

# Analyse Puyo Puyo et Super off road

Rachel Humbert   Emma Mange   Alphée Grosdidier  
Anton Dolard   Silvio Vescovo

Université de Franche-Comté

2020 – 2021

## 1 Puyo Puyo

## 2 Super off road

# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes



- Jeu de type Puzzle
- Développé par *Compile*
- Sorti en 1991

# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Les structures et variables

## ● Les variables du jeu

- constante entier NBCOLORS
- constante entier HEIGHT
- constante entier WIDTH
- constante entier FALLSPEED
- constante entier SIZEPUYO
- constante entier WIDTHMAT
- constante entier HEIGHTMAT
- constante entier WAITINGPOSX
- constante entier WAITINGPOSY
- booleen doStartTour
- booleen gameOver
- entier penaltyReps1
- entier penaltyReps2

# Les structures et variables

- Structures générales (qui ne représentent pas un objet du jeu)
  - structure Position
- Structures qui représentent un élément du jeu
  - structure Block
  - structure BlockFall
  - structure Player
  - structure Game

# Block & BlockFall

```
Struct Block
    boolean exist
    caractere color
    entier groupID
finStruct

Struct BlockFall
    entier orient
    caractere color1
    caractere color2
    Position pos
    Position posMat
    entier speed
finStruct
```



# Player & Game

```
Struct Player
    BlockFall bf1
    BlockFall bf2
    Block[][] blocks
    entier score
finStruct
```

```
Struct Game
    Player p1
    Player p2
finStruct
```

# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# État initial



# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- **Évolution du jeu**
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Pression sur L ou R

```
Si (blockTest(p1.blocks, p1.bf1.posMat.x, p1.bf1.posMat.y
    -1) ET (p1.bf1.posMat.y >0) ) alors
    p1.bf1.posMat.y <- p1.bf1.posMat.y - 1
    p1.blocks[p1.bf1.posMat.x][p1.bf1.posMat.y].color <- p1.
        bf1.color1
    p1.blocks[p1.bf1.posMat.x][p1.bf1.posMat.y].exist <- vrai
    p1.blocks[p1.bf1.posMat.x][p1.bf1.posMat.y + 1].color <-
        v
    p1.blocks[p1.bf1.posMat.x][p1.bf1.posMat.y + 1].exist <-
        faux
FinSi
```

# La touche D

## Pression sur D

```
p1.bf1.speed <- p1.bf1.speed * 2
```

## Relâchement de D

```
p1.bf1.speed <- FALLSPEED
```

# Appui sur une flèche directionnelle

Pour la flèche gauche :

```
Si (blockTest(p1.blocks, p1.bf1.posMat.x, p1.bf1.posMat.y  
-1) ET ( p1.bf1.posMat.x > 0))  
    p1.bf1.orient <- 0  
finSi
```

On fera de même pour la flèche haut, droite et bas. Nous attribuerons respectivement à p1.bf1.orient 1, 2 ou 3, et modifierons les conditions de manière à ce qu'elles correspondent à chaque direction.

# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

## Règles du jeu

Le jeu se termine une fois que les nouveaux Puyo ne peuvent plus accéder à la grille d'un joueur, c'est-à-dire quand les deux blocs où ils apparaissent sont déjà occupés par d'autres Puyo qui ne sont pas détruits par combinaisons et lorsque la colonne est remplie d'autres Puyo. Le joueur a alors perdu, et son adversaire gagne



# Plan

## 1 Puyo Puyo

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# continueFall

```
boolean fonction continueFall (entier[][] blocks, BlockFall bf)
    boolean test
    dansLeCasDe (bf.orient)
        cas 0 :
            test <- blockTest(blocks, bf.posMat.x, bf.posMat.y
                + 1) OU blockTest(blocks, bf.posMat.x - 1, bf.
                posMat.y + 1) OU (bf.posMat.y < HEIGHTMAT -1)
        //Pour la valeur test le cas 0 et 2 sont semblables
        cas 1 :
            test <- blockTest(blocks, bf.posMat.x, bf.posMat.y
                + 1) OU (bf.posMat.y < HEIGHTMAT -1)
        //Pour la valeur test le cas 1 et 3 sont semblables
        finCasDe
        retourner test
finFonction
```

# blockDown

```
action blockDown (->Player p1->)
    dansLeCasDe (p1.bf1.orient)
        cas 0 :
            blocks[p1.bf1.posMat.x][p1.bf1.posMat.y - 1].color <- p1.
                bf1.color2
            blocks[p1.bf1.posMat.x][p1.bf1.posMat.y - 1].exist <-
                true

// On changera respectivement pour 1, 2 et 3 la position
dans la matrice [p1.bf1.posMat.x][p1.bf1.posMat.y + 1],
[p1.bf1.posMat.x + 1][p1.bf1.posMat.y] et [p1.bf1.
posMat.x - 1][p1.bf1.posMat.y]

    finCasDe
finAction
```

# AttribuerGroupe

```
entier fonction attribuerGroupe(->Blocks[][] Matrice->, ->
    Position posBlock,-> entier groupID)
entier x, y
entier longueurChaine <- 0
Pour i allant de 0 a 3 pas de 1 faire
    si (i%2 == 0) alors
        x <- posBlock.x +i -1
        y <- posBlock.y
    sinon
        x <- posBlock.x
        y <- posBlock.y +i -2
finSi
```

```
si (x >= 0 ET x < longueur(2, Matrice) ET y >= 0 ET y
    <longueur(1, Matrice)) alors
    si (Matrice[y][x].exist) alors
        si (Matrice[posBlock.y][posBlock.x].couleur ==
            Matrice[y][x].couleur && Matrice[y][x].groupID == 0) alors
            Matrice[y][x].groupID <- groupID
            Position pos <- new Position()
            pos.x <- x
            pos.y <- y
            longueurChaine += (1 + AttribuerGroupe(
                Matrice, pos, groupID))
        finSi
    finSi
finSi
finPour
retourner longueurChaine;
finFonction
```

# resetBlocksForID

```
action resetBlocksForID(->block[][] block ->, ->k)
    pour i de 0 a (WIDTHMAT -1) pas de 1 faire
        pour j de 0 a (HEIGHTMAT -1) pas de 1 faire
            si (block[i][j].blockID == k ET block[i][j].color
                != w) alors
                block[i][j].exist <- faux
                block[i][j].color <- v
                si (block[i][j+1].color == w) alors
                    block[i][j+1].exist <- faux
                    block[i][j+1].color <- v
//On regarde aussi les 3 autres blocks a cote et on verifie
    s'il faut les detruire
        finSi
        finSi
    finPour
finPour
finAction
```

# destroyBlock

```
entier fonction destroyBlock(->block[][] mat ->)
    entier i, j, k, nbrChain
    pour k de WIDTHMAT*HEIGHTMAT a 1 pas de -1 faire
        si (countNbBlocksEqualID(mat, k) > 3) alors
            resetBlocksForID(mat, k)
            nbrChain <- nbrChain +1
        finSi
    finPour
    pour i de 0 a (WIDTHMAT -1) pas de 1
        pour j de 0 a (HEIGHTMAT -1) pas de 1 faire
            mat[i][j].blockID <- 0
        finPour
    finPour
    retourner nbrChain
finFonction
```

# setClearBlocksOnPlayer

```
action setClearBlocksOnPlayer (->Player p1->, -> entier reps)
    entier i, nb
    pour i de 0 a (reps) Pas de 1 faire
        nb <- random0toNb(WIDTHMAT)
        si (p1.blocks[0][nb].exist == faux) alors
            p1.blocks[i][0].color <- w
            p1.blocks[i][0].exist <- vrai
            doGravityOnAll(p1.blocks)
        finSi
    finPour
finAction
```

# Boucle Principale I

```
faire
    si (doStartTour) alors
        startTour()
        doStartTour <- faux
    finSi
    //On inserera ici le resultat des actions des joueurs

    doGravityOnAll(p1.blocks)
    si (!continueFall (p1.blocks , p1.bf1))
        blockDown(p1)
        doGravityOnAll(p1.blocks)
    faire
        checkAllChains()
        nbCombinations <- destroyBlock(p1.blocks)
        doGravityOnAll(p1.blocks)
        si (nbCombinations !=0) alors
            penaltyReps2 += puissance(2, nbCombinations)
        finSi
    tantQue (nbCombinations != 0)
```

# Boucle Principale II

```
si (penaltyReps1 > 0) alors
    setClearBlockOnPlayer(pi, penaltyReps1)
    penaltyReps1 <- 0
finSi
si (!blockAtStart) alors
    doStartTour <- true
sinon
    gameOver <- true
finSi
finSi
tantQue (!gameOver)
```

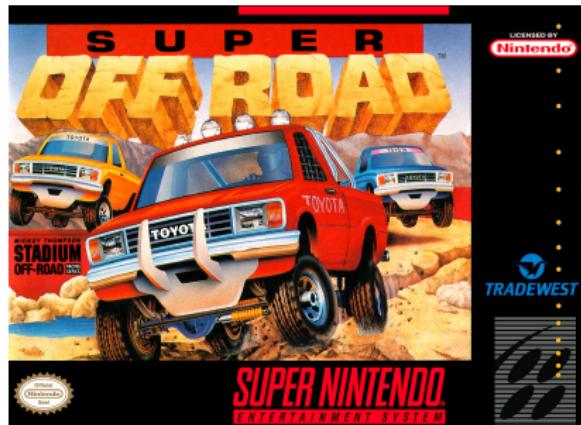
## 1 Puyo Puyo

## 2 Super off road

# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes



- Jeu de course
- Développé par Leland Corporation
- Sorti en 1989 sur borne d'arcade

# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Les structures et variables

- Variables générales du jeu

- constante entier ACCELERATION
- constante entier NB\_LAPS
- constante entier NITRO\_TIME
- constante entier NITRO\_WIDTH
- constante entier NITRO\_SPAWN\_TIME
- constante entier CAR\_WIDTH
- constante entier CAR\_HEIGHT
- Car playerCar
- Bonus [] nitroList
- entier malusBonusSpeed <- 1

# Les structures et variables

- Structures générales (qui ne représentent pas un objet du jeu)
  - structure Position
  - structure Hitbox2P
  - structure Hitbox4P
  - structure Speed
- Structures qui représentent un élément du jeu
  - structure Flag
  - structure Wall
  - structure Car
  - structure Bonus
  - structure Mud
  - Structure Ground

# Hitbox2P et Hitbox4P

```
Struct Hitbox2P
    Position corner1
    Position corner2
finStruct
```

```
Struct Hitbox4P
    Position corner1
    Position corner2
    Position corner3
    Position corner4
finStruct
```

# Flag & Wall

```
Struct Flag
    Hitbox2P hitbox
    entier nb
finStruct

Struct Wall
    Hitbox2P hitbox
    entier directionStop
finStruct
```



# Car & Bonus

```
Struct Car
    Position pos
    Speed speed
    entier direction
    entier laps
    Hitbox4P hitbox
    entier flag
    entier nbNitro
finStruct
```

```
Struct Bonus
    Position pos
    Hitbox4P hitbox
finStruct
```



# Mud & Ground

```
Struct Mud
    Position pos
    Hitbox4P hitbox
finStruct

struct Ground
    Wall [] walls
    Position [] spawnPosNitro
    Mud [] muds
    Flag [] flags
finStruct
```



# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# État initial



# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- **Évolution du jeu**
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Actions utilisateurs

```
si (key 'z' is pressed) alors
    haut <- true
finSi
si (key 'z' is released) alors
    haut <- false
finSi
```

On fait de même avec les touches q, s, d et spacebar où l'on change respectivement les valeurs de *gauche*, *bas*, *droite* et *nitro*.

```
si (gauche) alors
    playerCar.direction <- (playerCar.direction -1)%16
finSi
si (droite) alors
    playerCar.direction <- (playerCar.direction +1)%16
finSi
```

# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# Condition de victoire

Le joueur gagne lorsqu'il est le premier à effectuer les 3 tours de circuit dans le sens imposé. Pour cela, il peut utiliser de l'argent avant la course pour s'acheter des améliorations, et peut aussi utiliser les boosts disponibles sur le terrain durant la course.



# Plan

## 2 Super off road

- Présentation du jeu
- État du jeu
- Configuration initiale du jeu
- Évolution du jeu
  - Action du joueur
  - Règles du jeu
- Algorithmes

# calculateSpeed

```
fonction Speed calculateSpeed ( -> Car car , -> entier
acceleration , -> entier avgAcceleration , -> boolean
isAccelerate , -> boolean isBreak , -> boolean isNitro ,
-> reel dt )
```

On calcule la nouvelle vitesse de la voiture en fonction de son accélération (qui dépendra des obstacles) ainsi que des actions de l'utilisateur.

# isCollision

```
fonction boolean isCollision(-> Position[] verticle1, ->
    Position [] verticle)
Position [] listAxis
listAxis <- getAxisList(verticle1)
boolean collision
entier count
pour i de 0 a longueur(listAxis) pas de 1 faire
    si (isCollisionAxis(listAxis[i], verticle1, verticle2)
        ) alors
            count ++
    finSi
finPour
si (count == longueur(listAxis)) alors
    collision <- true
finSi
retourner collision
finFonction
```

# countTour

```
fonction entier countTour(->Car car,->Flag flag,->int nbFlag
)
    si (flag.nb == 0 ET car.flag == nbFlag -1) alors
        car.laps++
        car.flag <- 0
    sinon si (flag.nb - car.flag == 1) alors
        car.flag++
    finSi
    retourner car.laps
finFonction
```

# spawnNitro

```
fonction Bonus generateNitro(-> Position[] spawnNitro, ->
    Bonus[] nitroList)
    Bonus nitro
    boolean present <- faux
    faire
        entier hasard <- random(0,longueur(spawnNitro))
        nitro.pos.x <- spawnNitro[hasard].pos.x
        nitro.pos.y <- spawnNitro[hasard].pos.y
        pour i allant de 0 a longueur(nitroList) faire
            si (nitroList[i].pos.x == nitro.pos.x ET nitroList[
                i].pos.y == nitro.pos.y) alors
                present <- vrai
            finSi
        finPour
    tantQue (present)
    retourner nitro
finFonction
```

## Conclusion

Des questions ?