



Bienvenue au Hackathon 2019 du DIRO, à l'Université de Montréal!

Ce document vous guidera à travers l'évènement, où vous et votre équipe allez programmer une version interactive du très connu niveau 1-1 du jeu **Super Mario Bros**.

An **English** version of the instructions is [available here](#).

Warning: *note that these are the instructions for an older hackathon, while most of the objectives are still correct, some of the information might not be relevant for this year's Hackathon.*

In doubt, don't hesitate to ask questions to any of the organizers.

Table des matières...

- [Avant de commencer...](#)
- [Objectif 1 : Créer un simple monde de tuiles](#)
- [Objectif 2: Collisions](#)

- Objectif 3 : Charger le niveau
- Objectif 4 : Gestion avancée du niveau
- Objectif 5 : Faites bouger Mario!
- Objectif 6 : Ajouter des items et des ennemis
- Objectif 7 : Que Mario fracasse des trucs !
- Objectif 8 : Effets spéciaux
- Bonus

Avant de commencer...

Voici quelques informations importantes sur le défi :

Développement

Nous avons découpé le développement du jeu en 8 étapes qui vous amèneront progressivement à un jeu fini. Une fois les étapes fondamentales complétées, vous étendrez le jeu de plusieurs façons amusantes. Vous devriez séparer les tâches entre vos coéquipiers, et nous suggérons de suivre les indications suivantes:

	Étapes fondamentales			Mécaniques de jeu			Cerise sur le gâteau		
Objectif	1	2	3	4	5	6	7	8	Bonus
Dépend de...	Rien	1	1	1 à 3	1 et 2	1 à 4	1 à 6	1 à 7	1 à 8

Présentations

À la fin de l'évènement, chaque équipe aura 2 minutes pour montrer son jeu à la foule (et aux juges) ! Vous serez évalués selon divers critères, dont la proximité au jeu original et la qualité des fonctionnalités supplémentaires. La structure et la qualité du code, la créativité, la fidélité au jeu original et la distribution du travail seront utilisés en cas d'égalité.

Jeu original

Si vous n'avez jamais joué à *Super Mario Bros* (vraiment!?) ou si vous voulez vous rafraîchir la mémoire, vous pouvez passer voir un des responsables de l'événement qui vous fera jouer au Super Mario Bros original.

Processing

Vous coderez avec *Processing*, un langage et environnement de développement axé sur le développement de jeux et d'animations graphiques.

Si *Processing* n'est pas déjà installé sur votre ordinateur, rendez-vous sur <https://processing.org/download/> avant de commencer.

Pour plus d'information sur le langage et pour avoir accès à la documentation : <https://processing.org/>.

Code de base

Nous vous fournissons une base de code pour vous aider à démarrer avec une bonne organisation.

Téléchargez ce fichier zip, dézippez le code, ouvrez `SuperMarioBros/SuperMarioBros.pde` dans Processing (en double-cliquant dessus) et appuyez sur "Run" : vous devriez voir s'ouvrir une fenêtre contenant une image. Sinon, faites nous signe rapidement.

Structure

Voici la structure du code de base et des données (et vous ne devriez pas avoir à modifier cette structure) :

```
SuperMarioBros/  
  data/  
    img/           // images du jeu  
    level/         // tuiles, ennemis et leurs images  
  SuperMarioBros.pde // liaison avec Processing  
  Game.pde         // classe Game, contrôle général  
  Level.pde        // cellules et leurs états
```

```
Player.pde           // position et états de Mario
Body.pde             // classe parent pour éléments
Enemy.pde            // classe Enemy (goombas, etc.)
Tile.pde             // classes tuiles (solid, etc.)
Item.pde             // classes items (cents, etc.)
Trigger.pde          // évènements (e.g. ennemi créé)
Animation.pde        // effets spéciaux
Utils.pde            // utilitaires, ne pas modifier
```

Sauvegardes

Après complétion d'un objectif, **sauvegardez votre travail** : sauvez une copie datée de votre code (avec code et données) dans votre Dropbox, Google Drive, ou ailleurs sur vos ordinateurs.

Notez quelles étapes du code sont complétées dans chaque sauvegarde, pour que vous sachiez facilement à quelle version revenir pour tester des comportements antérieurs.

Les sauvegardes sont une solution de secours, et permettent aux organisateurs de voir votre progrès et de vous aider à résoudre des problèmes.

Ne sous-estimez pas les sauvegardes : sans elles, de petites erreurs d'inattention peuvent suffire à vous faire perdre des heures de travail.

À chaque fois que vous complétez un objectif, copiez votre code quelque part pour pouvoir revenir à une version précédente en cas de problème

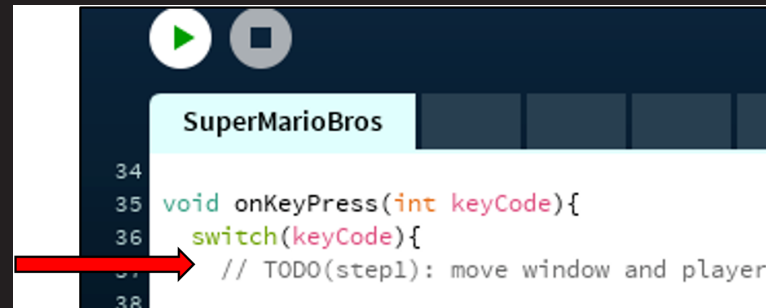
Séparation & fusion du code

Si vous partagez du code sur Google Drive, Dropbox ou autre partage de fichier similaire, assurez-vous de faire attention à ce que deux personnes **ne modifient pas le même code en même temps**.

Combiner des fichiers peut être délicat, donc communiquez bien avec vos coéquipiers pour ne pas briser leur code. **Les sauvegardes seront utiles ici.**

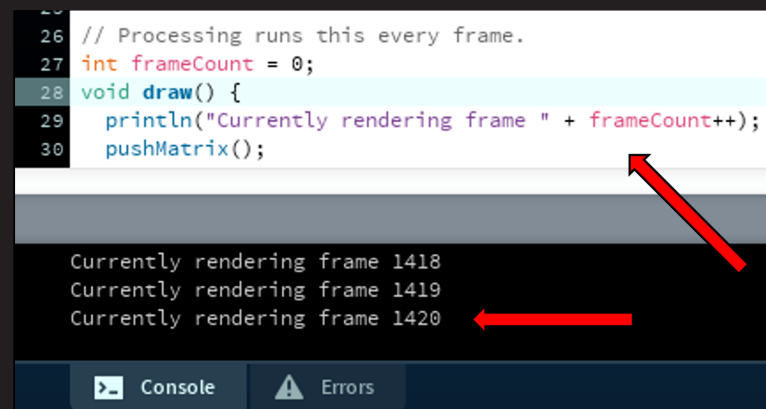
Objectif 1 : Créer un simple monde de tuiles

Familiarisez-vous au code en travaillant à quatre pour compléter le premier objectif : créer un damier et un "joueur" rectangulaire qui s'y promène.



```
34  
35 void onKeyPress(int keyCode){  
36   switch(keyCode){  
37     // TODO(step1): move window and player  
38
```

Conseil 1 : Recherchez les TODO dans le code.



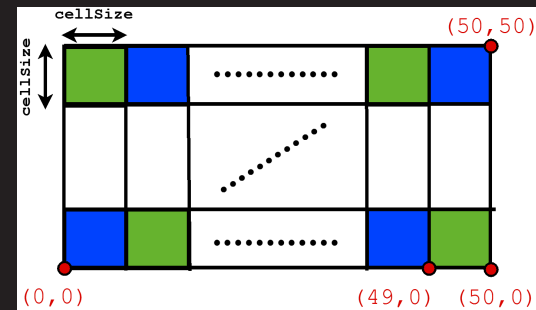
```
26 // Processing runs this every frame.  
27 int frameCount = 0;  
28 void draw() {  
29   println("Currently rendering frame " + frameCount++);  
30   pushMatrix();  
}
```

Currently rendering frame 1418
Currently rendering frame 1419
Currently rendering frame 1420

Conseil 2 : Affichez à la console avec println(...).

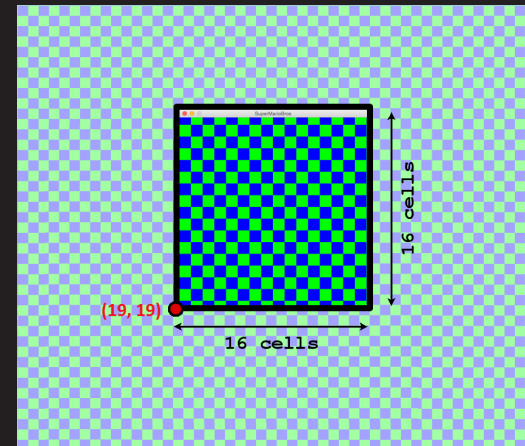
1. Vous allez construire le monde 2D de Mario à partir de cellules : 1×1 mètre carré, alignées sur une grille. Modifiez `Game.draw()` pour créer un tableau de 50×50 cellules en un damier bleu et vert. Assurez-vous de dessiner des tuiles sans bordure à l'aide de `noStroke()`.

Une cellule est représentée par `cellSize × cellSize` pixels sur l'écran, autrement dit, 1×1 mètre carré du monde de Mario correspond à `cellSize` pixels sur l'écran.



2. En tout temps, votre **fenêtre** de jeu n'affichera que 16×16 cellules sur l'entièreté du damier **monde** de 50×50 . Éventuellement, quand Mario se déplacera dans le monde, la fenêtre devra se déplacer avec lui. Commencez par redimensionner et positionner la fenêtre à (19, 19).

La position de la fenêtre peut être modifiée en changeant la valeur de l'attribut `window.bottomLeft`, qui est un vecteur correspondant à la coordonnée (x, y) du coin en bas à gauche de la fenêtre. Vous pouvez créer un vecteur en utilisant `new Vec2(valeurX, valeurY)`.



3. Ensuite, créez un joueur très simple dans notre monde : premièrement, placez votre joueur à (25,25) dans le monde dans `Game.init()`.

Pour l'instant, votre joueur sera un simple et ennuyeux rectangle 1×2 : initialisez sa taille dans `Game.init()` et appelez `player.draw()` dans `Game.draw()`. Finalement, dessinez le joueur dans `Body.draw()`.

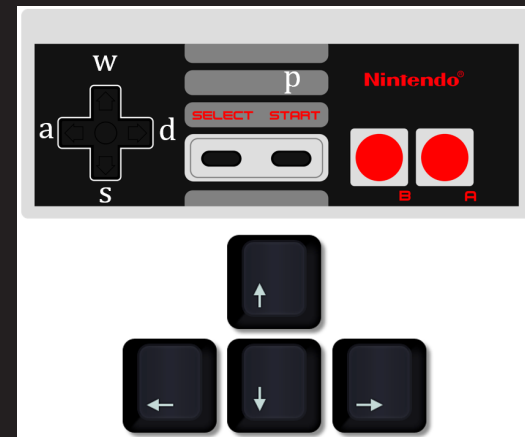
La position du joueur peut être modifiée avec l'attribut `player.pos`.

4. Il est temps de permettre au joueur de se déplacer avec les touches du clavier.

Modifiez `onKeyPress` dans `SuperMarioBros.pde` pour déplacer le joueur de 1 / 2 cellule dans les quatre directions, utilisant les touches W, A, S et D. Vous pouvez procéder en ajoutant un vecteur de déplacement à la position du joueur avec `game.player.pos.add(new Vec2(...))`.

Similairement, ajoutez du code pour bouger la fenêtre dans le monde en utilisant les touches fléchées (keycodes UP, DOWN, LEFT et RIGHT), encore une fois de 1 / 2 cellule. Ici, vous ajoutez un vecteur de déplacement en utilisant `game.window.translateBy(new Vec2(...))`.

Finalement, utilisez P pour activer/désactiver `game.play` et court-circuiter la fonction `Game.step()` lorsque `game.play == false`.

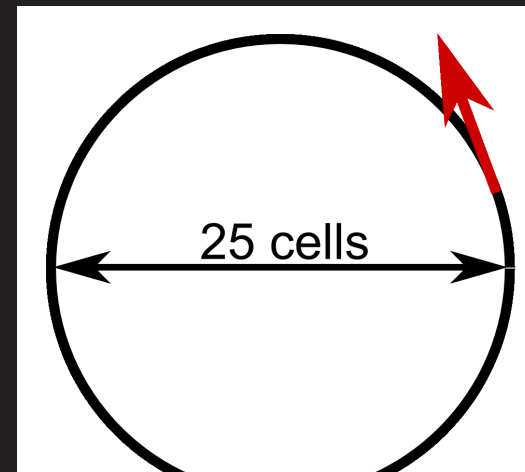


5. Pour fins de débogage futur, il sera utile de déplacer le joueur automatiquement, disons en un cercle.

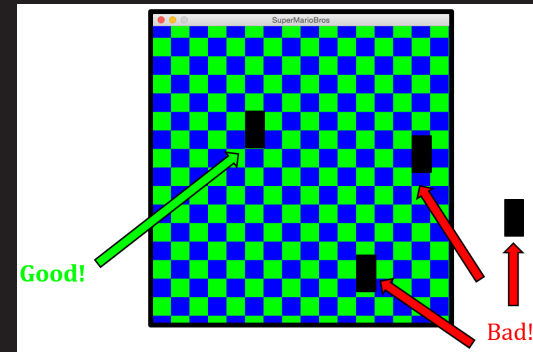
Dans `Player.step()`, utilisez `pos.add(new Vec2(vx,vy))` pour déplacer le joueur le long de la trajectoire d'un cercle, où (vx,vy) est un petit vecteur tangent au cercle qui dépend de `game.time`.

Jouez avec la taille et la vitesse de changement de ce vecteur pour déplacer le joueur sur un cercle d'approximativement 25 cellules de diamètre.

Les fonctions `sin` et `cos` vous seront utiles ici.



-
6. Assurez-vous que votre Mario rectangulaire ne sorte pas de la fenêtre : plutôt que de déplacer manuellement la fenêtre pour suivre le joueur, déplacez-la automatiquement pour conserver une distance d'au moins trois cellules entre le joueur et chaque bordure de la fenêtre.



-
7. Prenez une grande inspiration.

Sauvegardez votre progression et faites une copie de tout le dossier contenant le code en utilisant un schéma d'horodatage.

Par exemple, si vous avez terminé cet objectif à 10h30 le 3 février, utilisez un nom comme
`/backups/2019.02.03_10.30/SuperMarioBros/`



Objectif 2: Collisions

La capacité du joueur d'entrer en collision avec des obstacles formera la base de toutes les interactions du jeu.

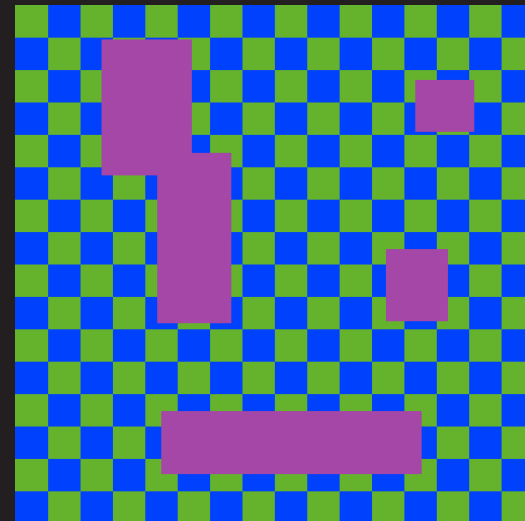
1. Ajoutez les cinq contrôles suivant :

- Utilisez `onMouseMove(...)` dans `SuperMarioBrox.pde` pour déplacer le joueur avec le curseur
- Utilisez `M` pour activer/désactiver le contrôle du joueur par la souris
- Utilisez `T` pour activer/désactiver le déplacement circulaire automatique du joueur
- Utilisez `L` pour activer/désactiver le déplacement automatique de la fenêtre
- Utilisez `1` pour réinitialiser le jeu (en appelant `game.init()`)

2. Avant d'implémenter les collisions, placez quelques objets "tests" que vous utiliserez pour entrer en collision avec le joueur : ajoutez quelques objets `Body` à `game.obstacles` en utilisant `obstacles.add(new Body(...))` dans `Game.init()`.

Donnez un peu de variété à vos objets : générez des positions et tailles aléatoires pour vos obstacles en utilisant `random(a, b)`, qui retourne un `float` aléatoire entre `a` et `b`. La taille minimale d'un objet doit être maintenue à 1 par dimension.

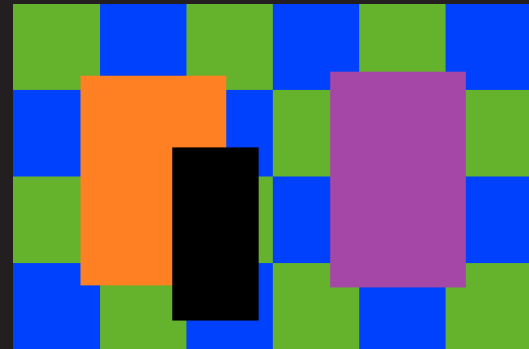
Donnez une couleur mauve aux obstacles et dessinez-les en utilisant une boucle dans `Game.draw()`, comme ceci :
`for(Body o : obstacles) { o.draw(); }`



Nous avons séparé le processus de collision en trois problèmes simplifiés:

3. Implémentez `Body.intersects(...)`, qui retourne un boolean indiquant qu'il y a, ou non, une intersection entre deux `Body`.

Dessinez les objets qui intersectent le joueur d'une autre couleur.

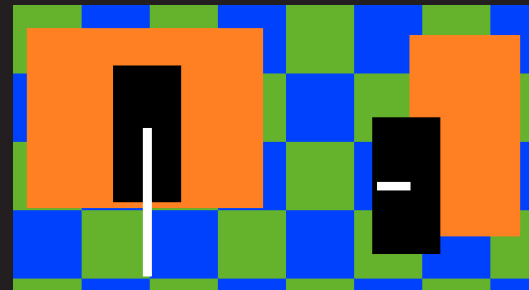


4. Implémentez `Body.computePushOut(Body arg)`, qui retourne un vecteur correspondant au plus petit déplacement nécessaire pour résoudre la collision entre `arg` et `this`.

Si un obstacle intersecte le joueur, dessinez le vecteur `pushOut` à l'écran, partant du centre du joueur, en utilisant `line(x1, y1, x2, y2)`.

Vous pouvez utiliser `stroke(255, 255, 255)` pour dessiner la ligne en blanc).

Attention à l'ordre dans lequel vous dessinez les différents éléments.



5. Associez la touche `0` à une nouvelle valeur boolean `solveObstacles`, dans `Game` : lorsque cette valeur est `true`, effectuez une translation du joueur pour résoudre la collision; lorsqu'elle est `false`, visualisez simplement ce vecteur comme à l'étape 4 précédente. Attention à l'ordre dans lequel vous dessinez le joueur et les obstacles.

Pour vous simplifier la tâche, nous avons créé plusieurs fonctions `handle<X>` et `interactWith<X>` dans `Body`.

Placez votre code qui repousse le joueur des obstacles dans `player.interactWith(Body)`. Passez en revue la structure de ces fonctions `handle<X>` et `interactWith<X>` pour bien comprendre comment votre code de déplacement est appelé à chaque obstacle. Cette structure sera importante plus tard lorsqu'il y aura différentes sortes de collisions dans le jeu.

6. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.



Objectif 3 : Charger le niveau

Donnons du piment à notre monde et remplaçons l'ennuyeux damier par des cellules du World 1-1 de Super Mario Bros.

Le niveau est composé d'une grille de cellules, comme à l'habitude, mais chaque cellule peut désormais contenir une combinaison d'**au plus trois éléments** (voir étapes 3 à 5, ci-dessous) : une image d'arrière-plan, une tuile (une brique, par exemple) et un item statique (une cent, par exemple).

Toutes les données dont vous aurez besoin pour le niveau sont dans un répertoire avec la structure suivante :

```
levels/  
lvl1-1/  
  lvl.txt // informations du niveau, ainsi que couleur d'arrière-plan  
  map.txt // description par caractère des tuiles du niveau  
  cellProperties.txt // explique la signification de chaque caractère dans map.txt  
  triggers.txt //emplacement des ennemis et autres objets dynamiques (=> Objectif 6)
```

Nous vous fournissons le code pour lire et convertir `lvl.txt` dans `Level.load()`, vous n'avez qu'à le décommenter : facile!

Ce code lit trois autres fichiers et leurs informations sont utilisées pour remplir les tableaux `backgroundImages`, `tiles` et `items`. Prenez le temps de bien comprendre la structure des données.

1. Complétez l'implémentation de `Level.loadMap(...)` pour remplir `map` adéquatement. Vous pouvez accéder au j^{e} caractère de la i^{e} ligne en utilisant `lines[i].charAt(j)`.

Attention! Assurez-vous de charger le niveau dans le bon ordre!

2. Complétez `loadTileProperties` pour décoder les propriétés de chaque symbole de la grille. Le code tokenise déjà `cellProperties.txt` pour que vous puissiez accéder aux attributs de ligne en utilisant `properties.get("<nom de l'attribut>")`.

Lorsqu'une ligne d'un seul caractère est lue, nous créons un nouveau `CellContent` et le remplissons avec les données des lignes suivantes. L'objet `CellContent` est finalement ajouté à la `tileProperties`.

Votre travail est d'interpréter les arguments d'une ligne et d'assigner la bonne information au `CellContent` courant :

Pour l'arrière-plan, chargez l'image dans l'attribut `image` avec
`currentCellContent.background = resources.getImage(<nom du fichier image>);`

Pour les tuiles, chargez uniquement les `SolidTile` pour le moment. Cela requiert de créer une nouvelle

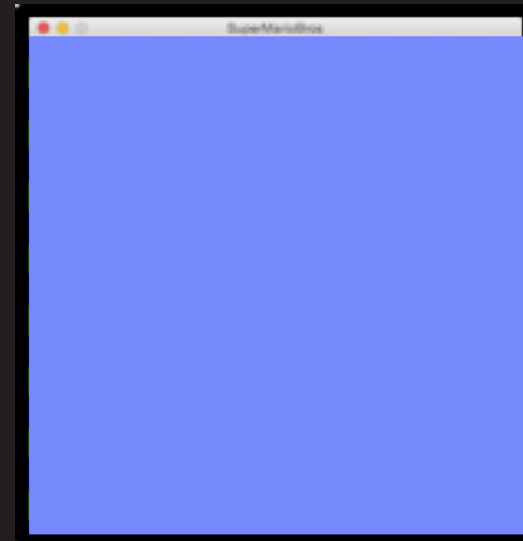
`SolidTile` et de lui assigner la bonne image. Nous nous occuperons des autres types de tuiles plus tard à l'Objectif 6.

Nous avons séparé l'affichage en quatre étapes, basées sur le contenu des cellules :

3. Implémentez et appelez

`level.drawBackgroundImages()` dans `Game.draw()` pour remplacer le damier par les tuiles du niveau :

Commencez par dessiner un rectangle (de couleur `level.backgroundColor`, valeur chargée depuis `Level.load(...)`) qui couvre l'entièreté du niveau **plus** une bande additionnelle de hauteur = 2 en haut de la fenêtre, qui servira plus tard à afficher des informations du jeu.



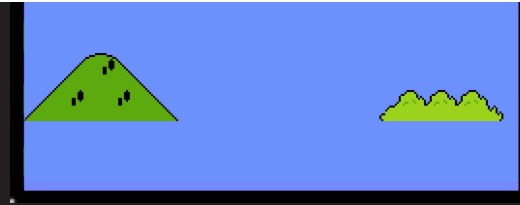
4. Ensuite, toujours dans

`level.drawBackgroundImages()`, bouclez sur toutes les cellules et dessinez son image d'arrière-plan (s'il y en a une) à l'emplacement approprié avec `drawer.draw(<image>, x, y)`.

Note : bien qu'il ne soit pas visible dans l'image ci-contre, le rectangle du joueur devrait être encore visible... en fait, vous pouvez même commencer à vous *déplacer dans le*



monde en utilisant les fonctionnalités du clavier et de la souris.



5. Implémentez et appelez `level.drawTiles()` dans `Game.draw()` : bouclera sur toutes les cellules, dessinant sa `Tile` (si elle en a une).

6. Dans `Body.draw()`, implémentez deux types d’affichage : si `Body.img` existe, dessinez cette image, sinon dessinez le `Body` en rectangle, comme avant.

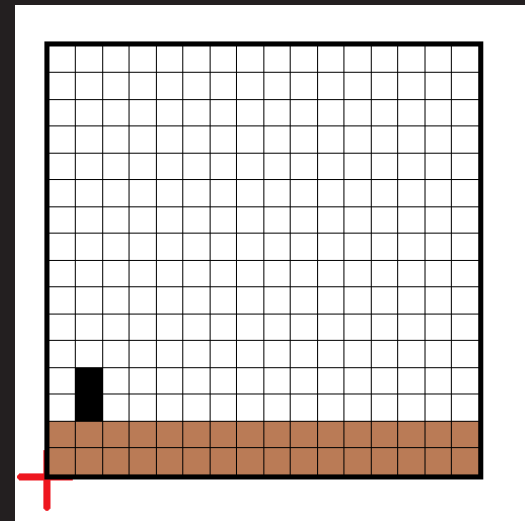


6. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.



Objectif 4 : Gestion avancée du niveau

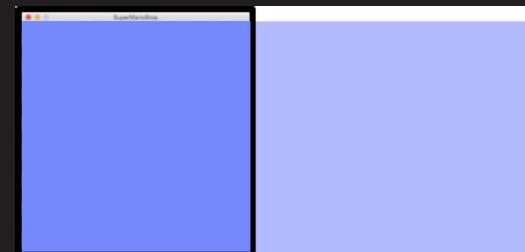
1. Positionnez le coin inférieur gauche de la fenêtre à l'origine et le joueur au niveau du sol.
2. Avant de vous fatiguer à mitrailler les touches `WASD` pour déplacer le joueur, associez les touches `1, 2, ..., 0` à la réinitialisation de la partie; ces touches transporteront le joueur à diverses positions également espacées horizontalement à travers le niveau.



3. Restreignez la taille de la fenêtre afin qu'elle n'affiche rien se situant à l'extérieur du niveau.

Restreignez le joueur afin qu'il demeure horizontalement dans le niveau (il peut encore le quitter par en haut et par en bas).

Dans le jeu officiel, la fenêtre ne peut se déplacer vers la

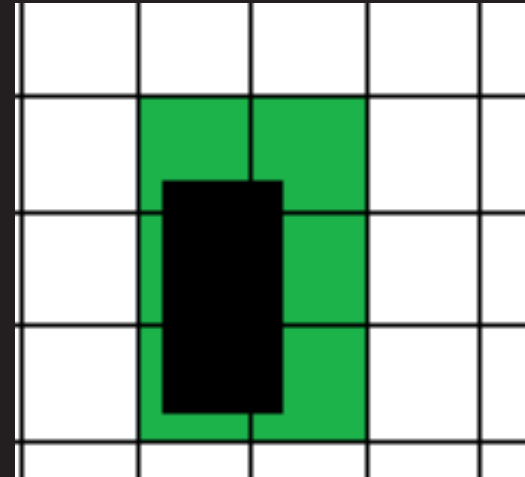


gauche, mais nous vous suggérons d'ignorer cette contrainte pour le moment et de la mettre en place plus tard si vous voulez recouvrer l'effet du jeu original.

-
4. Dans `Player.interactWith(Tile)`, réutilisez le code de collision de `player.interactWith(Body)` pour résoudre les collisions entre le joueur et les `Tiles`.

5. Effectuer le test de collision avec chacune des tuiles du niveau est très inefficace, sachant que les tuiles très éloignées du joueur n'ont aucune chance d'entrer en collision avec celui-ci.

Dans `Body.handleTiles()`, prenez avantage du fait que les tuiles sont positionnées sur une grille afin d'effectuer le test de collision uniquement avec les tuiles proches : `floor`, `ceil`, `min` et `max` vous seront utiles.

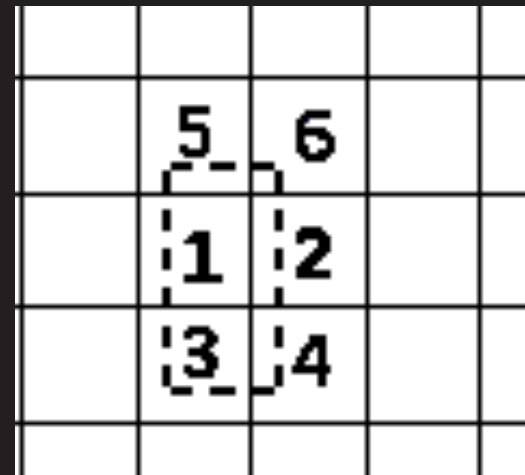


-
6. Vous remarquerez peut-être que votre joueur "s'accroche" à la grille en se déplaçant, spécialement lorsque vous activez son déplacement circulaire.

Il est **impératif** de régler ce problème pour s'éviter des problèmes plus tard.

Cela peut être solutionné en résolvant premièrement les collisions avec les tuiles les plus proches du joueur: ordonnez les tuiles dans `Body.handleTiles()` en ordre croissant de distance entre le centre du joueur et le centre de la tuile.

Utilisez un simple algorithme de tri pour ordonner ce



sous-tableau (*selection*, *bubble* ou *insertion sort*, par exemple).

-
7. Affichez le texte d'entête dans les deux rangées additionnelles en haut de la fenêtre: nous vous fournissons une fonction pour vous le permettre, `drawer.draw(String, x, y)`, où `String` est en lettres MAJUSCULES.

La fonction `nf(int, length)` est également utile : elle convertit un nombre en une chaîne de caractères d'une certaine longueur (concaténée de zéros). Un caractère spécial, `x` en lettre minuscule, peut être utilisé pour dessiner le symbole `x` à côté du nombre de cents.

- Ne dessinez pas la cent clignotante pour le moment.
- Enregistrez le compteur de cents dans `Game` et utilisez-le à l'affichage. Nous discuterons de l'incréméntation de ce compteur plus tard.
- Démarrez le chronomètre à 400 et décrémentez-le lentement en cours de jeu.
- Vous pouvez utiliser l'espace vide pour afficher toutes données de débogage qu'il vous plaira.



MARIO		WORLD	TIME
000123	x00	1-1	340

-
8. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.





Objectif 5 : Faites bouger Mario!

Il est temps de faire bouger Mario, en accord avec les lois de la physique:

$$\frac{dx(t)}{dt} = v(t) \qquad \frac{dv(t)}{dt} = a(t) - b v(t)$$

où x = position du joueur (vecteur 2D)
 v = vitesse du joueur (vecteur 2D)
 a = accélération (vecteur 2D)
 b = constante d'amortissement qui simule la friction

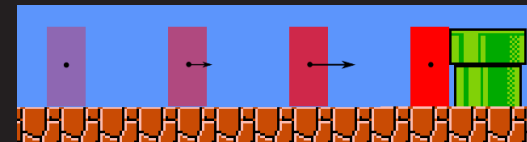
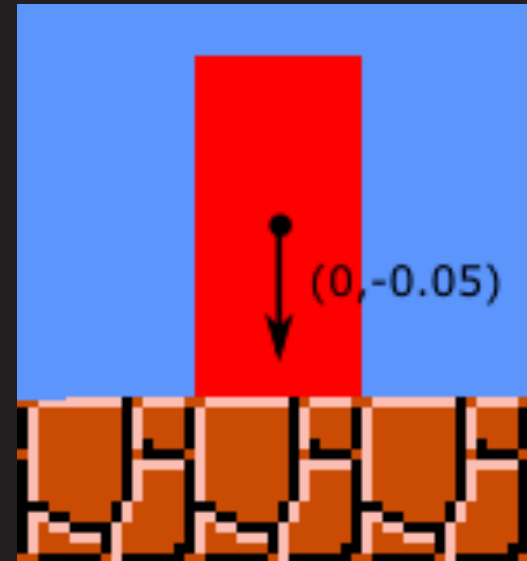
Ces équations peuvent être discrétisées et résolues avec l'algorithme suivant :

```
v += dt * a;  
v *= b * dt;  
x += dt * v;
```

où dt = pas de temps, soit le "temps de simulation"
qui passe entre les images (défini dans Game)

Note : cet algorithme n'est qu'une version un peu plus complexe des mouvements de l'Objectif 1, où vous mettiez à jour la position en lui ajoutant un petit vecteur à chaque pas de temps (en images).

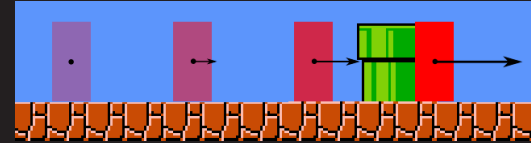
1. Dans `Game`, ajoutez un vecteur de `gravity` et initialisez-le pour qu'il pointe vers le bas avec une magnitude de `0.05`. Initialisez la constante d'amortissement à une valeur légèrement plus basse que `1` : nous vous suggérons d'utiliser différentes valeurs d'amortissements horizontal et vertical, que vous pouvez ajuster plus tard pour corriger le comportement du joueur.
2. Dans `Player.step()`, assignez à `player.accel` la valeur de gravité du jeu et mettez à jour `player.vel` et `.pos`.
3. Retirez les contrôles `WASD` de `onKeyPress`. Dans `Player.step`, utilisez `Keyboard.isPressed` et `Keyboard.isReleased` pour interroger l'état du clavier : de cette façon, la touche peut simplement être retenue pour déplacer le joueur, plutôt que d'être mitraillée.
4. Changez les touches `A` et `D` pour affecter l'accélération du joueur plutôt que de changer directement sa position. Il en résultera un mouvement beaucoup plus en douceur. Ajustez la constante d'amortissement afin que le joueur ralentisse adéquatement lorsque les touches sont relâchées.
5. Lorsque le joueur rencontre un obstacle, assignez `0` aux



composantes x ou y de sa vélocité, l'axe dépendant de la situation.

-
6. Les collisions peuvent parfois échouer lorsque le joueur rencontre un petit obstacle. Cela se produit si le déplacement du joueur est trop grand entre chaque pas de temps, puisque la mise-à-jour de la position peut positionner le joueur de l'autre côté de l'obstacle.

Afin de prévenir cela, limitez le déplacement du joueur à `0.49` tuiles par pas de temps. Vous pouvez utiliser la fonction `clamp` définie dans `Utils.pde`.



-
7. Implémentez un saut réaliste en augmentant `vel.y` du joueur et en laissant la physique s'occuper du reste. Souvenez-vous que le joueur ne peut sauter une fois qu'il est dans les airs, mais seulement lorsqu'il touche au "sol".

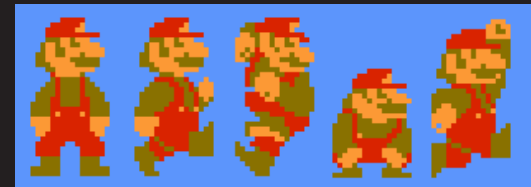
Faites également en sorte qu'il saute plus haut lorsque la touche est retenue plus longtemps.

8. Faites en sorte que le joueur se "penche" lorsque la touche `S` est appuyée. Lorsque penché, changez sa taille à (1, 1) plutôt que (1, 2).

-
9. **Le moment est venu de pimenter un peu les choses :** remplacez le rectangle par un **vrai Mario** !

Dans `Player.draw`, chargez le `ImageSet` de Mario avec `imgSet = bigMarioSet`.

Note : Cet `ImageSet` regroupe les images de `data/img/players/`, et vous pourrez plus tard changer l'apparence



de Mario simplement en changeant ce `ImageSet`.

Utilisez l'image appropriée dans le `ImageSet`, dépendant de l'état de Mario (en course, mort, etc.): par exemple, lorsque mort, assignez `img = imgSet.get("dead")`.

Les images peuvent être animées (comme `run` dans `bigMarioSet`) ; ajustez la vitesse d'animation avec `img.setSpeed` afin que l'animation soit relative à la vitesse de Mario.

Dessinez les images avec `drawer.draw` (définie dans `Utils.pde`), en utilisant la fonction qui vous permet de retourner et d'orienter les images selon la direction dans laquelle Mario fait face.

-
10. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.



Objectif 6 : Ajouter des items et des ennemis

Il est temps de faire bouger les `Enemy` et `Item`, et de les faire interagir avec le monde (mais pas avec Mario, pas encore... patience !)

1. Certaines classes `Enemy` et `Item` ont été créées pour vous. Commencez par dessiner de faux (générés par vous) `Goombas`, `Koopas`, `Mushrooms`, `Flowers`, `OneUps` et `Stars` dans le niveau, depuis `Game.draw()`.

2. Chargez les images pour les `Mushrooms`, `Flowers`, `OneUps` et `Stars`.

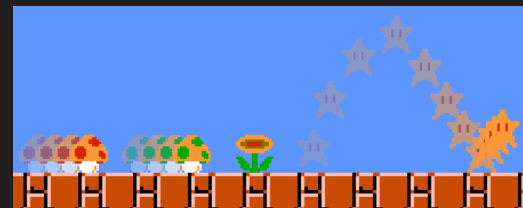
Notez que les `Flowers` et les `Stars` ont des images animées et peuvent être chargées en utilisant le caractère générique `%d` : notre système de chargement d'images va prendre en charge toutes les images correspondant à ce caractère, créant ainsi une image animée.

Par exemple, vous pouvez charger l'image animée de la `Flower` en utilisant

```
img = resources.getImage("data/img/items/flower/%d.png")
```

3. Implémentez les mouvements correspondant à chaque `Item`. Cette étape sera similaire à celle où vous avez implémenté les mouvements du joueur, et vous pouvez (prudemment) réutiliser du code si vous structurez tous les mouvements communs dans `Body.step()` et appelez par la suite `super.step()` de la méthode `step`.

Portez attention aux variantes des mouvements : les `Goombas` et `Koopas` font demi-tour face aux murs et ont



une vitesse constante, alors que les `Stars` rebondissent constamment, etc.

-
4. Pour les `Goombas` et `Koopas`, associez l'`ImageSet` approprié, comme cela a été fait avec le joueur. Regardez dans `data/img/enemies/`.

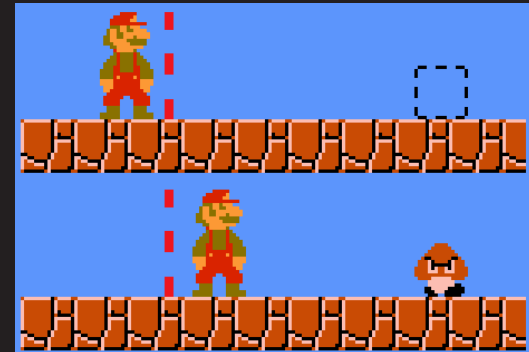
Dessinez les images basées sur le mouvement des ennemis (encore une fois très similaire à ce qui a été fait pour le joueur).



-
5. Maintenant que les items et ennemis sont “en vie”, placez-les vrais ennemis là où ils devraient être (les vrais items seront chargés plus tard).

Ce sont des `Triggers` qui font apparaître les ennemis dans le monde. Il s'agit d'évènements qui ont lieu après une action précise. Dans le cas des ennemis, leur apparition est déclenchée selon la position de Mario.

- Implémentez `Level.loadTriggers` : le code lit et tokenize déjà chaque ligne du fichier. Pour chaque ligne associée à un ennemi (donc excluant le drapeau final, qui sera utilisé plus tard), créez un `EnemyTrigger` qui place le bon ennemi à la bonne position, tout en lui assignant le `ImageSet` approprié et en ajoutant l'ennemi au `EnemyTrigger`. Finalement, ajoutez ce `EnemyTrigger` à la liste des `Triggers`, `Level.triggers`.
- Implémentez la classe `EnemyTrigger` : lorsque Mario s'approche de la position de départ de l'ennemi, ce dernier devrait être ajouté à `Game.enemies`. Jetez un



œil aux trois fonctions dans la classe `Trigger` et prenez le temps de comprendre comment elles sont utilisées dans `Game`. Cela vous donnera une idée de la façon dont ces fonctions devraient être surchargées dans `EnemyTrigger`.

6. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.



Objectif 7 : Que Mario fracasse des trucs !

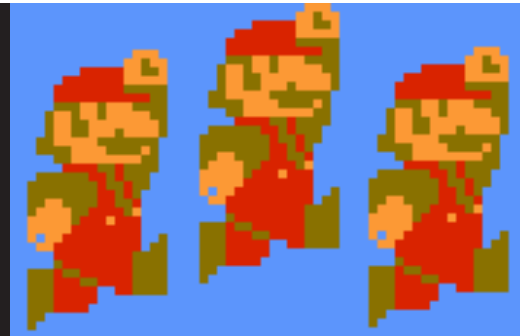
Commencez à implémenter les types de tuiles restantes et quelques interactions plus complexes. Ne vous préoccupez pas des animations pour le moment (bientôt, bientôt...)

1. Dans `Level.loadTileProperties`, écrivez le code pour charger des `BreakableTiles`. Ces tuiles utilisent un `ImageSet` plutôt qu'une image seule. Elles conservent deux images : une pour avant sa destruction et une autre



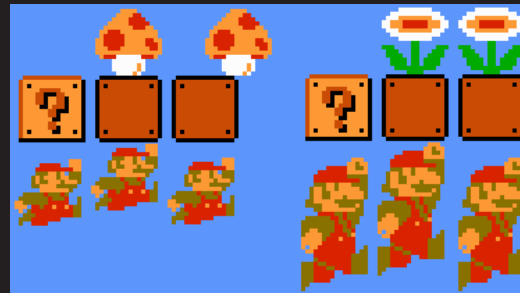
pour après.

Utilisez le `ImageSet` listé dans `cellProperties.txt` et affichez l'image appropriée selon l'état de la tuile.



-
2. Chargez également des `ContainerTiles`. Celles-ci contiennent `n` `Items` (ex.: un `ContainerTile` peut contenir 10 `Coins`, par exemple), et utilisent également un `ImageSet` pour afficher l'image appropriée selon si la tuile contient encore des `Items` ou non.

Notez les `Items` spéciaux `Grow` : ceux-ci peuvent être soit un `Mushroom` ou une `Flower`, selon l'état dans lequel est Mario lorsqu'il frappe la `ContainerTile`.



-
3. Implémentez les interactions joueur-ennemis:

- Lorsque Mario frappe un ennemi **d'en haut**, il le tue (faites disparaître ce dernier pour le moment).
- Les ennemis qui frappent Mario **autrement** changent son état, son `ImageSet` et parfois même sa taille : il peut être petit, grand ou en "mode fleur".
- Lorsque Mario est frappé, il devient invincible et son image *clignote* pour un petit moment.

-
4. Implémentez `Player.interactWith(Tile)` :

- une `BreakableTile` est détruite lorsque frappée par en dessous (sauf si Mario est petit)
- Une `ContainerTile` relâche son contenu lorsque frappée par en dessous.

Relâchez l'Item approprié pour les tuiles avec un Item de type Grow.

5. Faites en sorte que les ennemis se repoussent entre eux.

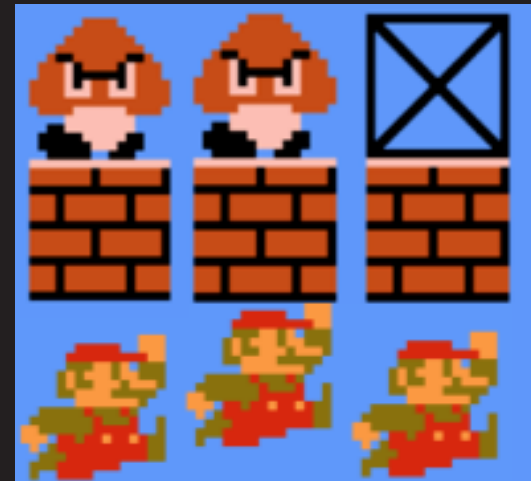
6. Les Koopas ne meurent pas directement lorsque Mario leur saute dessus : ils se transforment en une coquille qui peut être ensuite propulsée par le joueur, tuant d'autres ennemis sur son passage.

Implémentez ce comportement un peu plus complexe.

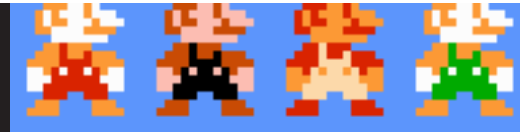


7. Si un ennemi se trouve sur une tuile lorsque Mario frappe cette dernière par dessous, l'ennemi meurt.

Implémentez ce comportement.



Implémentez les interactions avec les `Mushrooms` et les `Flowers`, qui changent l'état de Mario, ainsi que les `Stars` qui le rendent temporairement invincible.



Finalement, les `OneUps` doivent lui ajouter une vie et les `Coins` doivent affecter le compteur de cents.

Pour créer l'effet de clignotement lorsque Mario est invincible, utilisez les images de `smallStarMarioSet` ou `bigStarMarioSet` pour son `ImageSet`.

-
9. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.



Objectif 8 : Effets spéciaux

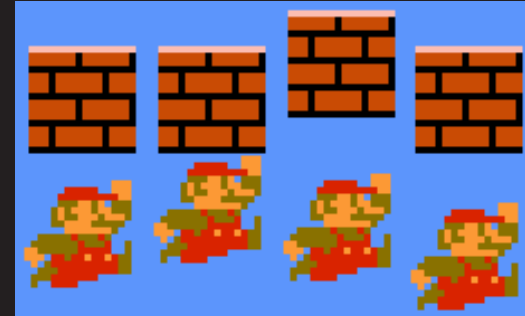
Pimentez le jeu en lui ajoutant des `Animations`. Ces images apparaîtront à l'écran mais n'auront aucun effet sur les interactions physiques : elles sont donc plus faciles à coder, mais ajoutent un plus significatif à la "valeur de production".

Le code pour évoluer à travers une `Animation` et la supprimer vous est fourni dans `Game`, mais vous devrez cette fois écrire vous-mêmes le code pour la dessiner, encore dans `Game`.

Pour chaque effet présenté ci-dessous, vous allez devoir créer une classe qui implémente l'interface `Animation`.

1. Lorsque Mario frappe une `ContainerTile` ou une `BreakableTile` par dessous, rendez la tuile invisible et remplacez-la par une séquence d'`Animation`.

Une fois l'`Animation` terminée, remettez la tuile visible et assurez-vous que la fonction `completed()` retourne `true` afin que cet objet `Animation` soit supprimé par `Game`.



2. Animez les morceaux de brique lorsqu'une `BreakableTile` est détruite. Les images d'animations pour ces morceaux vous sont fournies dans l'`ImageSet` d'une `BreakableTile`.



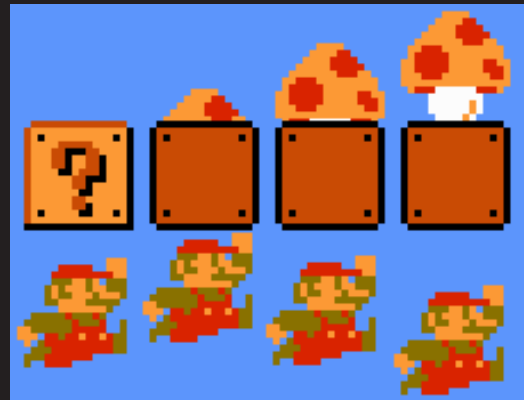
3. Implémentez une `Animation` pour les `Goombas` lorsque



Mario les écrase.



-
4. Faites en sorte que les objets émergent lentement des `ContainerTiles` lorsqu'ils sont distribués.



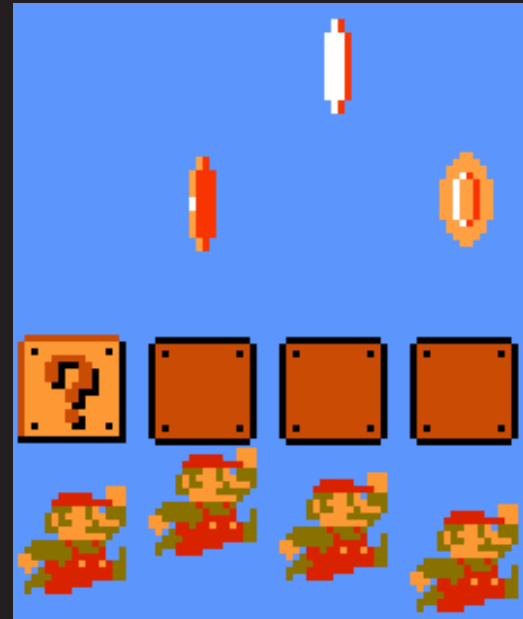
-
5. Lorsque les ennemis sont tués indirectement par une tuile, ou lorsque Mario collecte une `Star` et touche un ennemi, une mort animée se produit.

Implémentez cette `Animation`. Chaque `ImageSet` d'ennemis contient une image pour l'état de mort.





6. Animez les cents sortantes des `ContainerTiles`.



7. Animez la cent clignotante dans l'entête de jeu. `data/img/misc/hudCoin`.



8. Sauvegardez votre progression, synchronisez le code avec votre équipe et faites un backup horodaté.



FÉLICITATIONS, VOUS AVEZ COMPLÉTÉ LE HACKATHON!

Mais attendez, d'autres défis vous attendent plus bas... si vous osez...



Bonus : Un *Kooparti*, épatez-nous!

Félicitations ! Pour vous être rendu si loin, considérez-vous comme un Kamek du Mario-codage.

Mais pourquoi s'arrêter ici ? À vous de prendre le contrôle... plus d'instructions, on vous lâche la main ! Laissez aller votre imagination, surpassez-vous, créez un monde complètement fou ! Surprenez-nous !

Vous êtes maintenant libre d'implémenter plusieurs nouvelles fonctionnalités, mais attention à ne pas avoir les yeux plus grands que la panse : planifiez soigneusement vos ajouts. Sentez-vous libres de créer de nouvelles images et de prendre avantage de la structure du code et des fonctionnalités de chargement que vous connaissez maintenant bien.

Assurez-vous tout de même de continuer à créer des sauvegardes à chaque ajout significatif.

Voici quelques suggestions de fonctionnalités supplémentaires que, avoir été dans vos souliers, nous aurions implémentées :

-
1. N'est-il pas frustrant d'atteindre la fin du niveau 1-1 sans pouvoir descendre le drapeau... peut-être pouvez-vous arranger cela ?



-
2. Pourquoi vous limiter au niveau 1-1 ? Vous avez déjà tout ce qui vous faut pour créer les prochains niveaux !

Fouillez un peu dans les images fournies dans `data/img/`,



vous pourriez peut-être trouver des choses utiles...



-
3. Rien ne vous oblige à vous limiter au premier jeu de Mario. Pourquoi ne pas introduire des visages familiers à Mario?



-
4. Aucun jeu n'est complet sans une épique bataille de boss... qu'en feriez-vous?



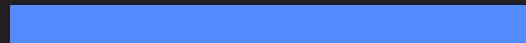
5. Le pauvre Mario en mode fleur ne peut toujours pas lancer de boules de feu à ses ennemis...



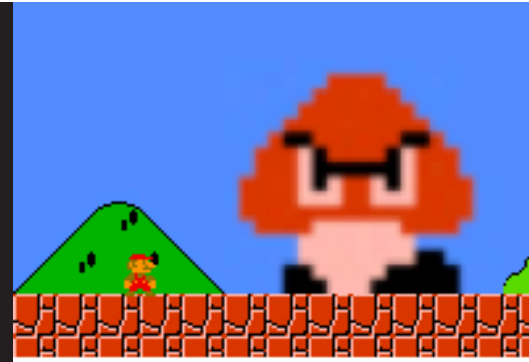
6. Créez une version ridiculement difficile du Niveau 1-1, vous pourrez défier vos amis à le réussir!



7. Rendez Mario géant. Ou, encore mieux, créez des ennemis



géants! Comment les tueriez-vous?



8. Inviteriez-vous Luigi à la fête?



9. Un item qui permet à Mario de lancer des lasers !

10. Une tuile spéciale qui inverse la gravité...

11. Un jetpack à boules de feu et qui permet de voler !

12. Que se passe-t-il quand le chrono atteint zéro ?

13. Tout ça manque de *Warp Zone*...
