Documentation of the Fox2d demo project

Gokudomatic

September 16, 2014
v1.1

Revisions

1.0 Original version using Kirby's assets
1.1 All visual assets and musics from Kirby are replaced by free to use assets

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to explain how the demo project Fox2d is done. This project is a demo made with the Game Engine Godot[1], and it have been made in the context of learning 2d platform games with Godot, and it has been commented and document so it can help new people to start with Godot.
It is however important to note that **it's a documented demo and not a tutorial**. It won't explain the procedure step by step how to make the demo.

1.2 Target audience

This document is destined to those who begin with Godot Engine. A basic knowledge of the Godot IDE and GDScript is still required, as well as some knowledge in general programming.

1.3 Where to start

As said, this document is not a tutorial. The best approach is to get familiarized first with the demo by playing with it. Then, with both Godot and this document opened, discover how are made the main actors, the singleton script files, and the levels.

1.4 About Godot Engine

The Godot Engine is a game engine with its own visual editor and script language. This engine and tool have been internally developed since 2001 by Okame for its own games. Since February 2014 it became open source with MIT licence and it's free to use. Not only it supports compilation for many platforms, as Android, Windows, Linux, Mac, etc., but the editor itself supports also multiple platforms, namely Windows, Linux and Mac.

Highlights of the engine Unlike most of others existing game engines and game authoring tools, Godot Engine offers both a visual editor and a scripting language. The user don't have then to program the whole game at hand in source code, like it would be for most game engines, and yet it doesn't sacrifice the flexibility of scripting in sake of "making it affordable for non developers" like most game makers do.

Nodes The game engine is composed with nodes. Everything is a node, except for standard libraries for scripting and stuff from the engine itself (script engine, display, input, etc.). However everything that is visible is a node.
A node is an object that has properties, functions and a custom script. Like in a XML tree, a node has a parent and can have children, all of them being nodes too. Nodes are the fundamental bricks of the game. And they have also the capability to inherit from others nodes the same way as in object oriented

programming. Some of them are native nodes coded in C++, and others are extending from the native ones.

Scenes The scene is a special node that be edited by the visual editor of Godot. A scene in Godot represents either literally a scene, where objects are shown and animated, either an actor of the game. The scene is a native node, yet it's not specific which node it must be. Usually a Node (the most abstract and common native node type) or a Node2D/Spatial (abstract node with spatial reference) is used as the node of a scene. And under this node are added others nodes like sprites and physic bodies, but it can be any kind of node, even a music player. What make scenes particular is that they can be used like objects in others scenes. Typically a sprite that moves and get animated would be a scene on its own and it would be added to another scene that represents the level of the game, with a terrain, monsters and others stuffs. And with scripting, the scene can be extended in its functionalities. That makes the scene one of the most versatile nodes of the engine, and the most used one by the user to create his game assets.

GDScript GDscript is a custom scripting language based on Python syntax. The language is very simple and can be learned in one day. It suffers however from being young and with only a small framework. Its selling point however is its deep integration with the node system and internal threads of the engine. It results in a very efficient and pragmatical language that gives good performances even on platforms with limited resources.
And this gives also the feature to chose how to structure the code. The script can be contained in a class file, or directly embedded in a scene, or even directly embedded in an instance of a scene. Having those three possibilities allows to very simply and efficiently write the code without having the framework in the way. The drawback, however, is the risk of inconsistency in the project and hard maintainability. It is very recommended, when starting a medium to large project, to set strict rules about code architecture. And breaking the rules should be avoided as much as possible but still allowed to make some tweaks that would very hard to implement otherwise.

1.5 About the author

I'm essentially a programmer with a few experience in 2d and 3d graphics. I have some personal experience in game programming, mostly in Java and Delphi, yet all my projects were for personal training and didn't reach any public status.

2 Game

2.1 In a nutshell

This demo is based on the popular Nintendo's[3] franchise Kirby[2] (Figure 1). It started in the nineties on Gameboy as a very simple and easy platform game for young kids, and then it evolved way beyond anyone's expectation thanks to

Figure 1: Original Kirby game

the huge potential in its game concept.
This demo is based on the very first game, without all the actual mechanics of the franchise. The reason is mostly because it's simple and yet it uses lot of fundamental features of a platform game.

2.2 Gameplay

The player moves in a 2d linear world, can jump and fall, and has one special power, which is to eat almost anything. This power is however very versatile. The player can create a vacuum in which every enemy and movable object get sucked and eaten. Then the player can ”spite” the result his digestion, which happen to be star shaped and that can hurt enemies it touches. The player can also suck air and inflate like a balloon, which gives him the ability to float freely (but slowly). In the second game of the franchise, a new mechanism of absorbing the power of an enemy is introduced, which extends even more the versatility of the player's power.

3 Project structure

3.1 file structure

In this demo, the files are meant to be structured this way:

- audio : all sounds and musics.
- res : all graphics.
- enemies : common assets for enemies.
- maps : levels of the game and their specific scripts.

Actually most assets and files were thrown at the root folder because of convenience. But when the project gets bigger, it will be needed to restructure files in a more meaningful folder structure.
Important files are:

- engine.cfg : the project definition file.
- main.scn : the very first scene automatically loaded. It acts like a scene loader and manages the HUD (heads-up display).
- global.gd : singleton utility script for scene loading and playing music.
- soundmanager.res : singleton node managing background musics and sounds that are meant to be played through scenes without interruption.
- game_data.gd : singleton script that manages game's common data, like player's life and high score.

3.2 Node structure

Figure 2 is the general structure of nodes running in the game. It is composed by singletons and layers.
Singletons are classes that provide data and functions through levels. They are not visible.
MapLayer is in fact not a layer but a Node for technical reasons. However it acts like a layer which contains the scene of the current level or cut-scene. Therefore its children are regularly removed and replaced by new ones. But the node itself is never removed or modified.
HudLayer contains common visual nodes that come in every level, like player's life bar, high score, etc.
ScreenEffectLayer is a node that can make the whole screen white by playing with its transparency. It is used for scene transition.

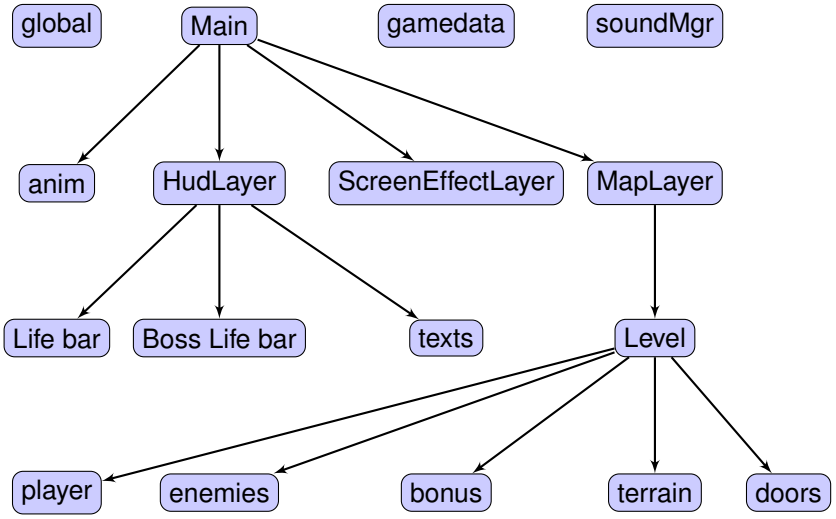

Figure 2: General structure of nodes

4 Actors

4.1 Player

The player is the only playable actor of this demo. It has a state, follow rules of a state machine.

Node structure The root of the player is a KinematicBody2D. It is composed of:

Sprite	The image of the actor
collisionShape	hitbox of the player for physics collisions, used by the root node
anim	Animation player of the actor, mostly for the sprite
Camera2D	The camera following the actor
breath_shoot	Point for spawning a bullet
cone_shoot	Point for spawning the vacuum cone
Area2D	Hitbox of the player for enemies and bonus
enemyActivator	Area for activating enemies
traverseTimer	Timer for traversing floors
sfx	Sound manager for the player

State machine The player has multiple states, as in Figure 3. Its initial state is *normal*.
When in cutscene mode, the player doesn't listen anymore to inputs nor it gets influenced by physics. The idea is that an animation player takes care of moving the actor.

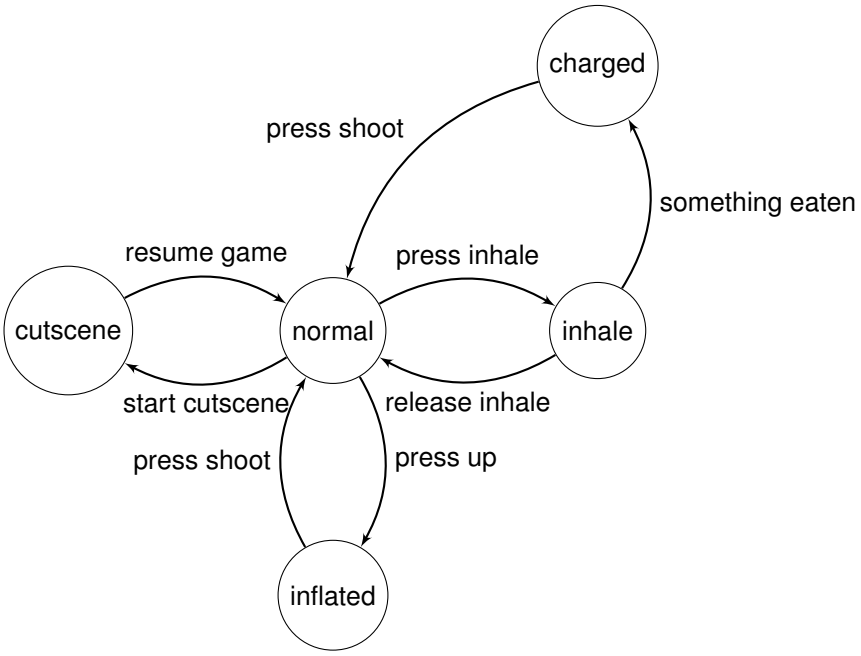

Figure 3: Player's state machine

4.2 Common enemy

Enemies are very identical. Only their sprite and their movement pattern change. Thus, instead of copying the same node structure into the node of each kind of enemy, one node is used by all enemies, with all sprites, and only their script and a parameter for the sprite change.
It has the advantage of being easier to maintain, but the disadvantages of limitations in what the enemy can do and its collision shape (limitation of the engine at the time when this document was written. Maybe a solution will be found in the future).

Node Structure The root of the enemy is a KinematicBody2D, implementing the Edible class. It is composed of:

sprite	The image of the actor
CollisionShape2D	Hitbox of the actor, used by the root node
AnimationPlayer	Animation player of the actor, mostly for the sprite
animTreePlayer	Manager of the different animations for all types of enemies
raycast_left	Collision sensor
raycast_right	Collision sensor
explodeSprite	Image of the explosion of the actor

Sensors The sensors of the common enemy, which are called Raycast2D in Godot, are a special kind of node that can check if their colliding with something else. Thanks to their position, as in Figure 4, the enemy can then know if he's about to collide with something and can make a decision, like turn the other way.
Note : This feature is not always used, depending on the movement pattern. Some kinds of enemy just don't need it.

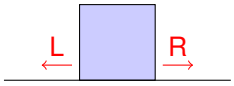

Figure 4: Sensors of the enemy, in red

Animation tree The sprites are managed by the Animation Tree Player, as in Figure 5. The node *transition* is the one deciding which kind of enemy to display, where the node *Eagle transition* defines more precisely in which state is the enemy for this specific kind.

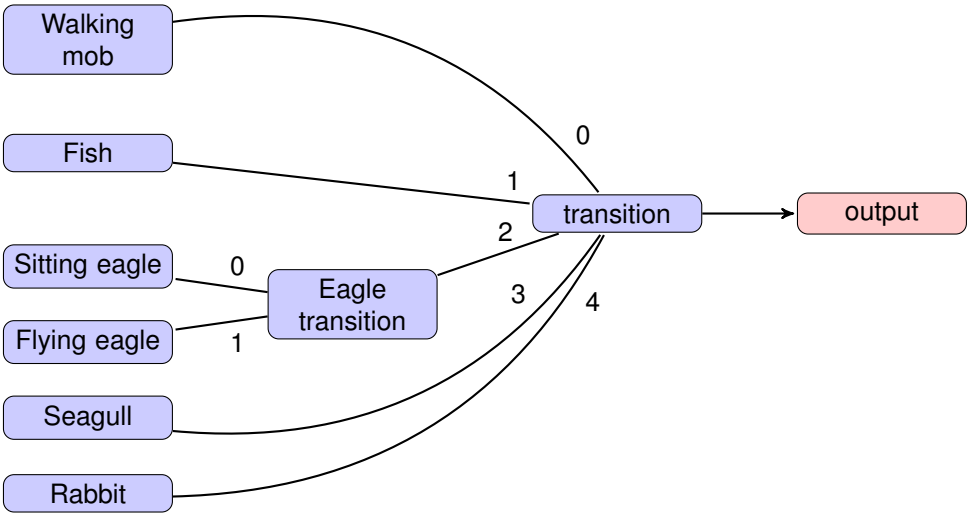

Figure 5: Animation tree of the common enemy

4.3 Boss

The boss is a special kind of enemy with the following characteristics:

- Cannot be eaten.
- Has multiple patterns.
- Can throw infinitely mobs or bombs, which the player must use to shoot a star to the boss.
- Has a life bar.
- Has its own stage (player cannot leave and camera is fixed).

Note : In version 1.1 bomb's visual asset is replaced by a rock. But its behavior didn't change.

Node structure In the context of this demo, this kind of boss had complex but static movement patterns, where he always go back to its original position. And so the movements are done with an animation player. An attempt was made by creating a sub-node that represents the collidable actor, which might not be the best solution. But it works. This sub-node is moved by the animation player. Its root is a KinematicBody2D, implementing the Edible class. It is composed of:

actor_boss	Sub-node containing the sprite and collision shape
sprite	The image of the actor
CollisionShape2D	Hitbox of the actor, used by the root node
explosion	The image of the explosion
bombSpawnPos	Position where to spawn a bomb
anim	Animation player

Movement pattern Bosses will always be implemented specifically for their needs because they have little in common. One can be just jumping here and there, where another one rushes to the player or go through different transformations. For the boss of this demo, the pattern is like in Figure 6.

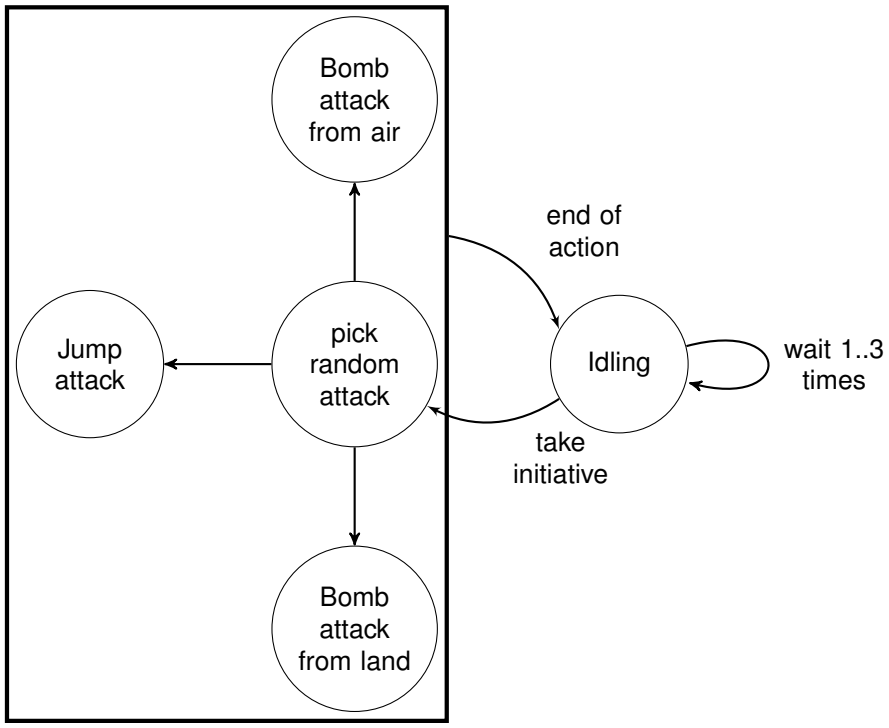

Figure 6: Boss' state machine

4.4 Bonus

The bonus is a very simple kind of node. It doesn't move but it can fall. When the player get in collision with it or eat it, it gives a bonus to the player, depending on its *Bonus type* parameter.

Node Structure Its root is a KinematicBody2D, implementing the Edible class. It is composed of:

sprite	The image of the actor
CollisionShape2D	Hitbox of the actor, used by the root node

Types of bonus From the original game, those are the know bonuses:

- Health : give a bit of health points to the player
- Full health : restore the whole health bar of the player
- 1up : give a continue to the player
- mik : special power that kills every visible enemy with a super amplified sound wave (not implemented in the demo)
- curry : give temporary invincibility (not implemented in the demo)
- lemon : give a temporary ability to fly and shoot breaths like stars. It's only for a specific boss stage. (not implemented in the demo)

5 Audio

5.1 Places

In this project, there is not only one place where audio is played. A singleton manages all background musics and scene transition sound effects. Actors can have their sound player, which also give a spatial position effect.

5.2 Sound Manager

The sound manager is a singleton that was created in order to keep the music playing while loading a new scene. It does have also a sound player for sound effects that must either be played while loading a new scene, or to play a sound that must not have a spatial effect. The sound manager doesn't have a position, and the sound is always played with spatial effect.

5.3 Types of audio player

In Godot, there are 2 players: one for music, which streams the file to avoid caching it completely, and one for sounds. The stream player can play very big files, but it doesn't have as many options or effects as the sound player.

5.4 Import audio files

It is possible to play directly a mp3 or an ogg file but specific parameters won't be saved. Godot can however import an audio file in its own format, which allows special parameters like looping in a specific time range of the sound or music. An imported file can also have some sound effects, like echo, that will be calculated at runtime. This allows to keep the original sound file small.

6 Problematic

6.1 Collisions&physics

One important aspect of the game is the collision management with physics. Godot Engine integrates its own physics engine, which offers all features needed for a 2d or 3d platform game. This project uses the physics engine to manage objects moving and colliding properly against each other.

For 2d collision, the engine offers 3 kind of nodes:

- StaticBody2D : for a body that won't move.
- RigidBody2D : for a body that follows standard physics (falling, bouncing, sliding) and can be controlled by applying force to it.
- KinematicBody2D : for a body that is programmatically controlled but that can use the physics engine to calculate the next move. It is the easiest to manipulate for active actors that move on their own.

In this project, only the KinematicBody2D, for actors, and the StaticBody2D, for the terrain, are used.

A bit of theory The way Godot manages collisions and sliding with kinematic bodies is, like in Figure 7, done in 2 steps:

1. Move until a collision is detected
2. Slide along the detected body

And Godot offers those two functions. Given an actor who moves with a velocity *V* from a position *P*, the function **move** will bring eventually the actor to position P_1 before it returns a collision detection. After that, no matter if how much the function **move** is called with the same velocity, the actor won't move. Then enters the function **slide**, which will calculate a motion *M* that goes along the surface of the collider. And then this motion *M* will replace the velocity *V*. And then the actor finished its movement to reach position P_2.

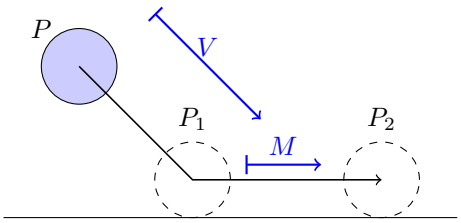

Figure 7: Collision and sliding

Code When an actor must be able to collide with its surrounding, some scripting code is required, especially because the engine is unaware of the kind of game the project is part of. For a platform game, the logic is like in Figure 8.

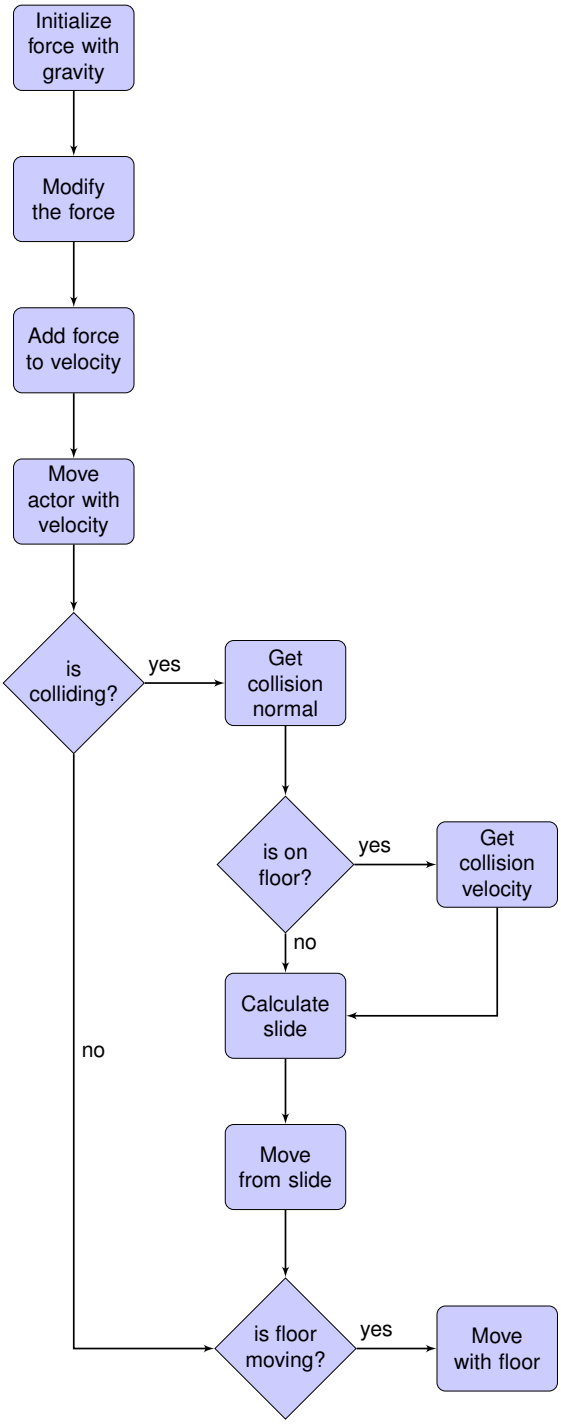

Figure 8: Collision flowchart

6.2 Traversable platforms

One issue that comes with using physics in a 2d platform game is the loss of features from older 2d games without physics. One of the most useful ones is the ability to go through a wall or a floor in a certain direction but not the other direction. Typically that would be a floor that the player can jump through from below but not from above.
As in Figure 9, actor 1 lands like usual on the box but actor 2 can jump through the box. And once it's above the box, actor 2 cannot go in any more.

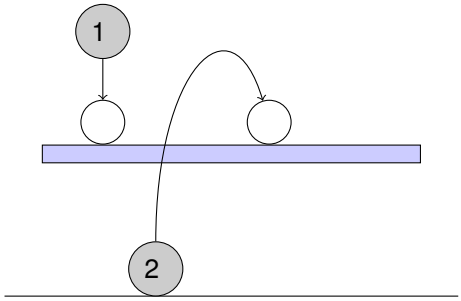

Figure 9: Traversable floor

Problem By default it's impossible to do such thing in Godot. The actor would never be able at any moment to be in the box or any other object. Such state is not allowed by the engine. It is possible to disable the collision feature of the actor until it got outside of the box, but during this time it would not collide with anything at all. No bonus, no enemy, no bullet, nothing. It would be some kind of unwanted invulnerability that happens every time the actor jump, which is not realistic for a platform game.

Solution However, with the collision layers feature, it became possible to disable collision detection of the actor with all objects of a specific layer, just by removing the actor from this layer for the time it needs to go through, usually when its vertical velocity is going up.

6.3 Vacuum cone & edible objects

When the player starts to suck, a vacuum cone is created. Some kind of actors that are within this cone will be sucked to the player's mouth and be eaten. Those actors can be as well enemies, bombs, bonuses or mere blocks. In this demo, they are all extending the abstract class Edible. In a bigger project, it would be better to use duck typing instead and give then a property edible.

Process When the player presses the inhale button, the process goes as in Figure 10.

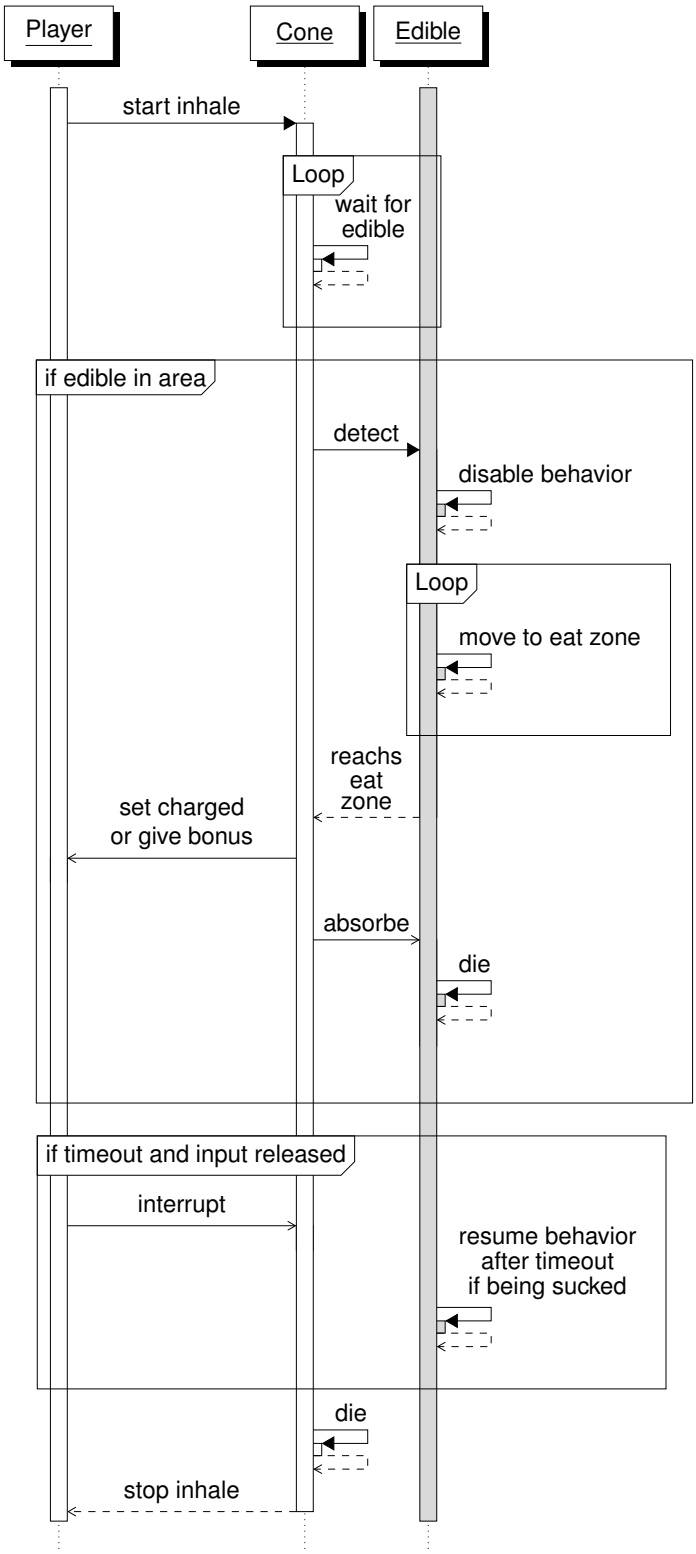

Figure 10: Vacuum process

Node Structure Its root is an Area2D. It is composed of:

particles	Particle emitter
ParticleAttractor2D	Attractor of the particles
CollisionPolygon2D	Vacuum area where Edible objects start to get sucked
eatPosition	Position used by Edible objects to know where to be sucked
eatArea	Area where sucked objects are considered to be eaten
sfx	Sound manager

Visual effect The cone is actually a particular case of particle emitter. Instead of emitting from a point to a direction, it emits in a region to the direction of a single point (see Figure 11).

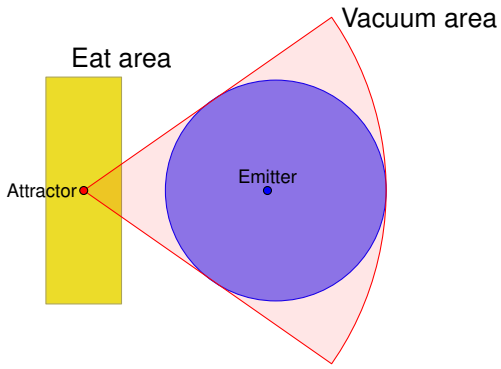

Figure 11: Vacuum cone

6.4 Scene loading

Scene loading consist of loading the asset and replace the node of the current scene with this one. The most basic example would be to replace the child of the */root* node, which then replaces everything visible. In this demo however, there's an HUD and others nodes that need to stay, mosty in order to not add a copy of them in each scene. A special node exists then to contain the currently loaded scene.

Cutscenes and playable scenes There are two kinds of scenes. The cutscenes and the playable scenes. The technical difference is that HUD don't show up in cutscenes. But nothing prevents to make a cutscene playable by adding to it a playable character and make the HUD visible again. However the way to load the scene differ from a cutscene to a playable scene. For a cutscene, it's only needed to replace the current scene with the cutscene to load. And for a playable scene, the player, which must have an instance in every playable scene, have to be moved to a certain position determined by a parameter.

Doors The parameter defining where the player must be exists because of a need to enter a scene from multiple entry points. A good example is a town with houses. The player start, for instance, at the entrance of the town. And he

can visit houses, which are smaller but distinct scenes. That means the player can travel through scenes and go back to the main scene, the town. And in sake of coherence, the player leaving a house should appear at the front of the house instead of the entrance of the town. This problem is solved by passed a parameter to where the player come from.
In this demo, doors act like areas where the player travel to another scene. And position nodes with a name are used to determine the new position of the player. The door node has two parameters: the name of the scene to load and the name of the position node to use.

Groups The player node doesn't travel through scenes. It is destroyed as the same time as its corresponding scene is destroyed. The player of the new scene is then used to play, making the illusion the player travelled to another scene.
The path to the player node is however unknown, as it may not be consistent through scenes. To solve this problem, groups are used. A group is a tag that nodes can have. The player has the tag "Player". It is then possible to retrieve every node of a group. In the case of the player, there is only one node in the group "Player", which makes it easy to find. There is also a group called "entry_point" which contains all nodes where the player can appear, in order to make easy to find the node by the name given in parameter from the door.

2 players bug There is still a trick with the way scenes are loaded. The code is very simple:

1. Remove the old scene from the container.
2. Call *queue_free()* of the old scene to destroy it.
3. Load the new scene.
4. Insert the instance of the new scene in the container.
5. Search the player in the group "Player".
6. Move the found player.

But in point 5, the player of the old scene is still returned, in addition of the player of the new scene. The reason is that the old scene is marked to be freed by the garbage collector but is still existing. And since the array of nodes returned by the search group function is in an unpredictable order, there is no easy way to know which player to move is the good one (from the new scene). This can be easily solved by moving indiscriminately all players, because the old player will be disposed soon anyway and doesn't matter anymore.

References

[1] Godot Game Engine *Okam Studio* 2014. http://www.godotengine.org/wp/

[2] Kirby© is a property of Nintendo[3].

[3] Nintendo Co., Ltd.©