Jume

Oct 12, 2019

Contents

This documentation describe the Godot project *Jume*, in order to make it easier to modify the code base if needed.

Jume is a side-project developped by myself. I really liked the *Archero* game on Android, so I decided to create a similar game !

CHAPTER 1

Installation

1.1 Install Godot

Download and install Godot from their official website.

1.2 Clone this repository

Clone the repository locally :

```
git clone https://github.com/astariul/jume.git
```

And now you're good to go !
Now just open Godot and import the project from where you cloned it.

CHAPTER 2

Nodes

2.1 Entity

Abstract Node for common behavior between all entities.

Inherits : KinematicBody2D

Inherited by : *Player*, *Ennemy*

2.1.1 Properties

knockbacker_pos
knockbacker_pow
knockback_frame
knockback_ratios
knockback_power
damage
destination
target
speed

2.1.2 Methods

set_destination()
reset_destination()
get_base_velocity()
knockback_from()
apply_knockback()
is_targetting()

select_target()
fire()
interrupt_shooting()
hit_by()

2.1.3 Constants

SELECT_CLOSEST = "closest"

One of existing targetting method.
This one will choose the closest ennemy to the player.

2.1.4 Signals

running

Signal emitted when the entity is moving.
This signal should be emitted from the `_physics_process()` function of the child class (not handled in this class).

Emitted by : *Player._process()* **Received by :** *Shooter._on_running_timeout()*

immobile

Signal emitted when the entity is **not** moving.
This signal should be emitted from the `_physics_process()` function of the child class (not handled in this class).

Emitted by : *Player._process()* **Received by :** *Shooter._on_immobile_timeout()*

2.1.5 Description

This is an *Abstract Node*, meaning it's just a script (there is no scene associated to this script).

It is used to define common behavior between all entities to avoid code duplication.

Common behaviors handled by this Node are mainly :

- Common properties and associated methods
- Knockback
- Fire projectile
- Hit by projectile

2.1.6 Properties description

`knockbacker_pos`

Type	*Vector2*
Default	`null`

Position of the knockbacker.
When an entity is knockbacked, the knockbacker position is kept in memory and used later for knockback direction computations.

knockbacker_pow

Type	*int*

Power of the knockbacker.
When an entity is knockbacked, the knockbacker power is kept in memory and used later for knockback direction computations.

knockback_frame

Type	*int*
Default	`0`

Current frame of the knockback.
Used together with *knockback_ratios* in order to create a smooth knockback animation.

knockback_ratios

Type	*array* of *float*
Default	`[9, 8, 6.5, 4.5, 2]`

Speed ratios of the knockback.
Used to create a smooth knockback animation.

knockback_power

Type	*int*
Default	`100`

Knockback power of this entity.

damage

Type	*int*
Default	`10`

Knockback damages inflicted by this entity to other knockbacked entities.

destination

Type	*Vector2*
Default	`null`

Aimed position by the entity.
This is used to control to entity : it will go in the direction of destination.
For example, if the entity is located to `(0, 0)` and the destination is set to `(1, 0)`, entity will move to the right.

target

Type	WeakRef
Default	`null`

Reference to the targetted entity.
Entity may be targetting nothing. The target is used when firing weapon : the bullet will be fired in the direction of the current target (and hopefully touch the target !).

speed

Type	*int*
Default	`100`

Walking speed of the entity.

2.1.7 Methods description

set_destination(dest=null)

Arguments
dest
The new destination for this entity.
Default to `null`.

Method used to set a new destination (or update current destination) for this entity.
Destination is not updated if `null` is given.

reset_destination()

Reset the current destination to `null`.

get_base_velocity()

Return *Vector2*

Compute the base velocity of the entity, going in the direction of the *destination* using *speed*.
The computed velocity is returned.

knockback_from(collider)

Arguments
collider
The Entity colliding with the current entity.
Position and *knockback_power* of the collider is used to compute knockback.

Knockback the current entity. This method only save the value of *knockbacker_pos* and *knockbacker_pow*.
The knockback computations happen in *apply_knockback()*.

apply_knockback(velocity)

Return *Vector2*

Arguments
velocity Velocity to update with the knockback.

Compute the knockback velocity, based on :

- *knockbacker_pos* & *knockbacker_pow*
- current position
- given velocity
- *knockback_ratios* & *knockback_frame*

is_targetting()

Return *bool*

In order to know if the entity is targetting something or not, this method should be called.

`select_target(group_name, selection_method="closest")`

Return *float*

Arguments
group_name The name of the group from where to choose a new target.
selection_method
Selection method to use for choosing a new target.
Defaults to *select_closest*.

Select a new target among the given group of Entity.
For now only one selection method is implemented (*select_closest*), but later it will be possible to choose among other methods.

The value returned is the best value used to compare Entity of the group. For example for *select_closest* method, it is the negation of the distance (so the biggest, the closest).
If there is no Entity is the group, it return `null`.

`fire(projectile, atk_speed)`

Arguments
projectile Instancied projectile to fire.
atk_speed Attack speed to use to fire.

This method is used as a general method for firing a projectile. For specific behaviors, please refer to each Node.

This method compute the angle between the Entity and the target, then rotate the projectile to this direction and update the speed of the projectile to fire it.
It also rotate the Entity sprite to face the target when firing.

This method does nothing if no target is selected.

`interrupt_shooting()`

This method does nothing. It may be overwrited by child Nodes to define a behavior in case shooting should be interrupted. Here there is nothing to interrupt due to the current implementation of *`fire()`*.
It is called by the shooter class.

hit_by(projectile)

Arguments
projectile Projectile colliding with the Entity.

This method does 2 things :

- Update the health bar of the entity based on the damage of the projectile.
- Knockback the entity from the projectile.

2.2 Player

Node representing the playable character.

Inherits : *Entity*

2.2.1 Methods

_ready()
_process()
_physics_process()
_on_HealthBar_dead()
fire()
interrupt_shooting()

2.2.2 Description

This Node represent the playable character.

It's an *Entity*, and simply modify some general behavior into more specific ones :

- Change a few characteristics
- Firing projectile is bow-specific

2.2.3 Methods description

_ready()

Method starting animations, adding the Node to the `players` group, and changing a few *Entity* characteristics.

_process()

Arguments
delta Delta (see Godot documentation for more details).

This method is executed every frame and do several things :

- If the player don't have a target anymore, select a new one.
- Get the input from user and set a new destination accordingly, as well as animation. It does not move the player, simply set the new destination.
- Emit right signal depending if the player is running or not.

Emitted signals
running This signal is emitted when the player is moving.
immobile This signal is emitted when the player is **not** moving.

`_physics_process(delta)`

Arguments
delta Delta (see Godot documentation for more details).

This method takes care of the physics engine processing : it moves the player according to :

- The base velocity of the player (computed from the destination, set in `_process()`)
- The knockback velocity, if any.

`_on_HealthBar_dead()`

Free itself upon receiving the *dead* signal.

Receives signals
dead This signal is emitted when the healthbar reach 0.

`fire(projectile, atk_speed)`

Arguments
projectile Instancied projectile to fire.
atk_speed Attack speed to use to fire.

Overwrite the parent method for firing the projectile. It delegate the actual firing to the *Bow*.

This method does nothing if no target is selected.

`interrupt_shooting()`

This method interrupt the animation of *Bow*.

2.3 Ennemy

Node representing an ennemy, controlled by the computer.

Inherits : *Entity*

2.3.1 Methods

_ready()
_physics_process()
_on_HealthBar_dead()

2.3.2 Description

This Node represent an ennemy, controlled by the computer (need to make IA yet).

It's an *Entity*, and simply modify some general behavior into more specific ones :

- Change a few characteristics

2.3.3 Methods description

_ready()

Method starting animations, adding the Node to the `enemies` group, and changing a few *Entity* characteristics.

_physics_process(delta)

Arguments
delta Delta (see Godot documentation for more details).

This method takes care of the physics engine processing.
It selects a new target if no target are selected, it moves the ennemy according to the base velocity and the knockback velocity if any, and it emits the right signal depending if the ennemy is running or not.

_on_HealthBar_dead()

Free itself upon receiving the *dead* signal.

Receives signals
dead This signal is emitted when the healthbar reach 0.

2.4 Projectile

Abstract Node for common behavior between all projectiles.

Inherits : KinematicBody2D

Inherited by : *Arrow*, *Bullet*

2.4.1 Properties

speed
bounce_nb
remain_time
damage
knockback_power
velocity

2.4.2 Methods

init()
physics_process()
bounce()
impact()
_on_timer_timeout()

2.4.3 Description

This is an *Abstract Node*, meaning it's just a script (there is no scene associated to this script).

It is used to define common behavior between all projectiles to avoid code duplication.

Common behaviors handled by this Node are mainly :

- Collision with Entities
- Bouncing (or not) on walls
- Projectile being stabbed into walls

2.4.4 Properties description

speed

Type	*int*
Default	1000

Speed of the projectile.

bounce_nb

Type	*int*
Default	`0`

Number of walls-bounce allowed.

remain_time

Type	*float*
Default	`0`

Number of seconds the projectile stay stabbed into a wall before being freed.

damage

Type	*int*
Default	`25`

Amount of damage inflicted when entity collide with this projectile.

knockback_power

Type	*int*
Default	`10`

Knockback power of this projectile.

velocity

Type	*Vector2*
Default	`(0, 0)`

Current velocity of the projectile.

2.4.5 Methods description

`init(s=1000, d=25, bn=0, rt=0, kp=10)`

Arguments

s
The new *speed* of the created Projectile.
Default to `1000`.

d
The new *damage* of the created Projectile.
Default to `25`.

bn
The new *bounce_nb* of the created Projectile.
Default to `0`.

rt
The new *remain_time* of the created Projectile.
Default to `0`.

kp
The new *knockback_power* of the created Projectile.
Default to `10`.

This method is used to initialize as we want a new projectile, instead of setting each property by hand.

`physics_process()`

Arguments

delta Delta (see Godot documentation for more details).

Process the main physic of the projectile :

- Move according to current *velocity*
- If a collision with an entity happen, free this projectile and hit the entity.
- If a collision with a wall happen, bounce, and later stay stabbed in the wall.

`bounce(collision)`

Return *bool*

Arguments

collider Colliding object.

If the projectile can still bounce (*bounce_nb* is not exhausted yet), update the current *velocity* to account the bounce.
It returns `true` if the projectile is bounced, `false` if it cannot bounce anymore.

impact()

Stab the projectile into the wall (effectively immobolizing the projectile).
It starts a timer for *remain_time* seconds, which call *_on_timer_timeout()* when done.

_on_timer_timeout()

Method called when the projectile have been stabbed into the wall for *remain_time* seconds.
Simply free the projectile.

Receives signals
timeout This signal is emitted when the timer (that waited for *remain_time* seconds) ends.

2.5 Arrow

Node representing an arrow from the player.

Inherits : *Projectile*

2.5.1 Methods

_ready()
_physics_process()

2.5.2 Description

This Node represent an arrow, which is a projectile fired by the player.

It's a *Projectile*, and does not change any of the parent behavior. It is created only for easy understanding of which projectile is friendly and which one is not.

2.5.3 Methods description

_ready()

In this method, the projectile is simply initialized with different values, specific to Arrow.

_physics_process(delta)

Arguments
delta Delta (see Godot documentation for more details).

This method simply calls the parent's physics. There is not additional behavior.

2.6 Bullet

Node representing a bullet from an ennemy.

Inherits : *Projectile*

2.6.1 Methods

_ready()
_physics_process()

2.6.2 Description

This Node represent a bullet, which is a projectile fired by the ennemies.

It's a *Projectile*, and does not change any of the parent behavior. It is created only for easy understanding of which projectile is friendly and which one is not.

2.6.3 Methods description

_ready()

In this method, the projectile is simply initialized with different values, specific to Bullet.

_physics_process(delta)

Arguments
delta Delta (see Godot documentation for more details).

This method simply calls the parent's physics. There is not additional behavior.

2.7 Bow

Node representing the player's weapon : a bow.

Inherits : AnimatedSprite

2.7.1 Properties

curr_projectile
curr_angle

2.7.2 Methods

_ready()
fire()
interrupt_animation()
_on_Bow_animation_finished()

2.7.3 Constants

BASE_FPS = 10

Minimum number of FPS for the bow animation.
The animation can be played faster (if the attack speed increase for example), but never slower.

NB_FRAME = 5 Number of frames for the bow attack animation.

BASE_ANGLE = -PI / 4 Angle of the bow when the player just hold it (not targetting anything).

BACKWARD_SPEED = 2.5

Speed of the projectile when drawing a bow.
It's basically just for smooth animation.

BASE_POS_X = 25

X-Position of the projectile to put it in the right place (the bow).
Originally, the projectile is placed on the center of the entity holding it. We need to change this position, to place the projectile on the bow.

BASE_POS_Y = -2

Y-Position of the projectile to put it in the right place (the bow).
Originally, the projectile is placed on the center of the entity holding it. We need to change this position, to place the projectile on the bow.

2.7.4 Description

This Node represent a bow, the player's weapon.

The code mainly handle smooth animation of firing arrow : going backward a bit, aiming at the right place, and finally firing the arrow !

2.7.5 Properties description

curr_projectile

Type	*:doc:'arrow'*

Current projectile handled and animated by the bow.

curr_angle

Type	*float*

Current targetting angle. At the end of animation, the arrow will be fired in this direction.

2.7.6 Methods description

_ready()

Simply start the animation.

fire(angle, projectile, attack_speed)

Arguments
angle Aimed angle. Arrow should be fired at this angle.
projectile Instancied projectile to fire.
attack_speed Attack speed to use to fire.

This method compute the right FPS (based on the attack speed) and animate the bow as well as the projectile for a smooth animation.

interrupt_animation()

This method interrupt the animation, resetting it and freeing the projectile.

_on_Bow_animation_finished()

This method is executed when the bow animation ends.
It set the current animation back to `idle` and fire the projectile !

Receives signals
animation_finished This signal is emitted when the animation being played ends.

2.8 Shooter

Node creating a level of abstraction in order to fire projectiles.

Inherits : Node2D

2.8.1 Properties

entity
projectile
can_shoot
attack_speed
hit_n_run

2.8.2 Methods

init()
_ready()
set_can_shoot()
_on_start_attack_timeout()
_on_stop_attack_timeout()
_on_Recharging_timeout()
_on_immobile_timeout()
_on_running_timeout()

2.8.3 Signals

start_attack Signal emitted when the entity can start attacking (after stop moving for example).

Emitted by : *set_can_shoot()* **Received by :** *_on_start_attack_timeout()*

stop_attack Signal emitted when the entity should stop attacking (when running for example).

Emitted by : *set_can_shoot()* **Received by :** *_on_stop_attack_timeout()*

2.8.4 Description

This Node acts as a layer. It's a layer above the player, and his job is to instance the projectiles.

Such a trick is needed, because if we simply instancied the projectile in the player Node, projectiles would be childrens of the player, and whenever the player move, the projectiles would move also. We need the position of the player and the projectiles to be independant.

This Node also handle the timer for the attack speed.

2.8.5 Properties description

entity

Type	WeakRef
Default	`null`

Reference to shooting *Entity*.

projectile

Type	*Projectile*
Default	`null`

Non-Instancied *Projectile* to shoot. The Shooter will instanciate a new one every time it fires.

can_shoot

Type	*bool*
Default	`true`

State of the Shooter : if it can shoot (not running for example), it is `true`, else `false`.

attack_speed

Type	*float*

Attack speed to use to fire projectiles.

hit_n_run

Type	*bool*
Default	`false`

If `true`, allow the *Entity* to attack while moving.
This is `false` for the player for example.

2.8.6 Methods description

init(shooting_entity, projectile_to_shoot, attck_spd=1, hit_run=false)

Arguments
shooting_entity Set the shooting *entity*.
projectile_to_shoot Non-instancied *projectile* to use when shooting.
attck_spd Set the *attack_speed*.
hit_run Set the *hit_n_run*.

Method to initialize the object with the value needed.

_ready()

This function simply call the *set_can_shoot()* function at startup time, in order to send the signal.

set_can_shoot(cs)

Arguments
cs *Bool* indicating if the player can shoot or not.

This method change the *can_shoot* property.
If the *can_shoot* is set to `true`, `start_attack` signal is emitted. Otherwise, `stop_attack` is emitted.

Emitted signals
start_attack This signal is emitted if *can_shoot* is set to `true` through this method.
stop_attack This signal is emitted if *can_shoot* is set to `false` through this method.

_on_start_attack_timeout(collider)

Method called when `start_attack` signal is emitted.
It will call the method *_on_Recharging_timeout()* and start the timer according to *attack_speed* for the next projectile.

Receives signals
start_attack This signal is emitted when it's time to shoot.

_on_stop_attack_timeout()

Method called when `stop_attack` signal is emitted.
It will interrupt the timer of attack_speed and call the *Entity* method to potentially interrupt other things such as animations.

Receives signals
stop_attack This signal is emitted when shooting is interrupted.

_on_Recharging_timeout()

Main timer, used for timing every time a projectile is fired, based on *attack_speed*.

If *entity* exist, it instanciate a new *projectile* and fire it.
If *entity* does not exist anymore (may be killed), it does nothing until the last fired projectile is freed (because if we free before, the child projectile will also be freed).

Receives signals
timeout This signal is emitted when the timer (that waited for some time based on *attack_speed*) ends.

_on_immobile_timeout()

Method called when signal `immobile` is emitted.
It simply set *can_shoot* to `true` if it was `false`.

Receives signals
immobile This signal is emitted when the *entity* stopped moving.

_on_running_timeout()

Method called when signal `running` is emitted.
It simply set *can_shoot* to `false` if it was `true`.

Receives signals
running This signal is emitted when the *entity* started moving.

2.9 HealthBar

Node for generic health bar.

Inherits : Node2D

2.9.1 Methods

init()
set_health()
set_max_health()
damage()
heal()

2.9.2 Signals

dead

Signal emitted when the health bar reach `0`.

Emitted by : *damage()* **Received by :** *Ennemy._on_HealthBar_dead()*, *Player._on_HealthBar_dead()*

2.9.3 Description

This is a general-purpose Node, implementing a neat health bar.

It has really basic behavior for now, no regeneration or stuff like this.

Behaviors handled so far :

- Possible to set the color of the health bar (the shadow is always set to orange).
- Change the maximum number of HP.
- Take damages.
- Heal HP.

2.9.4 Methods description

`init(max_health, color=Color(168, 0, 0))`

Arguments
max_health Maximum HP for this health bar.
color
Color of the health bar.
Defaults to red (code RGB : `168, 0, 0`)

Method used to setup the health bar, with a specific number of maximum HP and specific color for the health bar.

`set_health(h)`

Arguments
h New number of HP.

Set the current number of HP to a specific number.
This method should be used (and not manually change the property of the Node) in order to keep the animation clean.

`set_max_health(h)`

Arguments
h New number of maximum HP.

Set the maximum number of HP to a specific number.

damage(d)

Arguments	
d	Amount of damage to inflict.

This method update the health bar based on the amount of damage inflicted.
If the number of HP reach `0`, it emits the dead signal.

Emitted signals	
dead	This signal is emitted when the health bar reach `0` HP.

heal(h)

Arguments	
h	Amount of damage to heal.

This method update the health bar based on the amount of damage healed.
Literally the opposite of the *damage()* function.