HOT JUMP - Memoria

Descripción del Juego

Hot Jump es un juego de plataformas con movimiento 2d, a pesar de su vista en 3d. En este juego el usuario manejará al protagonista que es una pequeña ascua en llamas. Su misión es conservar esas llamas mientras cruzas todo un escenario lleno de obstáculos que tendrás que ir esquivando para no apagar tu fuego. Estos obstáculos serán cascadas de agua, tramos con viento (que se ven gracias a la pequeña neblina de color lila que se mueve indicando la dirección del viento y su posición) y torbellinos, además de procurar no precipitarse al vacio.
El final del nivel viene señalizado por una bandera roja y al tocar esta bandera, se dispararán unos fuegos artificiales para destacar la victoria y tras esto se iniciará el siguiente nivel.
Este juego solo dispone de 2 niveles así que al terminar el segundo, te llevara al primero, y el primero al segundo. Así se sucederá continuamente.
En caso de encontrarte con algún obstáculo, tu fuego se apagará, por lo que el nivel correspondiente se reiniciara.

Diagrama de clases

Ver en la última página de este documento.

Explicación de ecuaciones

- WorldManager
  - Movimiento y salto del jugador: `player->setLinearVelocity(player->getLinearVelocity() + Vector3(0,30,0));` A la velocidad lineal que ya tenia el jugador se le suma cierta cantidad para moverlo cada vez que se le da a una tecla, hasta un máximo. Permite que cuanto mas tiempo estes apretando, más largo será el movimiento que haga.
  - `gPhysics->createMaterial(50.0, 100.0, 1.0));` Aplicado al escenario. Dependiendo del tipo de suelo que sea tendrá unos parámetros u otros. Este ejemplo es del suelo de hielo que permite mas deslizamiento. En cambio para el resto, todos los parámetros están a 0.
  - Todos los elementos tienen el damping angular a 0 para no rotar, igual que la velocidad angular. En el jugador el damping lineal tampoco es muy alto para que se frene rápido y de la sensación de que para cuando el jugador para de pulsar, de la sensación de que ha parado. En cambio el damping de las plataformas flotantes es de 0.999 para que se puedan mover con mas libertad
- ParticleSystem
  - `(*p)->changeSize((*p)->getTimeAlive() * t, (*p)->getTransform(), (*p)->getColor());` Cálculo del nuevo tamaño de las partículas de fuego. Cuanto más tiempo llevan vivas, es decir, cuanto mas arriba están, mas pequeñas se hacen.
  - `(*p)->getPosition().y >= pos.y + 10 && ((*p)->getPosition().x >= pos.x-5 && (*p)->getPosition().x <= pos.x+5).` Comparo la posición de las partículas de agua con respecto a la del jugador para poder apagar el fuego. Si esta

cerca de la posición de la particula, entonces se apaga el fuego. Se comprueba asi porque las partículas de agua no son solidos rigidos.

- CircleGenerator
  - `auto v = Vector3(random*cos(angle * PI / 180.0), 0, random * sin(angle * PI / 180.0)); angle += 360.0 / nParticles;` Cálculo de una nueva velocidad de las partículas para que cada una vaya hacia una dirección formando un círculo perfecto entre todas
- GravityForceGenerator
  - `p->addForce(_gravity * p->getMass());` Cálculo de la gravedad aplicando el vector _gravity que ejerce una fuerza hacia abajo. Valor: {0, -10, 0}
- GaussianSolidRigidGen
  - `auto posi = Vector3(desTip_pos.x * d(gnd), desTip_pos.y * d(gnd), desTip_pos.z * d(gnd));auto v = Vector3(desTip_vel.x * d(gnd), desTip_vel.y * d(gnd), desTip_vel.z * d(gnd));` Cálculo de una nueva posición y velocidad de acuerdo a una distribución normal. Las desviaciones serán unos parámetros que dependerán del tipo de sistema que se quiera simulas (amplio para una cascada y bajo para una niebla como ejemplo de desviación de la velocidad)
- `Particle`
  - En el update de la particula se actualiza su posición proporcinalmente al tiempo y ya velocidad que lleva. Esa velocidad dependerá de las fuerzas externas a las que este esxpuesta, de su masa y el tiempo.

```
pos.p += vel * t;
Vector3 totalAcceleration = a;
totalAcceleration += force * inverse_mass;
// Update linear velocity
vel += totalAcceleration* t;
// Impose drag (damping)
vel *= powf(damping, t);
clearForce();
```

- BuoyancyForceGenerator

Para la flotación tenemos que tener en cuenta la altura del liquido t de la particula, además de la gravedad que afecta a esta particula la densidad y el volumen.

```
if (h - h0 > height * 0.5)
    immersed = 0.0;

else if (h0 - h > height * 0.5)
    immersed = 1.0;

else
    immersed = (h0 - h) / height + 0.5;

f.y = density * volume * immersed * gravity;
```

- SpringForceGenerator/AnchoredSpringFG
  - Se calcula la fuerza que ejerce el muelle según su constante elástica k y la ley de Hooke. Esta fuerza es proporcional a la constante y la distancia que se estira o contrae desde su longitud en reposo. `Vector3 f = other_->getPosition() - p->getGlobalPose().p; const float lenght = f.normalize(); const float delta_x = lenght - resting_lenght_; f *= delta_x * k_;`
- WindGenerator/WhirlwindGenerator
  - `if (f.x < (vel.x-1) && p->isFire()) over = true;` Si hay una particula fuera del rando determinado del viento, este la apaga.
  - `return (pos.x < maxPositive.x && pos.x > maxNegative.x && pos.y < maxPositive.y && pos.y > maxNegative.y && pos.z < maxPositive.z && pos.z > maxNegative.z);` Comprobación de si algo esta dentro de ese rango predeterminado
  -
```
float drag_coef = v.normalize();
Vector3 dragF;
drag_coef = _k1 * drag_coef + _k2 * drag_coef * drag_coef;
dragF = -v * drag_coef;
```
Calculo de la fuerza del viento a aplicar a una partícula teniendo en cuanta 2 constantes. K2 sera siempre 0 en nuestro caso ya que no

queremos un viento muy fuerte, por lo que no hace falta. La fuerza se aplica en sentido contrario.

Efectos

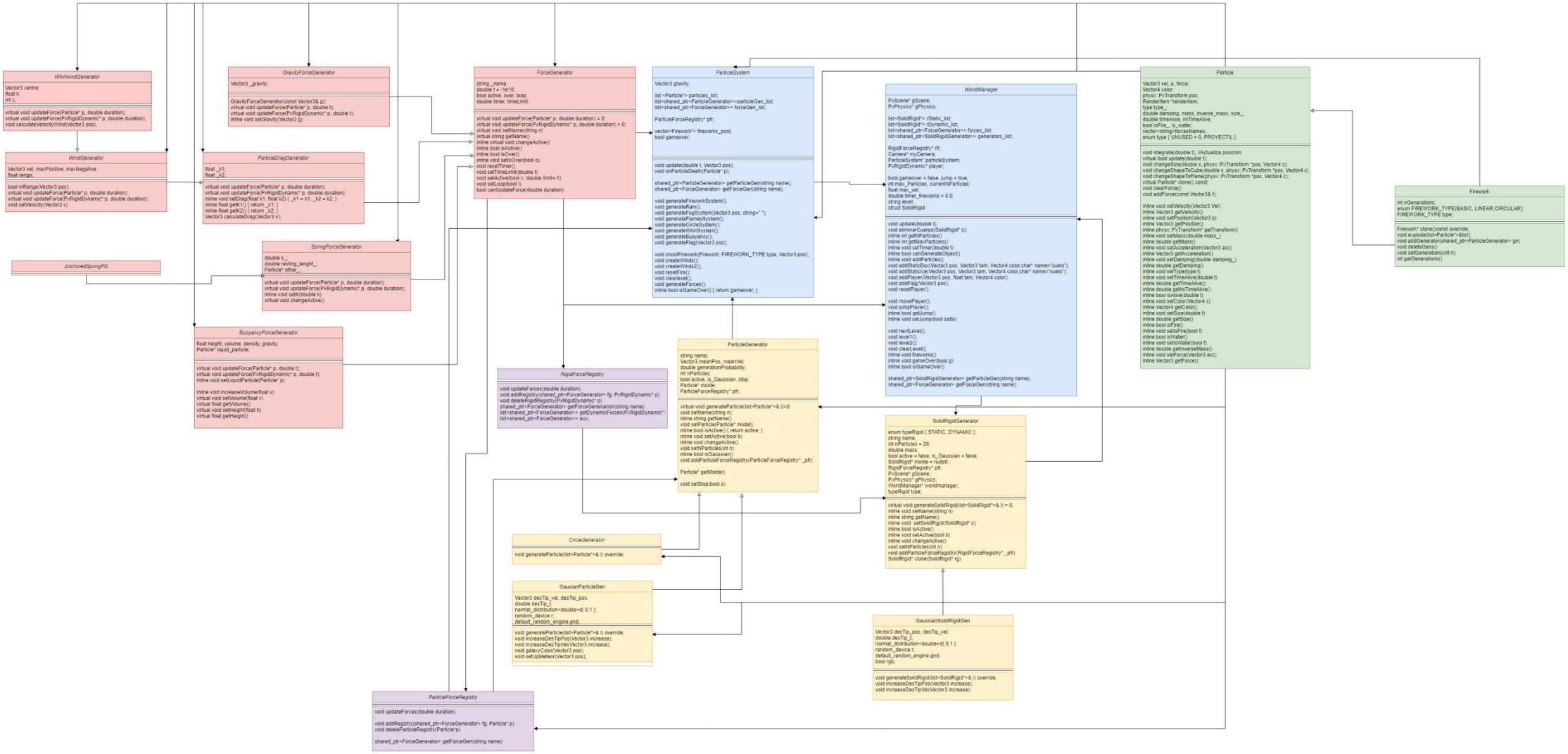
- Los sistemas de partículas se pueden activar y desactivar. Pero su particula molde no desaparece. Para dar el efecto de que desaparece se dejan de generar partículas y la molde se manda lejos donde no se ve. `molde->setPosition(Vector3(pos.x, pos.y + 4000, pos.z));`
- Para simular el efecto de fuego, a esas partículas se les ha aplicado una gravedad positiva y un dampling algo mas inferior. Ademas solo se mueve la particula molde con el jugador, por lo que le da mas sensación de realismo
- Para crear plataformas de hielo se ha cambiado los rozamientos estáticos y dinámicos del material para que resulte un poco mas resbaladizo.
- Los sistemas AnchoredSpringFG afectados por un viento que cada 2 segundos se activa y desactiva da una sensación de bandera
- La cámara sigue al jugador. Se cambia el Eye para que el jugador se mantenga siempre en el centro de la pantalla
- Las partículas de niebla no son solidos rigidos para que no empujen al jugador. No tienen gravedad y se mueven lentamente gracias a la fuerza del vient, asi que da ese aspecto de niebla que se mueve ligeramete
- Las plataformas flotantes están sobre bloques de agua
- La cascada de agua esta hecha de partículas que no son solidas rigidas para que no movieran al jugador. Caen solo por la fuerza de la gravedad.

Manual de usuario

Ver en el README del proyecto

Efectos/Experimentos extra

- Cambio del fondo y la iluminación para que se vea con colores más vivos
- Utilizacion de la clase ColorHSV para lograr degradados y cambios en saturación y brillo que con rgb no se puede conseguir
```
hsv col = { 24.0f, 0.9f, 0.55f };
rgb rgb = hsv2rgb(col); Vector4 colo = { rgb.r, rgb.g, rgb.b, 1.0 };
```
- Bloqueo en las rotaciones y movimiento lineal en el eje Z para darle más efecto 2d
```
Player->setRigidDynamicLockFlags(PxRigidDynamicLockFlag::eLOCK_LINEAR_Z);
PxRigidDynamicLockFlag::eLOCK_ANGULAR_X |
PxRigidDynamicLockFlag::eLOCK_ANGULAR_Y |
PxRigidDynamicLockFlag::eLOCK_ANGULAR_Z |
PxRigidDynamicLockFlag::eLOCK_LINEAR_X
```


WhirlwindGenerator
Vector3 centre;
float k;
int c;
virtual void updateForce(Particle* p, double duration);
virtual void updateForce(PxRigidDynamic* p, double duration);
void calculateVelocityWind(Vector3 pos);
GravityForceGenerator
Vector3 _gravity;
GravityForceGenerator(const Vector3& g);
virtual void updateForce(Particle* p, double t);
virtual void updateForce(PxRigidDynamic* p, double t);
inline void setGravity(Vector3 g)
ForceGenerator
string _name;
double t = 1e10;
bool active, over, loop;
double timer, timeLimit;
virtual void updateForce(Particle* p, double duration) = 0;
virtual void updateForce(PxRigidDynamic* p, double duration) = 0;
virtual void setName(string n)
virtual string getName()
inline virtual void changeActive()
inline bool isActive()
inline bool isOver()
inline void setIsOver(bool o)
void resetTimer()
void setTimeLimit(double t)
void setActive(bool x, double limit=-1)
void setLoop(bool l)
bool canUpdateForce(double duration)
ParticleSystem
Vector3 gravity;
list <Particle*> particles_list;
list<shared_ptr<ParticleGenerator>>particleGen_list;
list<shared_ptr<ForceGenerator>> forceGen_list;
ParticleForceRegistry* pfr;
vector<Firework*> fireworks_pool;
bool gameover;
void update(double t, Vector3 pos);
void onParticleDeath(Particle* p);
shared_ptr<ParticleGenerator> getParticleGen(string name);
shared_ptr<ForceGenerator> getForceGen(string name);
void generateFireworkSystem();
void generateRain();
void generateFogSystem(Vector3 pos, string=" ");
void generateFlamesSystem();
void generateCircleSystem();
void generateWhirlSystem();
void generateBuoyancy();
void generateFlag(Vector3 pos);
void shootFirework(Firework::FIREWORK_TYPE type, Vector3 pos);
void createWinds();
void createWinds2();
void resetFire();
void clearLevel();
void generateForces();
inline bool isGameOver() { return gameover; }
WorldManager
PxScene* gScene;
PxPhysics* gPhysics;
list<SolidRigid*> rStatic_list;
list<SolidRigid*> rDynamic_list;
list<shared_ptr<ForceGenerator>> forces_list;
list<shared_ptr<SolidRigidGenerator>> generators_list;
RigidForceRegistry* rfr;
Camera* myCamera;
ParticleSystem* particleSystem;
PxRigidDynamic* player;
bool gameover = false, jump = true;
int max_Particles, currentNParticles;
float max_vel;
double timer_fireworks = 0.0;
string level;
struct SolidRigid
void update(double t);
void eliminarCuerpo(SolidRigid* s);
inline int getNParticles()
inline int getMaxParticles()
inline void setTime(double t)
inline bool canGenerateObject()
inline void addParticles()
void addStaticBox(Vector3 pos, Vector3 tam, Vector4 color,char* name="suelo");
void addStaticIce(Vector3 pos, Vector3 tam, Vector4 color,char* name="suelo");
void addPlayer(Vector3 pos, float tam, Vector4 color);
void addFlag(Vector3 pos);
void resetPlayer();
void movePlayer();
void jumpPlayer();
inline bool getJump()
inline void setJump(bool salto)
void nextLevel();
void level1();
void level2();
void clearLevel();
inline void fireworks()
inline void gameOver(bool g)
inline bool isGameOver()
shared_ptr<SolidRigidGenerator> getParticleGen(string name);
shared_ptr<ForceGenerator> getForceGen(string name);
Particle
Vector3 vel, a, force;
Vector4 color;
physx::PxTransform pos;
RenderItem *renderItem;
type type_;
double damping, mass, inverse_mass, size_;
double timeAlive, imTimeAlive;
bool isFire_, is_water;
vector<string>forcesNames;
enum type { UNUSED = 0, PROYECTIL };
void integrate(double t); //Actualiza posicion
virtual bool update(double t);
void changeSize(double s, physx::PxTransform *pos, Vector4 c);
void changeShapeToCube(double s, physx::PxTransform *pos, Vector4 c)
void changeShapeToPlane(physx::PxTransform *pos, Vector4 c);
virtual Particle* clone() const;
void clearForce();
void addForce(const Vector3& f);
inline void setVelocity(Vector3 Vel)
inline Vector3 getVelocity()
inline void setPosition(Vector3 p)
inline Vector3 getPosition()
inline physx::PxTransform* getTranform()
inline void setMass(double mass_)
inline double getMass()
inline void setAcceleration(Vector3 acc)
inline Vector3 getAcceleration()
inline void setDamping(double damping_)
inline double getDamping()
inline void setType(type t)
inline void setTimeAlive(double t)
inline double getTimeAlive()
inline double getImTimeAlive()
inline bool isAlive(double t)
inline void setColor(Vector4 c)
inline Vector4 getColor()
inline void setSize(double t)
inline double getSize()
inline bool isFire()
inline void setIsFire(bool f)
inline bool isWater()
inline void setIsWater(bool f)
inline double getInverseMass()
inline void setForce(Vector3 acc)
inline Vector3 getForce()
Firework
int nGenerations;
enum FIREWORK_TYPE{BASIC, LINEAR,CIRCULAR};
FIREWORK_TYPE type;
Firework* clone()const override;
void explode(list<Particle*>&list);
void addGenerator(shared_ptr<ParticleGenerator> gn)
void deleteGens()
void setGenerations(int n)
int getGenerations()
WindGenerator
Vector3 vel, maxPositive, maxNegative;
float rango;
bool inRange(Vector3 pos);
virtual void updateForce(Particle* p, double duration);
virtual void updateForce(PxRigidDynamic* p, double duration);
void setVelocity(Vector3 v)
ParticleDragGenerator
float _k1;
float _k2;
virtual void updateForce(Particle* p, double duration);
virtual void updateForce(PxRigidDynamic* p, double duration);
inline void setDrag(float k1, float k2) { _k1 = k1; _k2 = k2; }
inline float getK1() { return _k1; }
inline float getK2() { return _k2; }
Vector3 calculateDrag(Vector3 v);
SpringForceGenerator
double k_;
double resting_lenght_;
Particle* other_;
virtual void updateForce(Particle* p, double duration);
virtual void updateForce(PxRigidDynamic* p, double duration);
inline void setK(double k)
virtual void changeActive()
AnchoredSpringFG
BuoyancyForceGenerator
float height, volume, density, gravity;
Particle* liquid_particle;
virtual void updateForce(Particle* p, double t);
virtual void updateForce(PxRigidDynamic* p, double t);
inline void setLiquidParticle(Particle* p)
inline void increaseVolume(float v)
virtual void setVolume(float v)
virtual float getVolume()
virtual void setHeight(float h)
virtual float getHeight()
RigidForceRegistry
void updateForces(double duration)
void addRegistry(shared_ptr<ForceGenerator> fg, PxRigidDynamic* p)
void deleteRigidRegistry(PxRigidDynamic* p)
shared_ptr<ForceGenerator> getForceGeneranion(string name)
list<shared_ptr<ForceGenerator>> getDynamicForces(PxRigidDynamic*)
list<shared_ptr<ForceGenerator>> aux;
ParticleGenerator
string name;
Vector3 meanPos, meanVel;
double generationProbability;
int nParticles;
bool active, is_Gaussian, stop;
Particle* molde;
ParticleForceRegistry* pfr;
virtual void generateParticle(list<Particle*>& l)=0;
void setName(string n);
inline string getName()
void setParticle(Particle* model);
inline bool isActive() { return active; }
inline void setActive(bool b)
inline void changeActive()
void setNParticles(int n)
inline bool isGaussian()
void addParticleForceRegistry(ParticleForceRegistry* _pfr)
Particle* getMolde()
void setStop(bool s)
SolidRigidGenerator
enum typeRigid { STATIC, DYNAMIC };
string name;
int nParticles = 20;
double mass;
bool active = false, is_Gaussian = false;
SolidRigid* molde = nullptr;
RigidForceRegistry* pfr;
PxScene* gScene;
PxPhysics* gPhysics;
WorldManager* worldmanager;
typeRigid type;
virtual void generateSolidRigid(list<SolidRigid*>& l) = 0;
inline void setName(string n)
inline string getName()
inline void setSolidRigid(SolidRigid* s)
inline bool isActive()
inline void setActive(bool b)
inline void changeActive()
void setNParticles(int n)
void addParticleForceRegistry(RigidForceRegistry* _pfr)
SolidRigid* clone(SolidRigid* rg)
CircleGenerator
void generateParticle(list<Particle*>& l) override;
GausianParticleGen
Vector3 desTip_vel, desTip_pos;
double desTip_t;
normal_distribution<double>d(0,1);
random_device r;
default_random_engine gnd;
void generateParticle(list<Particle*>& l) override;
void increaseDesTipPos(Vector3 increase);
void increaseDesTipVel(Vector3 increase);
void galaxyColor(Vector3 pos);
void setUpMeteor(Vector3 pos);
GaussianSolidRigidGen
Vector3 desTip_pos, desTip_vel;
double desTip_t;
normal_distribution<double>d(0,1);
random_device r;
default_random_engine gnd;
bool rgb;
void generateSolidRigid(list<SolidRigid*>& l) override;
void increaseDesTipPos(Vector3 increase);
void increaseDesTipVel(Vector3 increase);
ParticleForceRegistry
void updateForces(double duration)
void addRegistry(shared_ptr<ForceGenerator> fg, Particle* p)
void deleteParticleRegistry(Particle*p)
shared_ptr<ForceGenerator> getForceGen(string name)