# Entity

La classe **Entite** contient le necessaire pour une unité animée, affecté par la physique, avec une boite de collision et une IA.

## 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 IA)

## Animations

La première chose à faire avant de pouvoir utiliser des animations sur une entité est de les créer

Commençons par récupérer la texture et configurer la spritesheet

sf::Texture\* texture = &AssetManager::getTexture("axe");

int numberOfRow = 2;

int numberOfColumn = 8;

Spritesheet spritesheet = { texture, numberOfRow, numberOfColumn };

On crée ensuite l’animation a partir de la spritesheet

Animation \* anim = new Animation(spritesheet);

On y ajoute nos frames ({index X, index Y}, temps d’affichage). Le temps d’affichage peut être setter à -1 pour arrêter l’animation à ce point

anim->addFrame({ 0, 0 }, 31);

anim->addFrame({ 1, 0 }, 31);

anim->addFrame({ 2, 0 }, 31);

anim->addFrame({ 3, 0 }, 31);

anim->addFrame({ 4, 0 }, 31);

anim->addFrame({ 5, 0 }, 31);

Ne reste qu’à nommer l’animation et a l’ajouter à la collection en passant par la méthode addAnimation()

addAnimation(anim, "Roll");

## IA

L’intelligence d’une entité repose sur un système basique de State Machine. L’Action « currentAction » contient un Child d’Action et Update() 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 qui override 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;

}

}

Maintenant, pour créer une action personnalisée, commencez par créer une nouvelle classe qui hérite d’ActionEntity. 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) : ActionEntity(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 \* TimeManager ::DeltaTime;

if (parent->velY > 2000)

return (int) MonEnum::EXPLODE;

return -1;

}