

Télécom SudParis
Année scolaire 2020/2021

PRO3600

Gravitime

HUOT Martial
NOMICO Henri
LAFERRÈRE Alexandre
CHAUVEL Thomas

Enseignant responsable:
BLANC Gregory

Livrable 3



21/05/2021

Table des matières:

| | |
|---|-----------|
| 1/ Analyse des besoins | 3 |
| 2/ Spécification fonctionnelle générale | 4 |
| 3/ Conception préliminaire | 5 |
| 3.1- Regroupement modulaire des fonctionnalités | 5 |
| 3.2 - Description du flux des données entre les modules | 6 |
| 4/ Conception détaillée | 7 |
| 3.1-Tests unitaires | 16 |
| 3.2-Tests d'intégration | 16 |
| 3.3-Tests de validation | 17 |
| 6/ Manuel utilisateur | 25 |
| 6.1-Manuel utilisateur pour jouer au jeu | 25 |
| 6.2-Manuel utilisateur pour les tests | 25 |
| 6.3-Manuel d'utilisation du code | 26 |
| 7/ Bilan | 27 |
| 8/ Bibliographie | 28 |

1/ Analyse des besoins

Gravitime sera un court jeu de plateforme-puzzle en 2D où le joueur aura accès à des mécaniques de manipulation du temps et de la gravité pour progresser dans le jeu.

- ❖ Le joueur devra pouvoir se mouvoir à l'aide des flèches directionnelles du clavier dans un environnement 2D avec vue de côté.
- ❖ La caméra sera centrée sur le joueur, et une commande ou un élément du jeu permettra d'avoir une vision d'ensemble de la salle.
- ❖ Le joueur doit pouvoir sauter et interagir avec des éléments tels que des boutons, des caisses, des portes et les ennemis.

Le joueur doit pouvoir **changer la direction et l'intensité de la gravité** dans des zones spécifiques prévues à cet effet :

- ❖ Le vecteur champ de pesanteur pourra être modifié à l'aide de la souris (click n' drag) dans les zones spécifiques.
- ❖ Un indicateur visuel permettra au joueur de connaître l'état actuel du vecteur champ de pesanteur.
- ❖ Cela permettra au joueur de se propulser ou de propulser des objets dans une certaine direction pour atteindre la fin du niveau.
- ❖ De plus, en fonction de l'intensité du champ, certains objets/ennemis seront plus affectés que d'autres, notamment ceux de masse importante.

Concernant **la mécanique de contrôle du temps**, il y aura 2 trames temporelles.

- ❖ Le jeu doit pouvoir enregistrer les actions du joueur pour pouvoir générer un clone temporel qui reproduira les actions effectuées par le joueur au cours de la première trame temporelle.
- ❖ Le joueur pourra ensuite interagir avec son clone lors de la seconde trame temporelle. En particulier le clone conservera sa hitbox, permettant au joueur de s'en servir de plateforme auxiliaire dans la seconde trame.
- ❖ Le clone pourra aussi pousser des objets ou appuyer sur des boutons à la place du joueur.

Les **ennemis** se comportent de plusieurs manières différentes et auront **diverses manières d'interagir avec le temps et la gravité**.

- ❖ Certains redirigeront la gravité dans une zone autour d'eux, d'autres seront insensibles aux modifications gravitationnelles.
- ❖ Certains viseront le clone temporel plutôt que le joueur lui-même. Par exemple, si un ennemi cherchait à frapper le joueur de la trame originelle, il sera peut-être amené à frapper le joueur de la deuxième trame temporelle. Un indicateur visuel sur les ennemis permettrait d'indiquer dans quelle trame celui-ci interagit.

Nous serons peut-être amenés à ajouter plus de comportements au fil du développement.

- ❖ Chaque action du jeu entraîne un signal sonore, comme par exemple un bruit d'attaque d'ennemi ou du joueur, l'activation des pouvoirs spatio-temporels du joueur ou des ennemis.

Le jeu se termine sur un **boss final** manipulant gravité et temps.

2/ Spécification fonctionnelle générale

Sauter : Le joueur doit **sauter**.

Avancer : Le joueur doit pouvoir **avancer** selon l'axe horizontal.

Attaquer (joueur) : Le joueur doit pouvoir **attaquer** un ennemi.

Prendre des dégâts (joueur) : Le joueur doit **prendre** des dégâts d'un ennemi.

Créer un clone : Le joueur doit pouvoir **créer** un clone temporel.

Modifier gravité (joueur) : Le joueur doit pouvoir **changer** la gravité (angle et intensité) dans des zones prédéfinies.

Affichage (joueur) : le joueur doit pouvoir **afficher** différentes animations en fonction de ses actions.

Bruitage (joueur) : le joueur doit pouvoir **jouer** différents **sons** en fonction de ses actions.

Interactions physiques (caisse) : Les caisses doivent pouvoir **interagir** physiquement avec les entités (joueurs, ennemis)

Interactions physiques (bouton) : Les boutons doivent pouvoir **interagir** physiquement avec les entités (joueurs, ennemis, caisse)

Ouverture : Les portes doivent pouvoir **s'ouvrir** si des conditions sont remplies.

Interactions physiques (portes) : Les portes doivent pouvoir **interagir** physiquement avec les entités (joueurs, ennemis, caisse)

Interactions physiques (pièges) : Les pièges doivent pouvoir **interagir** physiquement avec les entités (joueurs, ennemis, caisse)

Prendre des dégâts (ennemi) : Les ennemis doivent pouvoir **prendre** des dégâts du joueur.

Attaquer (ennemi) : Les ennemis doivent pouvoir **attaquer** le joueur.

Se déplacer : Les ennemis doivent pouvoir se **déplacer** dans le monde.

Modifier gravité (ennemi) : Certains ennemis doivent pouvoir **modifier** des champs de gravité.

Prioriser : Certains ennemis doivent pouvoir **prioriser** entre le joueur et le clone temporel.

Affichage (ennemi) : Les ennemis doivent pouvoir **afficher** différentes animations en fonctions de leurs actions

Bruitage (ennemi) : Les ennemis doivent pouvoir **jouer** différents **sons** en fonction de leurs actions.

Affichage HUD : L'interface doit **afficher** les barres de vie (joueur, ennemis).

Démarrer le jeu : L'utilisateur doit pouvoir **lancer** le jeu (bouton start)

Selectionner Niveau : L'utilisateur doit pouvoir **choisir** les niveaux.

Mettre en pause : L'utilisateur doit pouvoir **mettre** en pause.

Suivre le joueur : La caméra doit **suivre** le joueur.

Perdre : Le jeu doit **afficher** un game over si des conditions sont remplies

Jouer la musique : Le jeu doit **jouer** une musique en fond.

Enregistrer les actions : La Trame Temporelle doit **enregistrer les actions** du joueur.

Remettre à zéro : La Trame Temporelle doit pouvoir **remettre** à zéro les entités (ennemis, joueur, clone, caisses, boutons, porte, pièges)

Appliquer gravité : Les champs de gravité **appliquent** une gravité sur leur zone définie

3/ Conception préliminaire

3.1- Regroupement modulaire des fonctionnalités

Nous regroupons les fonctionnalités sous les modules suivants:

- ❖ Module Joueur :
 - Sauter
 - Avancer
 - Attaquer (joueur)
 - Prendre des dégâts (joueur)
 - Créer un clone
 - Modifier gravité (joueur)
 - Affichage (joueur)
 - Bruitage (joueur)
- ❖ Module Ennemis
 - Prendre des dégâts (ennemi)
 - Attaquer (ennemi)
 - Se déplacer
 - Modifier gravité (ennemi)
 - Prioriser
 - Affichage (ennemi)
 - Bruitage (ennemi)
- ❖ Module Trame Temporelle
 - Enregistrer les actions
 - Remettre à zéro
- ❖ Module Champs de Gravité
 - Appliquer gravité
- ❖ Module Caisse
 - Interactions physiques (caisse)
- ❖ Module Bouton
 - Interactions physiques (bouton)

- ❖ Module Porte
 - Ouverture
 - Interactions physiques (porte)
- ❖ Module HUD
 - Affichage HUD
- ❖ Module Pièges
 - Interactions physiques (pièges)
- ❖ Module Gestion du Jeu
 - Démarrer le jeu
 - Sélectionner Niveau
 - Mettre en pause
 - Suivre le joueur
 - Perdre
 - Jouer la musique

3.2 - Description du flux des données entre les modules

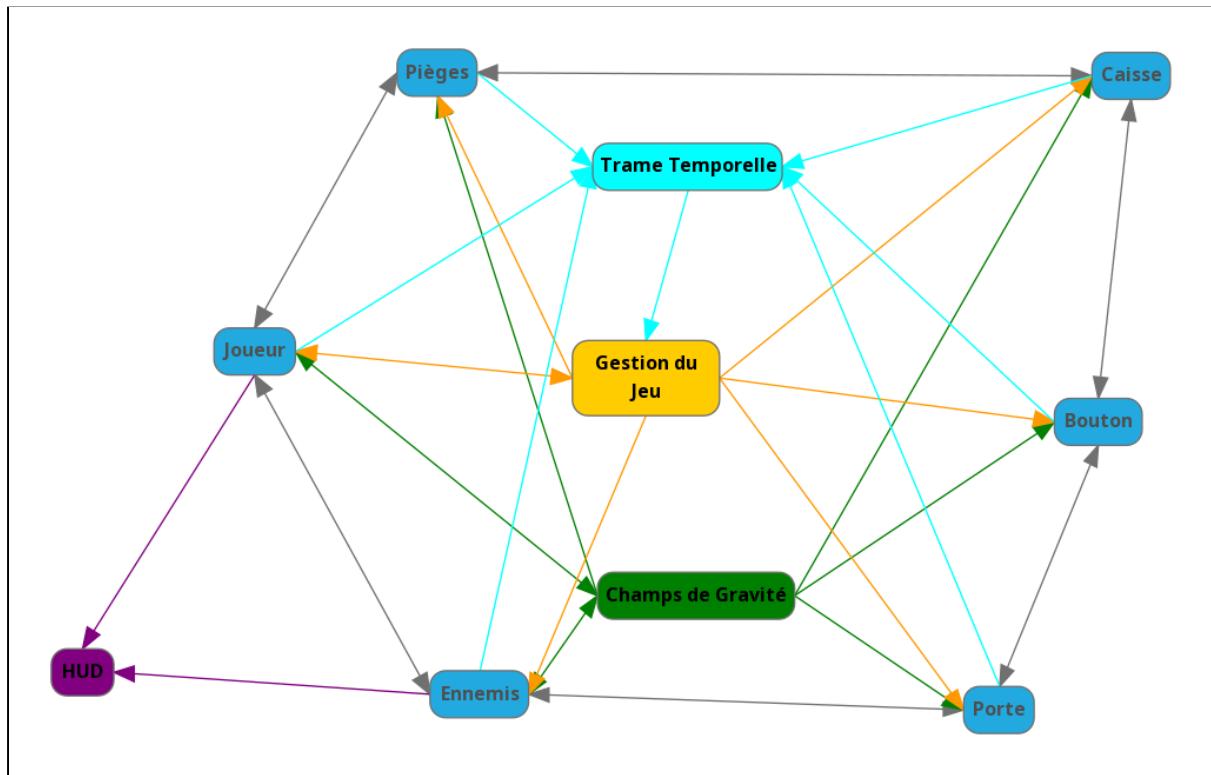


Figure 1 - Schéma des flux des données entre les modules

4/ Conception détaillée

Structure globale :

La structure de nos fichiers est :

Resources: contient toutes les ressources artistiques du jeu

Art : contient toutes les images utilisées par le jeu

Music : contient la musique du jeu

SFX : contient la police d'écriture utilisée par le jeu

SoundEffects : contient les bruits utilisés par les personnages

Source: contient tous les éléments source du jeu (codes et scènes) rangés dans des dossiers détaillés ci-dessous en "Description des scripts et fonctions"

Chaque dossier est ensuite séparé en "Scene" qui contiennent les scènes qui contiennent toutes les informations et paramètres utilisés par Godot et en "Scripts" qui contiennent les codes sources

Autoload

Scene

Script

Actors

Entities

Levels

UserInterface

World

Gravity

Time

Gravitime Executable : **contient le .exe du jeu**

addons: contient l'addon utilisé pour réaliser la plupart des tests rapidement - l'addon en question est Gut (et provient d'Internet, cf. biblio)

test: contient la scène utilisée pour visualiser les tests et les scripts des tests

Les fichiers **default_env.tres**, **project.godot** sont des fichiers contenant les différents paramètres globaux du jeu (comme les commandes utilisateur, la résolution de l'écran etc)

Les fichiers **icon.png** et **icon.png.import** représentent l'icône de l'application (pour le moment non touchée)

Description des scripts et fonctions :

Actors : Contient tous les acteurs du jeu, soit le Player, les ennemis et les clones temporels

_____ Actor.gd : Interface qui détermine des variables et méthodes utilisées par tous les acteurs

Champs :

speed : Vector2 : contient la vitesse limite (selon x et y) des acteurs

gravity : Vector2 : contient l'intensité de la gravité

is_in_gravity_field : Bool : indique si l'acteur est dans un champs de gravité

→ **apply_gravity(area: GravityField)**: fonction qui applique la gravité actuelle à l'acteur (joueur, clone, ennemi)

_____ Boss.gd: Boss final du jeu, qui n'est affecté ni par la gravité, ni par le contrôle du temps. Il tire régulièrement en visant le joueur, ou son clone temporel s'il existe.

Champs :

boss_hp : Int : contient vie du boss

→ **_ready()**: appelée à linstanciation de lennemi; ajoute linstance au groupe "enemy"

→ **boss_hit(dmg)**: appelée lorsque le boss est touché par une attaque critique (i.e. une boîte); décrémente les points de vie du boss

→ **on_ReloadTimer_timeout()**: appelée lorsque le timer séparant les tirs du boss atteint 0

_____ Enemy.gd : Ennemi principal du jeu, qui se contente de bouger en ligne droite jusqu'à ce qu'il rencontre un mur ou du vide auquel cas il rebrousse chemin

Champs :

timeposition: Vector2 : contient la position de Enemy avant le rembobinage temporel

→ **_ready()**: appelée à linstanciation de lennemi; ajoute linstance aux groupes "timecontrol" et "enemy", et lance une animation

→ **_physics_process(delta: float)**: appelée à chaque frame du jeu; attribue sa vitesse à lennemi

→ **calculate_new_velocity(_velocity)**: fonction appelée à chaque frame, oriente la sprite, calcule la vitesse à attribuer à lennemi et teste notamment si celui-ci rencontre un vide ou un mur

→ **hit(dmg)**: fonction appelée par la classe MeleeAttack à chaque fois que lennemi est touché par lattaque du joueur (MeleeAttack); détruit cette instance de lennemi

→ **save()**: fonction appelée par TimeControl, qui sauvegarde la position de lennemi lorsque la trame temporelle de rembobinage débute

→ **timeReset()**: fonction temporelle appelée par TimeControl à la fin de la sélection de la trame temporelle de rembobinage, qui change la position de lennemi à sa position au début de ladite trame (donnée par fonction save())

→ **_on_PhysicalHitbox_area_entered(area: Area2D)**: fonction appelée dès que lennemi rentre dans une zone. Teste notamment si lennemi est rentré un champ de gravité pour activer ses effets.

→ **_on_PhysicalHitbox_area_exited(area: Area2D)**: fonction appelée dès que lennemi sort dune zone. Teste notamment si lennemi est sorti dun champ de gravité pour désactiver ses effets.

[Player.gd](#) : Le personnage que le joueur peut contrôler à l'aide des touches du clavier.

Champs :

spawn_position : Array : contient la liste des inputs du joueur pour un rembobinage
facing : Bool : contient l'état d'orientation du joueur (droite ou gauche)
timecontrol_active : Bool : indique si il y a rembobinage de temps
timeposition: Vector2 : contient la position de Player avant le rembobinage temporel
is_attacking : Bool : indique si le joueur est en train d'attaquer ou non
is_on_pause : Bool : indique si le jeu est en pause ou non

→ **_ready()**: appelée à l'instanciation du joueur; ajoute l'instance aux groupes "timecontrol" et "player" et lance une animation
→ **_physics_process(delta: float)**: appelée à chaque frame du jeu; attribue sa vitesse au joueur
→ **get_direction()**: traduit les commandes du joueur en direction
→ **movements_animation(direction: Vector2)**: lance les animations correspondant aux mouvements du joueur
→ **calculate_move_velocity(**
 linear_velocity: Vector2,
 direction: Vector2,
 speed: Vector2,
 is_jump_interrupted: bool
): calcule la vitesse à attribuer au joueur
→ **_on_PhysicalHitbox_body_entered(body: Node)**: appelée lorsque le joueur rentre en contact avec un corps étranger et appelle la fonction hit() si ce corps est un ennemi
→ **hit(dmg)**: appelée par la fonction _on_PhysicalHitbox_body_entered(), est la fonction qui traduit le fait que le joueur est touché : il perd une vie et revient à sa position initiale ou appelle la fonction die() si il n'a plus de vie
→ **die()**: appelée par hit() si le joueur n'a plus de vie, détruit l'instance du joueur et charge l'écran de fin de partie
→ **skills()**: appelée à chaque frame par _physics_process(), qui permet de gérer les skills du joueur (contrôle temporel et attaque)
→ **attack()**: appelée par skills() et permet au joueur d'attaquer en créant une instance de MeleeAttack en face de Player
→ **save()**: appelée par TimeControl, sauvegarde la position actuelle du joueur
→ **timeReset()**: appelée par TimeControl, remet le joueur à sa position sauvegardée par save()
→ **_on_PhysicalHitbox_area_entered(area: Area2D)**: fonction appelée dès que le joueur rentre dans une zone. Teste notamment si le joueur est rentré un champ de gravité pour activer ses effets.
→ **_on_PhysicalHitbox_area_exited(area: Area2D)**: fonction appelée dès que le joueur sort d'une zone. Teste notamment si le joueur est sorti d'un champ de gravité pour désactiver ses effets.
→ **timelInputs()**: fonction appelée à chaque frame lorsque le joueur lance le rembobinage temporel; enregistre les actions du joueur afin que le clone temporel puisse les refaire

TimeClone.qd : clone temporel du joueur qui sera instancié à la fin du rembobinage temporel effectué par TimeControl, qui répétera les actions du joueur qu'il a effectué durant la trame temporelle de sélection

Champs :

inputs : Array : contient la liste des inputs du joueur pour un rembobinage

input : String : contient l'input du joueur à une frame donnée

facing : Bool : contient l'état d'orientation du clone temporel (droite ou gauche)

→ **_ready()**: appelée lors de l'initialisation; ajoute le clone au groupe "timeclone" et lui attribue une durée de vie et une courte durée pendant laquelle il ignore les collisions avec le joueur

→ **_physics_process(delta: float)**: appelée à chaque frame du jeu; attribue sa vitesse au clone

→ **calculate_move_velocity(**

linear_velocity: Vector2,

direction: Vector2,

speed: Vector2,

is_jump_interrupted: bool

): calcule la vitesse à attribuer au joueur

→ **get_direction()**: traduit les commandes enregistrées en direction

→ **movements_animation(direction: Vector2)**: lance les animations correspondant aux mouvements du clone

→ **clone_action(input)**: appelée à chaque frame; traduit les commandes enregistrées en actions (comme attaquer) pour le clone

→ **jump()**: appelée lorsque le clone saute; donne une impulsion vers le haut au clone

→ **_on_PhysicalHitbox_body_entered(body: Node)**: appelée lorsque le clone rentre en contact avec un corps étranger et appelle la fonction **hit()** si ce corps est un ennemi

→ **hit(dmg)**: appelée par la fonction **_on_PhysicalHitbox_body_entered()**, est la fonction qui traduit le fait que le clone temporel est touché : il meurt immédiatement (appelle **die()**)

→ **die()**: appelée par **hit()**, se contente de détruire l'instance du clone

→ **attack()**: permet au clone d'attaquer en créant une instance de MeleeAttack en face de TimeClone

→ **on_TimeResetTimer_timeout()**: appelée quand le clone atteint la fin de sa durée de vie; appelle **die()**

→ **_on_PhysicalHitbox_area_entered(area: Area2D)**: fonction appelée dès que le clone rentre dans une zone. Teste notamment si le clone est rentré un champ de gravité pour activer ses effets

→ **_on_PhysicalHitbox_area_exited(area: Area2D)**: fonction appelée dès que le clone sort d'une zone. Teste notamment si le clone est sorti d'un champ de gravité pour désactiver ses effets.

Autoload : Contient tous les éléments du jeu qui sont “autoloadés”, c'est-à-dire qu'ils sont toujours actifs quoiqu'il arrive, peu importe si on change de niveau ou non - cela permet d'avoir des variables qui peuvent être actualisées et lues par tous les éléments du jeu à n'importe quel moment

GeneralData.qd : contient toutes les informations qui doivent être conservées à travers le jeu (vie du joueur et plus tard un score)

Champs :

player_hp : Int : contient la vie (entier) du joueur

→ **reset()**: appelée lors du reset du jeu, elle remet à l'état initial la vie du joueur

→ **set_player_hp()**: fonction appelée par les autres scripts dans le jeu, permet d'actualiser la vie du joueur (qui est stockée dans GeneralData)

→ **get_player_hp()**: fonction appelée par les autres scripts pour que ceux-ci puissent lire et récupérer la vie du joueur contenue dans GeneralData

→ **music()**: fonction appelée par les autres scripts, qui permet de lancer la musique du jeu

TimeControl.qd : gère le rembobinage temporel en lancant une trame de sélection temporelle au cours de laquelle tous les éléments du jeu enregistrent leurs positions initiales et le joueur enregistre les différentes inputs de l'utilisateur ; puis à la fin de cette sélection (déterminée par un timer de 5 secondes), le “rembobinage” s'effectue, en remettant les entités à leurs positions initiales

Champs :

player_inputs : Array : contient la liste des inputs du joueur

clone_exists : Bool : contient l'état d'existence d'un clone temporel

→ **timereset()**: appelée par Player lorsque le skill de contrôle temporel est activée par l'utilisateur ; appelle la fonction **save()** de toutes les entités affectées par le rembobinage temporel (pour le moment seulement le joueur et l'ennemi) et lance le timer de sélection de la trame temporelle

→ **_on_TimeResetTimer_timeout()**: appelée à la fin du timer, lance le rembobinage en appelant la fonction **timeReset()** des entités affectées par le rembobinage temporel

→ **set_player_inputs(value)** : appelée par Player lors du rembobinage temporel, ajoute à la liste player_inputs les inputs du joueur à chaque frame

→ **get_player_inputs()** : appelée par TimeClone à la fin du rembobinage temporel, renvoie player_inputs.

→ **set_clone_exists(value)** : appelée par Player, modifie clone_exists

→ **get_clone_exists()** : appelée par Boss, renvoie la valeur clone_exists

Entities: contient tous les éléments du jeu qui restent inertes sauf lorsqu'ils interagissent entre eux ou avec un acteur

[Box.gd](#) : Une simple boîte que le joueur peut pousser pour des puzzles

Champs :

nb_champs : int : *contient le nombre de champs que subit la boîte*

timeposition : Vector2 : *contient la position de Box avant le rembobinage*

temporel

→ **_ready()**: appelée à l'instanciation de la caisse; ajoute la caisse au groupe "timecontrol"

→ **save()**: appelée par TimeControl, sauvegarde la position actuelle de la boîte dans timeposition

→ **timeReset()**: fonction temporelle appelée par TimeControl à la fin de la sélection de la trame temporelle de rembobinage, qui change la position de la boîte à sa position au début de ladite trame (celle dans timeposition)

→ **_on_PhysicalHitbox_area_entered(area: Area2D)**: fonction appelée dès que la caisse entre dans une zone. Teste notamment si la caisse est rentrée dans un champ de gravité pour activer les effets de ce dernier et augmenter nb_champs de 1.

→ **_on_PhysicalHitbox_area_exited(area: Area2D)**: fonction appelée dès que la caisse sort d'une zone. Teste notamment si la caisse est sortie d'un champ de gravité pour désactiver les effets de ce dernier et diminuer nb_champs de 1.

[DoorButton.gd](#) : Un bouton qui peut appeler des fonctions dans d'autres scripts

Champs :

nb_corps: int : *contient le nombre de corps appuyant sur le bouton*

→ **_ready()**: appelée à l'instanciation de l'entité ; place le bouton dans la bonne animation

→ **_on_body_entered(body: KinematicBody2D)**: quand un corps entre en collision avec le bouton, une animation joue et nb_corps augmente de 1

→ **_on_body_exited(body: KinematicBody2D)**: quand un corps sort de collision avec le bouton, une animation joue et nb_corps diminue de 1

[Laser.gd](#) : Un piège simple qui capte les signaux envoyés par un bouton

Champs :

nb_corps: int : *contient le nombre de corps appuyant sur le bouton lié au laser*

→ **_ready()**: appelée à l'instanciation; place le laser dans la bonne animation et le bon niveau de détection de collision

→ **_on_Button_body_entered(body: Node)**: augmente nb_corps de 1 et désactive la détection des collisions du laser quand le bouton associé est enfoncé

→ **_on_Button_body_exited(body: Node)**: diminue nb_corps de 1 et désactive la détection des collisions du laser quand le bouton associé est enfoncé (nb_corps = 0)

→ **_on_Laser_body_entered(body: Node)**: quand un corps entre en collision avec le laser, cette fonction appelle la fonction hit du corps, s'il en possède une

[Level_End.gd](#) : Un portail qui permet de changer de niveaux

Champs :

next_level: String: *contient le chemin d'accès à la scène suivante*

→ **change_level()**: appelée par **_on_Level_End_body_entered()**; permet de changer de niveau en changeant de la scène courante à la scène indiquée par next_level

→ **_on_Level_End_body_entered()**: appelée lorsque le joueur rentre en contact avec le portail, appelle change_level()

MeleeAttack.gd : L'attaque du joueur qui permet d'attaquer un ennemi à côté de lui ; comme c'est une attaque de mêlée, elle doit s'effacer 0.5 seconde après son initialisation

→ **_ready()**: appelée lors de l'initialisation de l'instance de MeleeAttack, permet de lancer le timer DestroyTimer

→ **_on_body_entered(body: Node)**: appelée lorsque qu'un corps étranger rentre en contact avec MeleeAttack et appelle la fonction hit() de ce-dit corps

→ **_on_DestroyTimer_timeout()**: appelée lorsque le DestroyTimer atteint sa limite de temps, détruit l'instance de MeleeAttack

Projectile.gd : L'attaque à distance du boss qui vise le clone du joueur s'il existe, et le joueur sinon. Au cas où le projectile ne rencontre rien, il a quand même une durée de vie limitée

→ **_ready()**: appelée lors de l'instanciation; ajoute le projectile au groupe "projectile"

→ **_on_LifeTimer_timeout()**: appelée lorsque le LifeTimer atteint sa limite de temps, détruit l'instance de Projectile

→ **_on_body_entered(body: Node)**: appelée lorsque qu'un corps étranger rentre en contact avec Projectile et appelle la fonction hit() de ce-dit corps avant de détruire l'instance du projectile

Gravity : contient tout ce qui est relatif aux champs de gravité du jeu

----- [GravityField.gd](#) : Une zone dont on peut modifier la gravité avec la souris. Il suffit de cliquer et de glisser la souris au sein de la zone pour agir sur le champ de gravité.

Champs:

gravity_field : Vector 2, contient le vecteur champs de gravité de la zone qui s'applique sur le joueur et les ennemis. (celui pour les caisses est interne au moteur de jeu et est modifié au sein du

norm_max: float, correspond à la norme max du vecteur champ

norm_min: float , correspond à la norme minimale du vecteur champ

norm_default: float, correspond à la norme par défaut du vecteur champs

sensibility: float, facteur de sensibilité de la gravité (plus il est élevé, plus le champs de gravité est modifié rapidement par le joueur)

origin: Vector2, contient les coordonnées du dernier clic de la souris (pour calculer la nouvelle gravité plus tard)

is_dragging: bool, indique si le joueur est en train de modifier la gravité (click'n drag)

is_in_zone: bool, indique si la souris est dans la zone de gravité.

→ **_ready()**: appelée lors de l'initialisation de l'instance de GravityField, permet d'intialiser **gravity_field** ainsi que le vecteur gravité interne au moteur pour les caisses.

→ **_input(event: InputEvent)**: fonction appelée dès qu'un événement se produit (mouvement, ou clic de la souris ici). Traduit les commandes à la souris pour modifier le vecteur champ de la zone de gravité uniquement si le clic s'effectue au sein de la zone (uniquement si **is_in_zone=true** pour commencer à modifier le champs ou si **is_dragging = true** si le champs est déjà en cours de modification) . Appelle **calculate_gravity(Vector2, Vector2)** pour mettre à jour en temps réel **gravity_field**.

→ **indique_sens_intensite(angle: float, intensite: float)** : méthode qui modifie la direction (en fonction de **angle**) et l'épaisseurs et la direction des flèches (en fonction de **intensite**).

→ **get_actual_gravity()**: fonction qui renvoie le vecteur champ de gravité actuel de la zone (**gravity_field**).

→ **calculate_gravity(origin: Vector2, end: Vector2)**: Cette fonction permet de calculer le nouveau vecteur champs de gravité à partir des inputs de la souris

→ **_on_GravityField_mouse_entered()**: Cette fonction est appelée quand la souris rentre dans la zone de gravité. Modifie le booléen indiquant si la souris est dans la zone sur true.

→ **_on_GravityField_mouse_exited()**: Cette fonction est appelée quand la souris sort de la zone de gravité. Modifie le booléen indiquant si la souris est dans la zone sur false.

Levels : contient les niveaux du jeu ainsi que les écrans de début, fin et victoire du jeu. Ne contient aucun script (tout est assuré par les scripts des différents modules précédents).

UserInterface : Contient tous les éléments relatifs à l'interface utilisateur, comme les écrans de début/fin/pause et le HUD

[ChangeSceneButton.gd](#) : Bouton qui permet de changer de scène (niveaux)

Champs:

next_scene_path: String: contient le chemin d'accès à la scène suivante

→ _on_ChangeSceneButton_button_up(): Appelée lorsque le bouton est cliqué. Appelle reset() de GeneralData, change le niveau indiqué par next_scene_path et lance la musique

→ _get_configuration_warning(): Renvoie un message si le bouton n'est pas lié à un niveau (jamais appelée par le code)

[HUD.gd](#) : Element qui s'affiche perpétuellement sur l'écran et qui affiche les éléments importants tel la vie du joueur

→ _process(delta): appelée à chaque frame du jeu, affiche la vie du joueur actualisée

[Pause.gd](#) : Ecran de pause

→ _ready(): Appelée lorsque la pause est lancée. Affiche l'écran de pause.

→ _input(): Appelée lorsqu'une touche est pressée. Dé-pause le jeu si la touche est Echap.

[QuitButton.gd](#) : Bouton pour quitter le jeu

→ _on_QuitButton_button_up(): Appelée lorsque le bouton est cliqué. Quitte le jeu

[RetryButton.gd](#) : Bouton pour relancer le jeu

→ _on_RetryButton_button_up(): Appelée lorsque le bouton est cliqué. Relance le jeu

[TestButton.gd](#) : Bouton pour lancer les tests

→ _on_ChangeSceneButton_button_up(): Appelée lorsque le bouton est cliqué. Lance l'écran de Test

→ _get_configuration_warning(): Renvoie un message si le bouton n'est pas lié à un niveau (jamais appelée par le code)

[TimeHUD.gd](#) : HUD pour le rembobinage temporel (affiche le temps restant)

Champs :

tempis_restant : int : contient le temps restant au rembobinage temporel

→ _process(delta): Appelée à chaque frame. Affiche tempis_restant

→ _on_TimeControlTimer_timeout(): Appelée lorsque le TimeControlTimer s'épuise (à chaque seconde). Décrémente tempis_restant

World - Contient la tilemap. Ne contient aucun script (tout est assuré par les scripts des différents modules précédents).

5/ Tests

3.1-Tests unitaires

Pour tester les méthodes de chaque classe, nous avons fait appel à un plug-in de Godot dont le nom est *Gut* (les fichiers permettant à *Gut* de fonctionner se situent dans /addons/gut). Concrètement, Gut permet de lancer automatiquement tous les scripts de test unitaires et d'intégration que nous avons fait et de les afficher dans une fenêtre graphique. (Pour lancer cette fenêtre il suffit de cliquer sur le bouton "Test" au début du jeu ou sur l'écran pause. Cf **Manuel d'utilisateur** pour plus de détails)

Dans le dossier de tests unitaires (chemin ./test/unit) se trouvent tous les scripts pour effectuer les tests unitaires. Il y en a un pour chaque classe, et les noms de ces scripts suivent la nomenclature **test_nomdelaclassel_unit**. Dans chacun des scripts se situent tous les tests unitaire (commentés) de chaque fonction de la classe correspondante. Pour chaque fonction, les test associés sont effectués au sein de la méthode **test_nomdelafonction()**. Les **assert_eq(argument1, argument2, "Message d'explication")** servent à vérifier si argument1 == argument2. Si c'est bien le cas, le test réussit. Sinon il échoue. L'argument "Message d'explication" sert à afficher un message indiquant ce qu'il est attendu pour que le test réussisse. Au cours de ces tests nous avons examiné les cas limites, et intermédiaires. Par exemple, la méthode **test_calculate_gravity()** dans le script **test_gravity_field_unit()** doit bien s'assurer que la norme du champ de gravité est bien située entre la norme minimale et la norme maximale autorisées.

NB: Les tests unitaires faisant appel à des inputs du joueur (souris, clavier etc) ne peuvent pas être testés à l'aide de scripts, du moins nous n'avons pas trouvé le moyen de les simuler et les tester directement dans le script. De la même manière, pour raison que nous ignorons, les fonctions faisant appel à des animations causent un bogue et ne peuvent pas être testées unitairement.

3.2-Tests d'intégration

Maintenant que les tests unitaires ont été effectués et fonctionnent bien, nous allons détailler le fonctionnement des tests d'intégration concernant toutes les méthodes qui font interagir plusieurs classes distinctes.

De la même manière que pour les tests unitaires, Gut permet de lire et lancer automatiquement les scripts des tests d'intégrations que nous avons écrit au préalable. Dans le dossier de tests d'intégration (chemin ./test/integration) se trouvent tous les scripts pour effectuer les tests d'intégration. Il y en a un pour chaque classe, et les noms de ces scripts suivent la nomenclature **test_nomdelaclassel_integration**. Dans chacun des scripts se situent tous les tests d'intégrations (commentés) de chaque fonction de la classe correspondante. De la même manière que pour les test unitaire, nous faisons appel à **assert_eq(argument1, argument2, "Message d'explication")** vérifier si argument1 (ce que l'on veut tester) correspond bien à ce que est attendu (argument2)

avec un message d'explication pour détailler ce qui est attendu de la part de la méthode en fonction de ses entrées.

NB: Comme pour les tests unitaires, toutes les méthodes faisant directement appel à des inputs du joueur (souris, clavier etc) ne peuvent pas être testées à l'aide de scripts. Les méthodes qui font appel à des animations posent également les mêmes problèmes que lors des tests unitaires.

3.3-Tests de validation

À présent que les tests d'intégration ont été réalisés, nous pouvons effectuer les tests de validation du jeu (que nous vous exposons sous la forme de captures d'écran du jeu, faute de pouvoir faire mieux). En effet, puisqu'il s'agit d'un jeu vidéo, quelques images isolées ne suffisent pas toujours pour montrer ce qu'il se passe précisément, nous pourrons alors faire une démonstration plus détaillée lors de la soutenance.

- Marche gauche/droite du perso

Avant:



Après:



Marche gauche



Marche droite

Test validé

- Saut du Perso

Avant:



Après:



Test validé

- Attaque du perso



Animation

Avant:



Après:



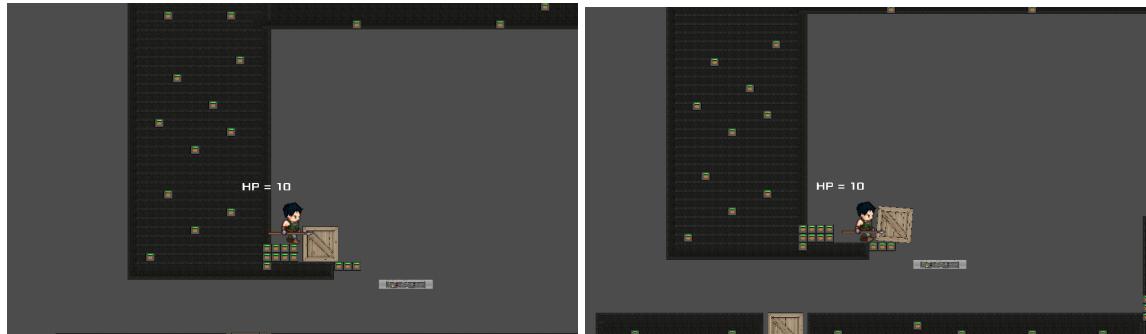
Attaque sur un ennemi

Test validé

- Pousser caisses

Avant:

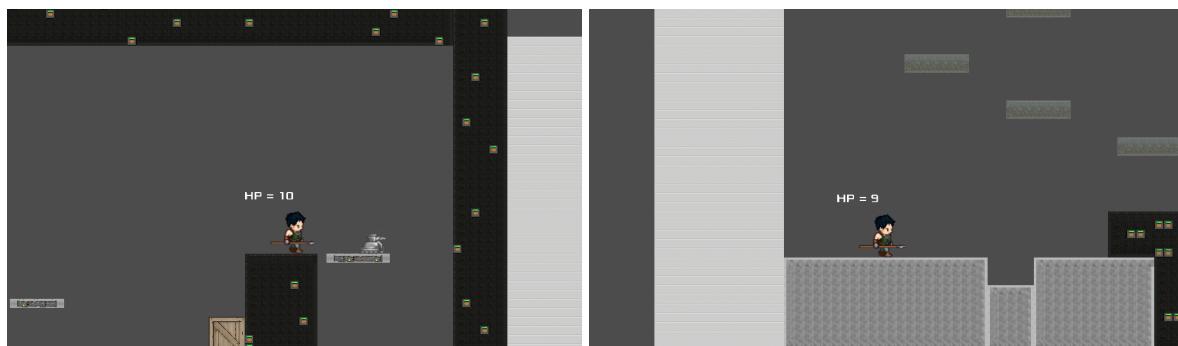
Après:



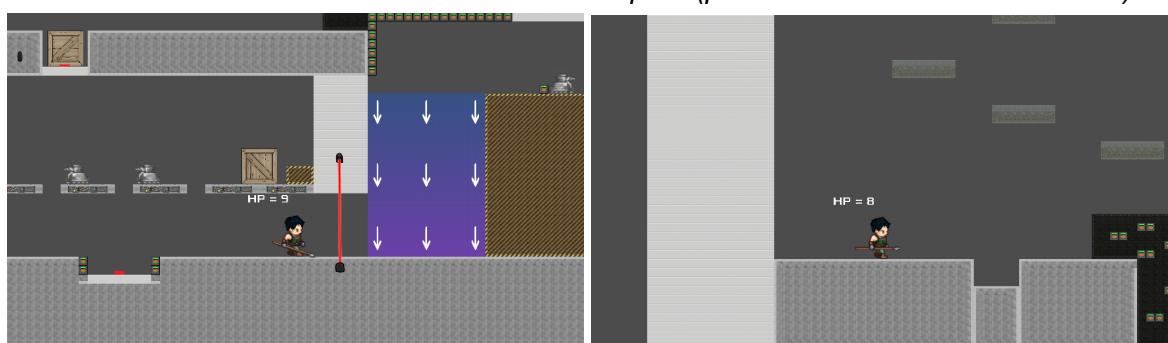
Test validé

- Laser et Ennemis font mal

Avant contact:



Avant contact:



Test validé

- Bouton désactive lasers

Avant:

Après:



Test validé

- Personnage, Clone, Caisses et Ennemis activent boutons



Personnage

Clone



Caisse

Avant:



Après:



Ennemi

Test validé

- Créez clone qui reproduit actions (exemple: saut)

Avant:



Après:



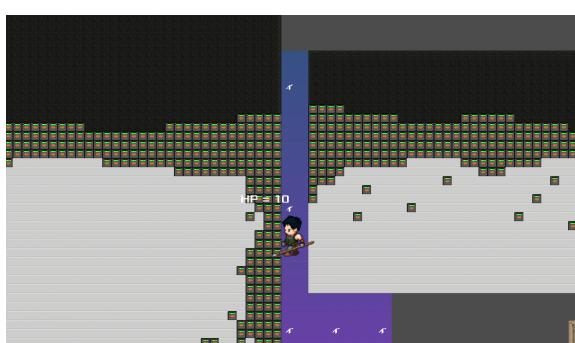
Test validé

- Changer Gravité

Avant:



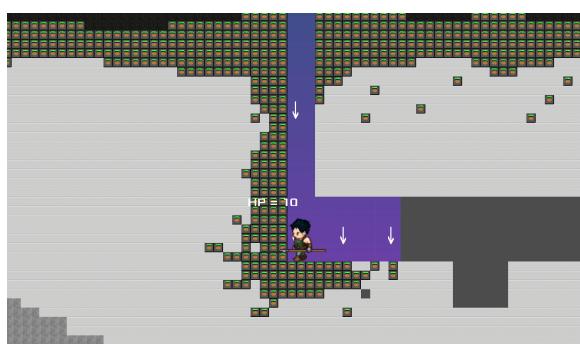
Après:



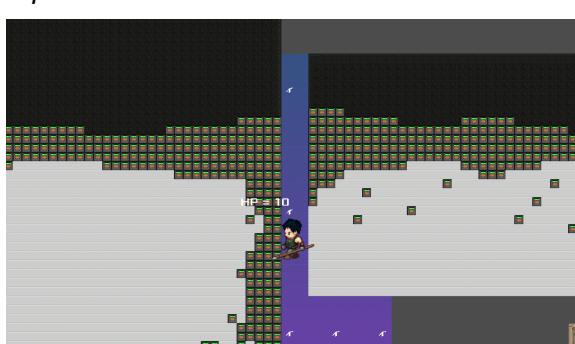
Test validé

- Gravité Affecte Joueurs, ennemis et caisses

Avant:



Après:



Joueur

Avant:

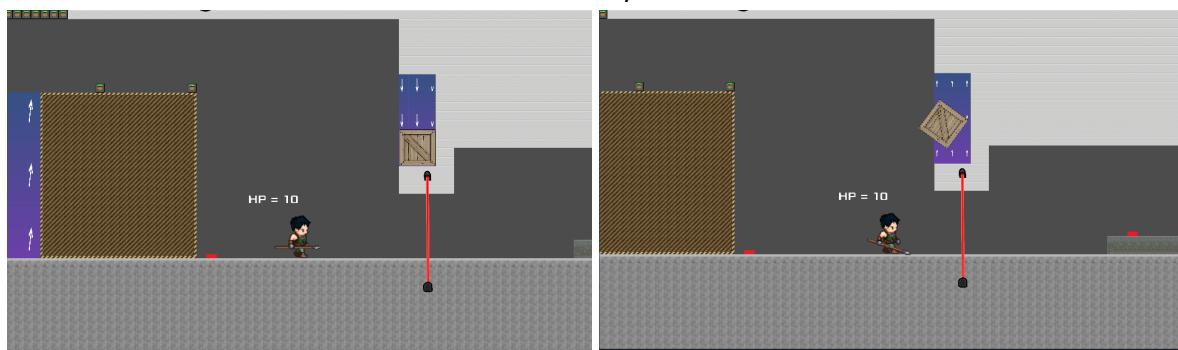


Après:



Ennemi

Avant:

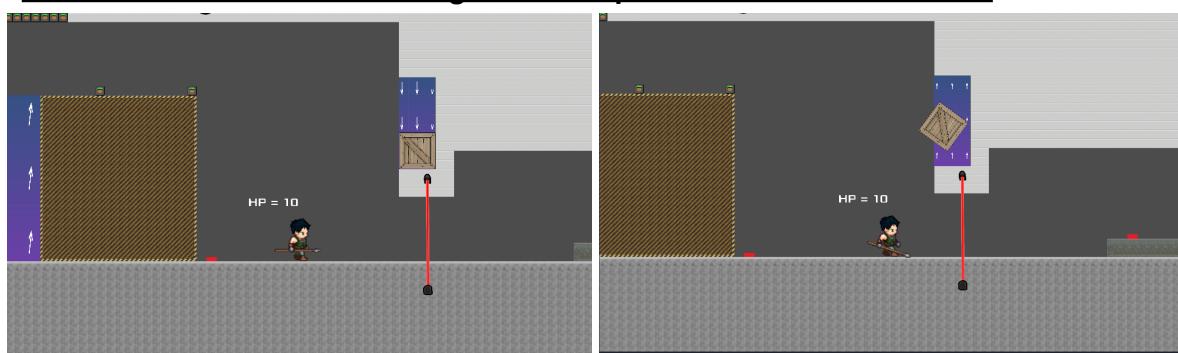


Après:

Caisse

Test validé

- Les flèches dans les zones de gravité indiquent les bonnes directions



Vers le bas, la caisse reste sur le sol

Vers le haut, la caisse s'élève

Test validé

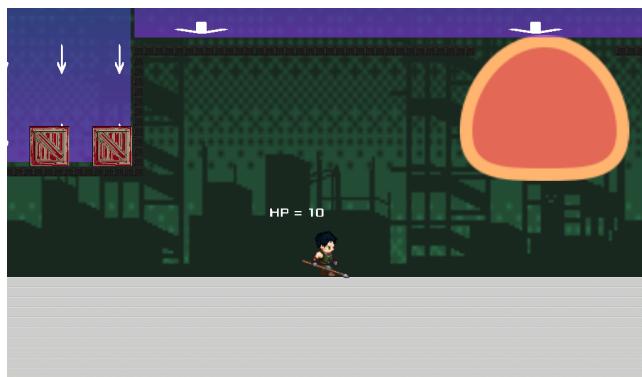
- Gravité n'affecte pas Clone



Les flèches sont vers le haut, le personnage et la caisse s'élèvent, mais pas le clone.

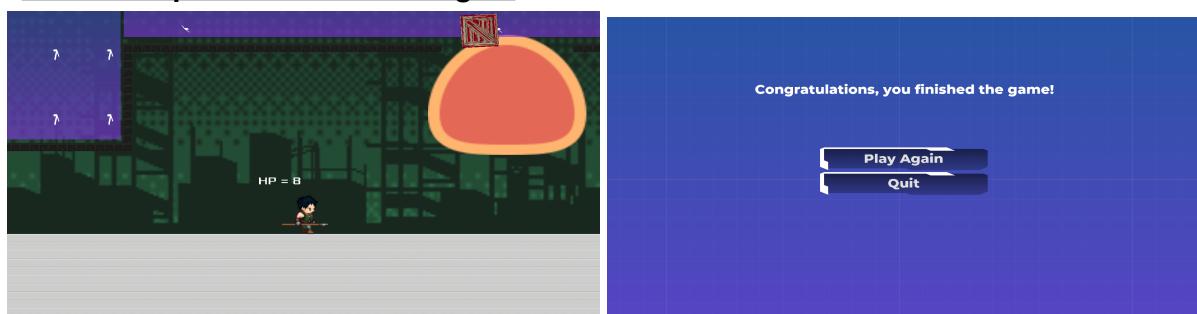
Test validé

- Gravité n'affecte pas le boss: il n'est pas dans un champ de gravité et n'est donc pas affecté



Test validé

- Caisses explosives font des dégâts



Le boss perd un HP à chaque caisse qui le touche, au bout de trois, il meurt et le niveau est terminé.

Test validé

- Boss tire vers le joueur



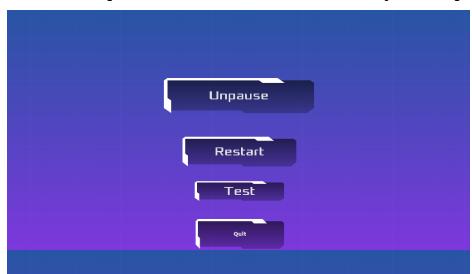
Test semi-validé : quand on lance le combat de boss directement, il tire vers le joueur. Cependant, lorsqu'on fait le niveau normalement et qu'on arrive au boss, tant qu'on invoque pas le clone ou qu'on ne touche pas le boss, il ne vise pas le joueur.

- Boss tire vers le clone si il existe



Test validé

- Menu pause marche bien (lorsqu'on appuie sur échap)



Test validé

- HUD existe quand on invoque le clone

Temps restant: 4s



Temps restant: 2s



Lorsqu'on invoque le clone, il apparaît au bout de 5 secondes, un HUD indique ce timer.

Test validé

De plus, les tests de son sont tous validés (cf prototype).

6/ Manuel utilisateur

Nous allons décomposer ce manuel utilisateur en 3 parties :

6.1-Manuel utilisateur pour jouer au jeu

6.2-Manuel pour tester le jeu

6.3-Manuel d'utilisation du code

6.1-Manuel utilisateur pour jouer au jeu

Le jeu nécessite un clavier et une souris. Un pavé tactile n'est pas bien adapté pour le jeu. Pour lancer le jeu, il suffit de cliquer sur l'exécutable “**Gravitime.exe**” présent dans le dossier. Il s'agit d'un jeu de type “arcade” : la progression du joueur n'est pas sauvegardée d'une partie sur une autre.

Touches utilisées par le joueur :

Z/Flèche Haut - sauter

Q/Flèche Gauche - bouger vers la gauche

D/Flèche Droite - bouger vers la droite

E - attaquer

F - démarrer l'enregistrement pour remonter dans le temps

Souris (click and drag) (sur les champs de gravité uniquement) - modifier la direction et l'intensité des champs de gravité

Echap - permet de mettre le jeu en pause

Lorsque la touche F est appuyée, les actions du joueur sont enregistrées pendant 5 secondes. A la fin de ce temps, le temps est rembobiné, et un clone répète les actions du joueur. Le clone a une hitbox qui lui permet d'interagir avec le monde et au joueur d'interagir avec lui.

Le joueur démarre avec **10 points de vie (HP)**. Un contact avec un ennemi, un projectile ou un laser cause la perte d'un point de vie et le ramène au dernier point de spawn.

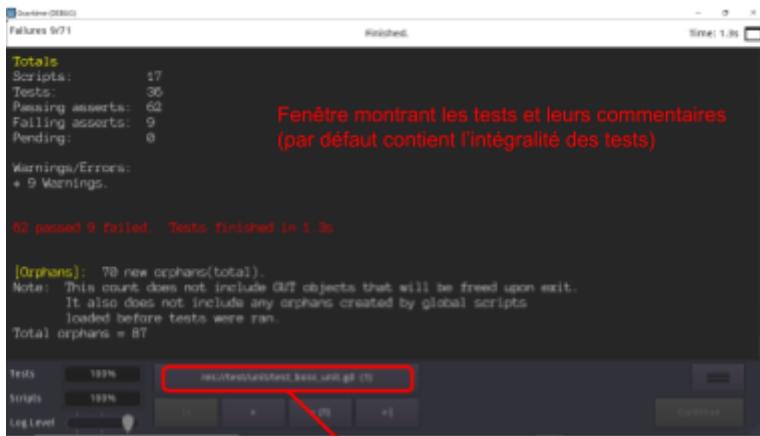
Le boss ne peut être endommagé que par un **impact avec une caisse explosive**. Il possède **3 points de vie**.

6.2-Manuel utilisateur pour les tests

L'entièreté des scripts des tests est accessible dans le dossier **test** du jeu.

Pour accéder aux tests (et les lancer), il faut cliquer sur le bouton **Tests** présent sur l'**écran de pause** et du **menu principal**.

Une fois le bouton appuyé, l'intégralité des tests se lancent automatiquement. Notez qu'une fois en mode Test, l'utilisateur ne peut revenir au jeu sans quitter et relancer le .exe.



Fenêtre montrant les tests et leurs commentaires
(par défaut contient l'intégralité des tests)

Menu déroulant permettant de sélectionner les différents tests individuellement

Espace des tests légendé

L'espace de test se décompose en plusieurs éléments, dont le plus important est la fenêtre présentant les tests. Par défaut, celle-ci contient l'intégralité des tests contenus par le projet (unitaires et d'intégration).

On peut sélectionner un test individuel (et l'appliquer) par le menu déroulant en bas de la fenêtre.

6.3-Manuel d'utilisation du code

Le projet se divise en 2 parties : les **Scenes** (.tscn) et les **Script** (.gd). Toutes les deux peuvent être ouvertes par un éditeur de texte à priori (vérifié avec WordPad sur Windows et TextEdit sur Mac). Le code correspond aux **Scripts**, les **Scenes** étant utilisées par le moteur de jeu.

Le code suit les conventions suivantes :

Variables/Champs : minuscules complètes

Classes : Majuscule à chaque début de mot

Méthodes : minuscules avec underscore si besoin

Variables indiquant un état (booléen) : minuscule avec underscore et commençant "is_"

Nodes : on garde une trace du type de la node (ex: "ReloadTimer", "JumpTimer" etc)

Tests : "test_nomdeclasse_unit.gd" pour les tests unitaires et "test_nomdeclasse_integration.gd" pour les tests d'intégration.

Une liste exhaustive des classes, de leurs méthodes et des variables utilisées est disponible dans la partie **Conception détaillée** de ce rapport.

7 / Bilan

Ce rapport décrit l'état du projet "Gravitime". Nous avons réussi à produire un jeu vidéo fonctionnel suivant le cahier des charges initial :

Le joueur est capable de se mouvoir dans un environnement 2D avec vue de côté et d'interagir avec les éléments du jeu : pièges, lasers, caisses, ennemis etc...

Les champs de gravité sont bien implémentés et modifiables par le joueur comme prévu. Les champs, leur direction et leur intensité sont clairement indiqués pour le joueur. Nous n'avons pas eu la chance de créer des ennemis de masses différentes qui seraient plus ou moins affectés par les champs de gravité, permettant ainsi de produire des puzzles plus complexes et variés.

La mécanique de contrôle du temps, bien que source de nombreux bugs, correspond bien à la vision que nous avions d'elle: les objets sont ramenés à la position où ils étaient lorsque le joueur active le contrôle du temps, les actions du joueur sont enregistrées et reproduites en gardant la taille des données sous contrôle, et le clone peut bien interagir avec le joueur et le reste du jeu.

Malheureusement, nous n'avons pas eu le temps de développer un ennemi capable de manipuler lui-même la gravité.

Le boss final est capable de prioriser entre le clone temporel et le joueur, en tirant sur le premier si celui-ci existe. Néanmoins, il n'est pas capable de manipuler la gravité comme nous ne nous l'étions fixé au début du projet.

Pour conclure, le progrès initial sur le projet a été très satisfaisant, nous permettant de programmer la quasi-totalité des capacités du joueur et des objets avec lesquels il viendra à interagir. Malheureusement, certains éléments souffrent toujours de bugs divers. Finalement, le jeu contient la quasi-totalité des éléments que nous voulions implémenter.

8/ Bibliographie

[1] Plugin GUT, Butch Wesley : <https://github.com/bitwes/Gut>