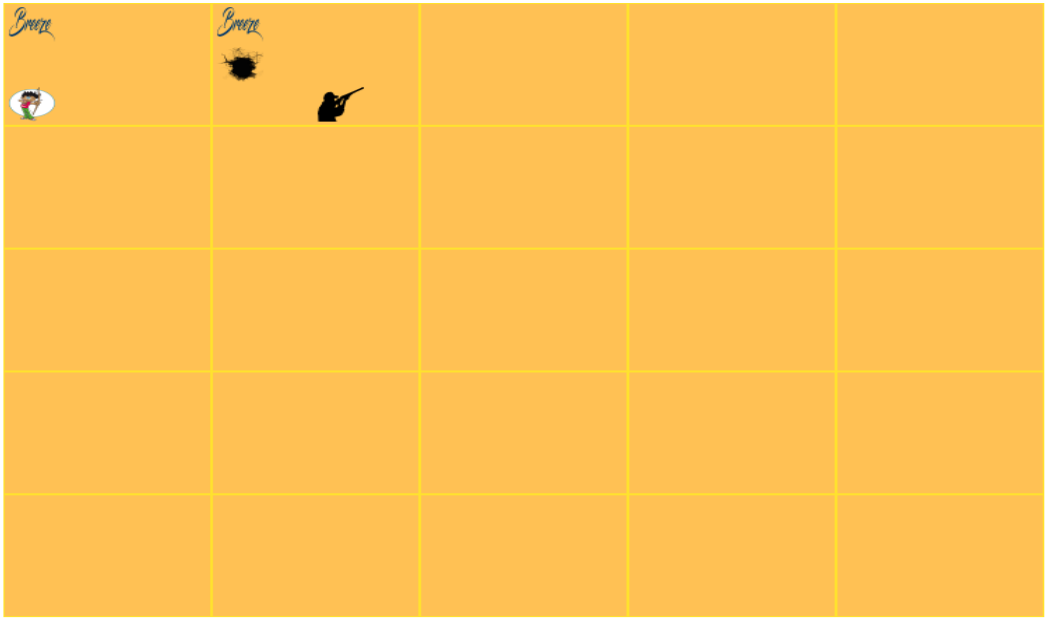
ECHEC Vous êtes mort : Votre score est :-1005 Cliquer sur A pour recomencer

S'il est mangé par le wumpus, son score est décrémenté de 1000 points, l'état du jeu est ECHEC, et il est invité à jouer une nouvelle partie.



ECHEC Dommage ,Chasseur noir a gagné! : Votre score est :-2 Cliquer sur A pour recomencer

Dans le cas ou c'est l'adversaire qui ramasse l'or , l'état du jeu est ECHEC , le joueur perds



Votre Tour : Votre score est-20

Dans ce cas , l'utilisateur a cliqué deux fois sur S, (SHOOT) , le joueur alors , a utilisé toutes ses flèches (il en a que deux). Il ne peut plus tirer

Conclusion

Au terme de ce projet, notre sentiment est très positif ; nous avons acquis de nombreuses connaissances et compétences nouvelles. Nous avons pu mettre en corrélation les aspects théoriques et pratiques. Tout en restant positif, il ne faut pas occulter la difficulté essentielle que nous avons rencontrée dans ce projet, à savoir les multiples fonctionnalités qu'offrent Java

Les circonstances imposées par la pandémie Corona Virus a rendu difficile, la gestion des codes réparties, mais grâce à de nombreuses réunions Meet , nous avons pu arriver à unifier les codes , et aboutir à un projet complet .

En fin, ces perpétuelles difficultés ont été d'une grande importance, car elles nous ont donné l'opportunité de visualiser plusieurs chemins de solutions du projet réalisé

Fin