# Entity

## Attributs d’une entité

Ces attributs devraient être configurer dans le constructeur

acc = 800;

maxVelX = 300;

drag = 400;

jumpingStrength = 500;

airdrag = 200;

gravity = 100;

\*Les valeurs utilisées sont uniquement fournies en exemple

Bien qu’optionnels, ils affectent le résultat de certaines méthodes du parent et devraient être utilisés par les actions de l’entité (voir section Logique)

## Animations

La première chose à faire avant de pouvoir utiliser des animations sur une entité est de les créée et de les envoyer à l’Animator,

Pour commencer il faut créer une [struct] Spritesheet à partir d’une texture et de quelques informations de base permettant de calculer la taille des sprites

sf::Texture\* texture = new Texture();

texture->loadFromFile("Assets/SpriteSheet/Hunter.png");

int nbRows = 2;

int nbColums = 16;

Spritesheet spritesheet = { texture, nbRows, nbColums };

On initie ensuite les vectors utilité pour construire les animations

vector<Coord> indexes;

vector<int> showTimes;

Pousse pour chaque frame d’animation sa position ([struct] Coord) et le temps d’affichage en MS

indexes.push\_back({ indexColonne, indexRangee });

showTimes.push\_back( tempsAffichageEnMS );

Ne reste qu’à créer l’Animation et l’envoyé en référence à l’Animator. La valeur booléenne optionnelle Loop l’animation, la String sert d’index.

animator.AddAnimation(new Animation(spritesheet, indexes, showTimes, true), "Stand");

Resettez vos vectors avant de créer une nouvelle animation

indexes = vector<Coord>();

showTimes = vector<int>();

## Logique

La logique d’une entité repose sur un système basique de State Machine. L’Action currentAction contient un Child d’Action et est Updaté à chaque frame. Des actions génériques pour les déplacement d’un personnage joueur sont incluses en exemple.

Pour commencer, settons l’action actuelle à l’action par default

MaClasse::MaClasse() : Entity()

{

CurrentAction = new Standing(this);

}

Ensuite une methode override appelée ChangeAction doit être ajouter à votre classe. Cette méthode est appelée par le parent quand un changement d’action se produit. Je conseille de Caster l’index reçu en Enum. Cette méthode devrait ressembler à quelque chose du genre :

#include « monEnum.h »

void MaClasse::ChangeAction(int enumIndex)

{

switch ((MonEnum)enumIndex)

{

case STAND:

animator.ChangeAnimation("Stand");

delete CurrentAction;

CurrentAction = new Standing(this);

break;

case WALK:

animator.ChangeAnimation("Walk");

delete CurrentAction;

CurrentAction = new Walking(this);

break;

case JUMP:

animator.ChangeAnimation("Jump");

delete CurrentAction;

CurrentAction = new HunterJump(this);

break;

case CROUNCH:

animator.ChangeAnimation("Crounch");

delete CurrentAction;

CurrentAction = new Crounching(this);

break;

}

}

Plutôt simple. Maintenant, pour créer une action personnalisée, commencez par créer une nouvelle classe qui hérite d’Action. Le corps (.cpp) de cette classe doit inclure « Entity.h », le constructeur doit recevoir Entity\* en paramètre et le passer à son parent.

#pragma once

#include “MonAction.h”

#include “Entity.h”

#include “TimeManager.h”

#include “MonEnum.h”

MonAction::MonAction(Entity\* e) : Action(e)`

{

}

La fonction override « Update() » est appelée une fois par frame. Cette méthode retourne un integer si un changement d’animation doit se produire. Encore une fois je conseille de caster un Enum en integer avant de renvoyer l’information. Une action peut affecter le parent gardé en référence (parent->)

int MonAction ::Update()

{

parent->velY += (500 + parent->drag) \* TimeManager ::DeltaTime;

if (parent->velY > 2000)

return (int) MonEnum::EXPLODE;

return -1;

}