ПРИЛОЖЕНИЕ А

(*обязательное*)

Код программы. Модуль pworld

#ifndef CYCLONE\_PWORLD\_H

#define CYCLONE\_PWORLD\_H

#include "pfgen.h"

#include "plinks.h"

namespace cyclone {

class ParticleWorld

{

public:

typedef std::vector<Particle\*> Particles;

typedef std::vector<ParticleContactGenerator\*> ContactGenerators;

protected:

Particles particles;

bool calculateIterations;

ParticleForceRegistry registry;

ParticleContactResolver resolver;

ContactGenerators contactGenerators;

ParticleContact \*contacts;

unsigned maxContacts;

public:

ParticleWorld(unsigned maxContacts, unsigned iterations=0);

~ParticleWorld();

unsigned generateContacts();

void integrate(real duration);

void runPhysics(real duration);

void startFrame();

Particles& getParticles();

ContactGenerators& getContactGenerators();

ParticleForceRegistry& getForceRegistry();

};

class GroundContacts : public cyclone::ParticleContactGenerator

{

cyclone::ParticleWorld::Particles \*particles;

public:

void init(cyclone::ParticleWorld::Particles \*particles);

virtual unsigned addContact(cyclone::ParticleContact \*contact,

unsigned limit) const;

};

} // namespace cyclone

#endif // CYCLONE\_PWORLD\_H

#include <cstddef>

#include <cyclone/pworld.h>

using namespace cyclone;

ParticleWorld::ParticleWorld(unsigned maxContacts, unsigned iterations)

:

resolver(iterations),

maxContacts(maxContacts)

{

contacts = new ParticleContact[maxContacts];

calculateIterations = (iterations == 0);

}

ParticleWorld::~ParticleWorld()

{

delete[] contacts;

}

void ParticleWorld::startFrame()

{

for (Particles::iterator p = particles.begin();

p != particles.end();

p++)

{

// Remove all forces from the accumulator

(\*p)->clearAccumulator();

}

}

unsigned ParticleWorld::generateContacts()

{

unsigned limit = maxContacts;

ParticleContact \*nextContact = contacts;

for (ContactGenerators::iterator g = contactGenerators.begin();

g != contactGenerators.end();

g++)

{

unsigned used =(\*g)->addContact(nextContact, limit);

limit -= used;

nextContact += used;

// We've run out of contacts to fill. This means we're missing

// contacts.

if (limit <= 0) break;

}

// Return the number of contacts used.

return maxContacts - limit;

}

void ParticleWorld::integrate(real duration)

{

for (Particles::iterator p = particles.begin();

p != particles.end();

p++)

{

// Remove all forces from the accumulator

(\*p)->integrate(duration);

}

}

void ParticleWorld::runPhysics(real duration)

{

// First apply the force generators

registry.updateForces(duration);

// Then integrate the objects

integrate(duration);

// Generate contacts

unsigned usedContacts = generateContacts();

// And process them

if (usedContacts)

{

if (calculateIterations) resolver.setIterations(usedContacts \* 2);

resolver.resolveContacts(contacts, usedContacts, duration);

}

}

ParticleWorld::Particles& ParticleWorld::getParticles()

{

return particles;

}

ParticleWorld::ContactGenerators& ParticleWorld::getContactGenerators()

{

return contactGenerators;

}

ParticleForceRegistry& ParticleWorld::getForceRegistry()

{

return registry;

}

void GroundContacts::init(cyclone::ParticleWorld::Particles \*particles)

{

GroundContacts::particles = particles;

}

unsigned GroundContacts::addContact(cyclone::ParticleContact \*contact,

unsigned limit) const

{

unsigned count = 0;

for (cyclone::ParticleWorld::Particles::iterator p = particles->begin();

p != particles->end();

p++)

{

cyclone::real y = (\*p)->getPosition().y;

if (y < 0.0f)

{

contact->contactNormal = cyclone::Vector3::UP;

contact->particle[0] = \*p;

contact->particle[1] = NULL;

contact->penetration = -y;

contact->restitution = 0.2f;

contact++;

count++;

}

if (count >= limit) return count;

}

return count;

}

ПРИЛОЖЕНИЕ Б

(*обязательное*)

Код программы. Модуль plinks

#ifndef CYCLONE\_PLINKS\_H

#define CYCLONE\_PLINKS\_H

#include "pcontacts.h"

namespace cyclone {

class ParticleLink : public ParticleContactGenerator

{

public:

Particle\* particle[2];

protected:

real currentLength() const;

public:

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const = 0;

};

/\*\*

\* Cables link a pair of particles, generating a contact if they

\* stray too far apart.

\*/

class ParticleCable : public ParticleLink

{

public:

/\*\*

\* Holds the maximum length of the cable.

\*/

real maxLength;

/\*\*

\* Holds the restitution (bounciness) of the cable.

\*/

real restitution;

public:

/\*\*

\* Fills the given contact structure with the contact needed

\* to keep the cable from over-extending.

\*/

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const;

};

/\*\*

\* Rods link a pair of particles, generating a contact if they

\* stray too far apart or too close.

\*/

class ParticleRod : public ParticleLink

{

public:

/\*\*

\* Holds the length of the rod.

\*/

real length;

public:

/\*\*

\* Fills the given contact structure with the contact needed

\* to keep the rod from extending or compressing.

\*/

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const;

};

class ParticleConstraint : public ParticleContactGenerator

{

public:

Particle\* particle;

Vector3 anchor;

protected:

real currentLength() const;

public:

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const = 0;

};

class ParticleCableConstraint : public ParticleConstraint

{

public:

/\*\*

\* Holds the maximum length of the cable.

\*/

real maxLength;

/\*\*

\* Holds the restitution (bounciness) of the cable.

\*/

real restitution;

public:

/\*\*

\* Fills the given contact structure with the contact needed

\* to keep the cable from over-extending.

\*/

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const;

};

/\*\*

\* Rods link a particle to an anchor point, generating a contact if they

\* stray too far apart or too close.

\*/

class ParticleRodConstraint : public ParticleConstraint

{

public:

/\*\*

\* Holds the length of the rod.

\*/

real length;

public:

/\*\*

\* Fills the given contact structure with the contact needed

\* to keep the rod from extending or compressing.

\*/

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const;

};

} // namespace cyclone

#endif // CYCLONE\_CONTACTS\_H

using namespace cyclone;

real ParticleLink::currentLength() const

{

Vector3 relativePos = particle[0]->getPosition() -

particle[1]->getPosition();

return relativePos.magnitude();

}

unsigned ParticleCable::addContact(ParticleContact \*contact,

unsigned limit) const

{

// Find the length of the cable

real length = currentLength();

// Check if we're over-extended

if (length < maxLength)

{

return 0;

}

// Otherwise return the contact

contact->particle[0] = particle[0];

contact->particle[1] = particle[1];

// Calculate the normal

Vector3 normal = particle[1]->getPosition() - particle[0]->getPosition();

normal.normalise();

contact->contactNormal = normal;

contact->penetration = length-maxLength;

contact->restitution = restitution;

return 1;

}

unsigned ParticleRod::addContact(ParticleContact \*contact,

unsigned limit) const

{

// Find the length of the rod

real currentLen = currentLength();

// Check if we're over-extended

if (currentLen == length)

{

return 0;

}

// Otherwise return the contact

contact->particle[0] = particle[0];

contact->particle[1] = particle[1];

// Calculate the normal

Vector3 normal = particle[1]->getPosition() - particle[0]->getPosition();

normal.normalise();

// The contact normal depends on whether we're extending or compressing

if (currentLen > length) {

contact->contactNormal = normal;

contact->penetration = currentLen - length;

} else {

contact->contactNormal = normal \* -1;

contact->penetration = length - currentLen;

}

// Always use zero restitution (no bounciness)

contact->restitution = 0;

return 1;

}

real ParticleConstraint::currentLength() const

{

Vector3 relativePos = particle->getPosition() - anchor;

return relativePos.magnitude();

}

unsigned ParticleCableConstraint::addContact(ParticleContact \*contact,

unsigned limit) const

{

// Find the length of the cable

real length = currentLength();

// Check if we're over-extended

if (length < maxLength)

{

return 0;

}

// Otherwise return the contact

contact->particle[0] = particle;

contact->particle[1] = 0;

// Calculate the normal

Vector3 normal = anchor - particle->getPosition();

normal.normalise();

contact->contactNormal = normal;

contact->penetration = length-maxLength;

contact->restitution = restitution;

return 1;

}

unsigned ParticleRodConstraint::addContact(ParticleContact \*contact,

unsigned limit) const

{

// Find the length of the rod

real currentLen = currentLength();

// Check if we're over-extended

if (currentLen == length)

{

return 0;

}

// Otherwise return the contact

contact->particle[0] = particle;

contact->particle[1] = 0;

// Calculate the normal

Vector3 normal = anchor - particle->getPosition();

normal.normalise();

// The contact normal depends on whether we're extending or compressing

if (currentLen > length) {

contact->contactNormal = normal;

contact->penetration = currentLen - length;

} else {

contact->contactNormal = normal \* -1;

contact->penetration = length - currentLen;

}

// Always use zero restitution (no bounciness)

contact->restitution = 0;

return 1;

}

ПРИЛОЖЕНИЕ В

(*обязательное*)

Код программы. Модуль pfgen

#ifndef CYCLONE\_PFGEN\_H

#define CYCLONE\_PFGEN\_H

#include "core.h"

#include "particle.h"

#include <vector>

namespace cyclone {

/\*\*

\* A force generator can be asked to add a force to one or more

\* particles.

\*/

class ParticleForceGenerator

{

public:

/\*\*

\* Overload this in implementations of the interface to calculate

\* and update the force applied to the given particle.

\*/

virtual void updateForce(Particle \*particle, real duration) = 0;

};

/\*\*

\* A force generator that applies a gravitational force. One instance

\* can be used for multiple particles.

\*/

class ParticleGravity : public ParticleForceGenerator

{

/\*\* Holds the acceleration due to gravity. \*/

Vector3 gravity;

public:

/\*\* Creates the generator with the given acceleration. \*/

ParticleGravity(const Vector3 &gravity);

/\*\* Applies the gravitational force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that applies a drag force. One instance

\* can be used for multiple particles.

\*/

class ParticleDrag : public ParticleForceGenerator

{

/\*\* Holds the velocity drag coeffificent. \*/

real k1;

/\*\* Holds the velocity squared drag coeffificent. \*/

real k2;

public:

/\*\* Creates the generator with the given coefficients. \*/

ParticleDrag(real k1, real k2);

/\*\* Applies the drag force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that applies a Spring force, where

\* one end is attached to a fixed point in space.

\*/

class ParticleAnchoredSpring : public ParticleForceGenerator

{

protected:

/\*\* The location of the anchored end of the spring. \*/

Vector3 \*anchor;

/\*\* Holds the sprint constant. \*/

real springConstant;

/\*\* Holds the rest length of the spring. \*/

real restLength;

public:

ParticleAnchoredSpring();

/\*\* Creates a new spring with the given parameters. \*/

ParticleAnchoredSpring(Vector3 \*anchor,

real springConstant,

real restLength);

/\*\* Retrieve the anchor point. \*/

const Vector3\* getAnchor() const { return anchor; }

/\*\* Set the spring's properties. \*/

void init(Vector3 \*anchor,

real springConstant,

real restLength);

/\*\* Applies the spring force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that applies a bungee force, where

\* one end is attached to a fixed point in space.

\*/

class ParticleAnchoredBungee : public ParticleAnchoredSpring

{

public:

/\*\* Applies the spring force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that fakes a stiff spring force, and where

\* one end is attached to a fixed point in space.

\*/

class ParticleFakeSpring : public ParticleForceGenerator

{

/\*\* The location of the anchored end of the spring. \*/

Vector3 \*anchor;

/\*\* Holds the sprint constant. \*/

real springConstant;

/\*\* Holds the damping on the oscillation of the spring. \*/

real damping;

public:

/\*\* Creates a new spring with the given parameters. \*/

ParticleFakeSpring(Vector3 \*anchor, real springConstant,

real damping);

/\*\* Applies the spring force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that applies a Spring force.

\*/

class ParticleSpring : public ParticleForceGenerator

{

/\*\* The particle at the other end of the spring. \*/

Particle \*other;

/\*\* Holds the sprint constant. \*/

real springConstant;

/\*\* Holds the rest length of the spring. \*/

real restLength;

public:

/\*\* Creates a new spring with the given parameters. \*/

ParticleSpring(Particle \*other,

real springConstant, real restLength);

/\*\* Applies the spring force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that applies a spring force only

\* when extended.

\*/

class ParticleBungee : public ParticleForceGenerator

{

/\*\* The particle at the other end of the spring. \*/

Particle \*other;

/\*\* Holds the sprint constant. \*/

real springConstant;

/\*\*

\* Holds the length of the bungee at the point it begins to

\* generator a force.

\*/

real restLength;

public:

/\*\* Creates a new bungee with the given parameters. \*/

ParticleBungee(Particle \*other,

real springConstant, real restLength);

/\*\* Applies the spring force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* A force generator that applies a buoyancy force for a plane of

\* liquid parrallel to XZ plane.

\*/

class ParticleBuoyancy : public ParticleForceGenerator

{

/\*\*

\* The maximum submersion depth of the object before

\* it generates its maximum boyancy force.

\*/

real maxDepth;

/\*\*

\* The volume of the object.

\*/

real volume;

/\*\*

\* The height of the water plane above y=0. The plane will be

\* parrallel to the XZ plane.

\*/

real waterHeight;

/\*\*

\* The density of the liquid. Pure water has a density of

\* 1000kg per cubic meter.

\*/

real liquidDensity;

public:

/\*\* Creates a new buoyancy force with the given parameters. \*/

ParticleBuoyancy(real maxDepth, real volume, real waterHeight,

real liquidDensity = 1000.0f);

/\*\* Applies the buoyancy force to the given particle. \*/

virtual void updateForce(Particle \*particle, real duration);

};

/\*\*

\* Holds all the force generators and the particles they apply to.

\*/

class ParticleForceRegistry

{

protected:

/\*\*

\* Keeps track of one force generator and the particle it

\* applies to.

\*/

struct ParticleForceRegistration

{

Particle \*particle;

ParticleForceGenerator \*fg;

};

/\*\*

\* Holds the list of registrations.

\*/

typedef std::vector<ParticleForceRegistration> Registry;

Registry registrations;

public:

/\*\*

\* Registers the given force generator to apply to the

\* given particle.

\*/

void add(Particle\* particle, ParticleForceGenerator \*fg);

/\*\*

\* Removes the given registered pair from the registry.

\* If the pair is not registered, this method will have

\* no effect.

\*/

void remove(Particle\* particle, ParticleForceGenerator \*fg);

void clear();

/\*\*

\* Calls all the force generators to update the forces of

\* their corresponding particles.

\*/

void updateForces(real duration);

};

}

#endif // CYCLONE\_PFGEN\_H

void ParticleForceRegistry::updateForces(real duration)

{

Registry::iterator i = registrations.begin();

for (; i != registrations.end(); i++)

{

i->fg->updateForce(i->particle, duration);

}

}

void ParticleForceRegistry::add(Particle\* particle, ParticleForceGenerator \*fg)

{

ParticleForceRegistry::ParticleForceRegistration registration;

registration.particle = particle;

registration.fg = fg;

registrations.push\_back(registration);

}

ParticleGravity::ParticleGravity(const Vector3& gravity)

: gravity(gravity)

{

}

void ParticleGravity::updateForce(Particle\* particle, real duration)

{

// Check that we do not have infinite mass

if (!particle->hasFiniteMass()) return;

// Apply the mass-scaled force to the particle

particle->addForce(gravity \* particle->getMass());

}

ParticleDrag::ParticleDrag(real k1, real k2)

: k1(k1), k2(k2)

{

}

void ParticleDrag::updateForce(Particle\* particle, real duration)

{

Vector3 force;

particle->getVelocity(&force);

// Calculate the total drag coefficient

real dragCoeff = force.magnitude();

dragCoeff = k1 \* dragCoeff + k2 \* dragCoeff \* dragCoeff;

// Calculate the final force and apply it

force.normalise();

force \*= -dragCoeff;

particle->addForce(force);

}

ParticleSpring::ParticleSpring(Particle \*other, real sc, real rl)

: other(other), springConstant(sc), restLength(rl)

{

}

void ParticleSpring::updateForce(Particle\* particle, real duration)

{

// Calculate the vector of the spring

Vector3 force;

particle->getPosition(&force);

force -= other->getPosition();

// Calculate the magnitude of the force

real magnitude = force.magnitude();

magnitude = real\_abs(magnitude - restLength);

magnitude \*= springConstant;

// Calculate the final force and apply it

force.normalise();

force \*= -magnitude;

particle->addForce(force);

}

ParticleBuoyancy::ParticleBuoyancy(real maxDepth,

real volume,

real waterHeight,

real liquidDensity)

:

maxDepth(maxDepth), volume(volume),

waterHeight(waterHeight), liquidDensity(liquidDensity)

{

}

void ParticleBuoyancy::updateForce(Particle\* particle, real duration)

{

// Calculate the submersion depth

real depth = particle->getPosition().y;

// Check if we're out of the water

if (depth >= waterHeight + maxDepth) return;

Vector3 force(0,0,0);

// Check if we're at maximum depth

if (depth <= waterHeight - maxDepth)

{

force.y = liquidDensity \* volume;

particle->addForce(force);

return;

}

// Otherwise we are partly submerged

force.y = liquidDensity \* volume \*

(depth - maxDepth - waterHeight) / 2 \* maxDepth;

particle->addForce(force);

}

ParticleBungee::ParticleBungee(Particle \*other, real sc, real rl)

: other(other), springConstant(sc), restLength(rl)

{

}

void ParticleBungee::updateForce(Particle\* particle, real duration)

{

// Calculate the vector of the spring

Vector3 force;

particle->getPosition(&force);

force -= other->getPosition();

// Check if the bungee is compressed

real magnitude = force.magnitude();

if (magnitude <= restLength) return;

// Calculate the magnitude of the force

magnitude = springConstant \* (restLength - magnitude);

// Calculate the final force and apply it

force.normalise();

force \*= -magnitude;

particle->addForce(force);

}

ParticleFakeSpring::ParticleFakeSpring(Vector3 \*anchor, real sc, real d)

: anchor(anchor), springConstant(sc), damping(d)

{

}

void ParticleFakeSpring::updateForce(Particle\* particle, real duration)

{

// Check that we do not have infinite mass

if (!particle->hasFiniteMass()) return;

// Calculate the relative position of the particle to the anchor

Vector3 position;

particle->getPosition(&position);

position -= \*anchor;

// Calculate the constants and check they are in bounds.

real gamma = 0.5f \* real\_sqrt(4 \* springConstant - damping\*damping);

if (gamma == 0.0f) return;

Vector3 c = position \* (damping / (2.0f \* gamma)) +

particle->getVelocity() \* (1.0f / gamma);

// Calculate the target position

Vector3 target = position \* real\_cos(gamma \* duration) +

c \* real\_sin(gamma \* duration);

target \*= real\_exp(-0.5f \* duration \* damping);

// Calculate the resulting acceleration and therefore the force

Vector3 accel = (target - position) \* ((real)1.0 / (duration\*duration)) -

particle->getVelocity() \* ((real)1.0/duration);

particle->addForce(accel \* particle->getMass());

}

ParticleAnchoredSpring::ParticleAnchoredSpring()

{

}

ParticleAnchoredSpring::ParticleAnchoredSpring(Vector3 \*anchor,

real sc, real rl)

: anchor(anchor), springConstant(sc), restLength(rl)

{

}

void ParticleAnchoredSpring::init(Vector3 \*anchor, real springConstant,

real restLength)

{

ParticleAnchoredSpring::anchor = anchor;

ParticleAnchoredSpring::springConstant = springConstant;

ParticleAnchoredSpring::restLength = restLength;

}

void ParticleAnchoredBungee::updateForce(Particle\* particle, real duration)

{

// Calculate the vector of the spring

Vector3 force;

particle->getPosition(&force);

force -= \*anchor;

// Calculate the magnitude of the force

real magnitude = force.magnitude();

if (magnitude < restLength) return;

magnitude = magnitude - restLength;

magnitude \*= springConstant;

// Calculate the final force and apply it

force.normalise();

force \*= -magnitude;

particle->addForce(force);

}

void ParticleAnchoredSpring::updateForce(Particle\* particle, real duration)

{

// Calculate the vector of the spring

Vector3 force;

particle->getPosition(&force);

force -= \*anchor;

// Calculate the magnitude of the force

real magnitude = force.magnitude();

magnitude = (restLength - magnitude) \* springConstant;

// Calculate the final force and apply it

force.normalise();

force \*= magnitude;

particle->addForce(force);}

ПРИЛОЖЕНИЕ Г

(*обязательное*)

Код программы. Модуль pcontacts

#ifndef CYCLONE\_PCONTACTS\_H

#define CYCLONE\_PCONTACTS\_H

#include "particle.h"

namespace cyclone {

/\*

\* Forward declaration, see full declaration below for complete

\* documentation.

\*/

class ParticleContactResolver;

class ParticleContact

{

// ... Other ParticleContact code as before ...

/\*\*

\* The contact resolver object needs access into the contacts to

\* set and effect the contact.

\*/

friend class ParticleContactResolver;

public:

/\*\*

\* Holds the particles that are involved in the contact. The

\* second of these can be NULL, for contacts with the scenery.

\*/

Particle\* particle[2];

/\*\*

\* Holds the normal restitution coefficient at the contact.

\*/

real restitution;

/\*\*

\* Holds the direction of the contact in world coordinates.

\*/

Vector3 contactNormal;

/\*\*

\* Holds the depth of penetration at the contact.

\*/

real penetration;

/\*\*

\* Holds the amount each particle is moved by during interpenetration

\* resolution.

\*/

Vector3 particleMovement[2];

protected:

/\*\*

\* Resolves this contact, for both velocity and interpenetration.

\*/

void resolve(real duration);

/\*\*

\* Calculates the separating velocity at this contact.

\*/

real calculateSeparatingVelocity() const;

private:

/\*\*

\* Handles the impulse calculations for this collision.

\*/

void resolveVelocity(real duration);

/\*\*

\* Handles the interpenetration resolution for this contact.

\*/

void resolveInterpenetration(real duration);

};

/\*\*

\* The contact resolution routine for particle contacts. One

\* resolver instance can be shared for the whole simulation.

\*/

class ParticleContactResolver

{

protected:

/\*\*

\* Holds the number of iterations allowed.

\*/

unsigned iterations;

/\*\*

\* This is a performance tracking value - we keep a record

\* of the actual number of iterations used.

\*/

unsigned iterationsUsed;

public:

/\*\*

\* Creates a new contact resolver.

\*/

ParticleContactResolver(unsigned iterations);

/\*\*

\* Sets the number of iterations that can be used.

\*/

void setIterations(unsigned iterations);

void resolveContacts(ParticleContact \*contactArray,

unsigned numContacts,

real duration);

};

class ParticleContactGenerator

{

public:

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) const = 0;

};

} // namespace cyclone

#endif // CYCLONE\_CONTACTS\_H

using namespace cyclone;

void ParticleContact::resolve(real duration)

{

resolveVelocity(duration);

resolveInterpenetration(duration);

}

real ParticleContact::calculateSeparatingVelocity() const

{

Vector3 relativeVelocity = particle[0]->getVelocity();

if (particle[1]) relativeVelocity -= particle[1]->getVelocity();

return relativeVelocity \* contactNormal;

}

void ParticleContact::resolveVelocity(real duration)

{

// Find the velocity in the direction of the contact

real separatingVelocity = calculateSeparatingVelocity();

// Check if it needs to be resolved

if (separatingVelocity > 0)

{

// The contact is either separating, or stationary - there's

// no impulse required.

return;

}

// Calculate the new separating velocity

real newSepVelocity = -separatingVelocity \* restitution;

// Check the velocity build-up due to acceleration only

Vector3 accCausedVelocity = particle[0]->getAcceleration();

if (particle[1]) accCausedVelocity -= particle[1]->getAcceleration();

real accCausedSepVelocity = accCausedVelocity \* contactNormal \* duration;

// If we've got a closing velocity due to acceleration build-up,

// remove it from the new separating velocity

if (accCausedSepVelocity < 0)

{

newSepVelocity += restitution \* accCausedSepVelocity;

if (newSepVelocity < 0) newSepVelocity = 0;

}

real deltaVelocity = newSepVelocity - separatingVelocity;

// We apply the change in velocity to each object in proportion to

// their inverse mass (i.e. those with lower inverse mass [higher

// actual mass] get less change in velocity)..

real totalInverseMass = particle[0]->getInverseMass();

if (particle[1]) totalInverseMass += particle[1]->getInverseMass();

// If all particles have infinite mass, then impulses have no effect

if (totalInverseMass <= 0) return;

// Calculate the impulse to apply

real impulse = deltaVelocity / totalInverseMass;

// Find the amount of impulse per unit of inverse mass

Vector3 impulsePerIMass = contactNormal \* impulse;

// Apply impulses: they are applied in the direction of the contact,

// and are proportional to the inverse mass.

particle[0]->setVelocity(particle[0]->getVelocity() +

impulsePerIMass \* particle[0]->getInverseMass()

);

if (particle[1])

{

// Particle 1 goes in the opposite direction

particle[1]->setVelocity(particle[1]->getVelocity() +

impulsePerIMass \* -particle[1]->getInverseMass()

);

}

}

void ParticleContact::resolveInterpenetration(real duration)

{

// If we don't have any penetration, skip this step.

if (penetration <= 0) return;

// The movement of each object is based on their inverse mass, so

// total that.

real totalInverseMass = particle[0]->getInverseMass();

if (particle[1]) totalInverseMass += particle[1]->getInverseMass();

// If all particles have infinite mass, then we do nothing

if (totalInverseMass <= 0) return;

// Find the amount of penetration resolution per unit of inverse mass

Vector3 movePerIMass = contactNormal \* (penetration / totalInverseMass);

// Calculate the the movement amounts

particleMovement[0] = movePerIMass \* particle[0]->getInverseMass();

if (particle[1]) {

particleMovement[1] = movePerIMass \* -particle[1]->getInverseMass();

} else {

particleMovement[1].clear();

}

// Apply the penetration resolution

particle[0]->setPosition(particle[0]->getPosition() + particleMovement[0]);

if (particle[1]) {

particle[1]->setPosition(particle[1]->getPosition() + particleMovement[1]);

}

}

ParticleContactResolver::ParticleContactResolver(unsigned iterations)

:

iterations(iterations)

{

}

void ParticleContactResolver::setIterations(unsigned iterations)

{

ParticleContactResolver::iterations = iterations;

}

void ParticleContactResolver::resolveContacts(ParticleContact \*contactArray,

unsigned numContacts,

real duration)

{

unsigned i;

iterationsUsed = 0;

while(iterationsUsed < iterations)

{

// Find the contact with the largest closing velocity;

real max = REAL\_MAX;

unsigned maxIndex = numContacts;

for (i = 0; i < numContacts; i++)

{

real sepVel = contactArray[i].calculateSeparatingVelocity();

if (sepVel < max &&

(sepVel < 0 || contactArray[i].penetration > 0))

{

max = sepVel;

maxIndex = i;

}

}

if (maxIndex == numContacts) break;

contactArray[maxIndex].resolve(duration);

// Update the interpenetrations for all particles

Vector3 \*move = contactArray[maxIndex].particleMovement;

for (i = 0; i < numContacts; i++)

{

if (contactArray[i].particle[0] == contactArray[maxIndex].particle[0])

{

contactArray[i].penetration -= move[0] \* contactArray[i].contactNormal;

}

else if (contactArray[i].particle[0] == contactArray[maxIndex].particle[1])

{

contactArray[i].penetration -= move[1] \* contactArray[i].contactNormal;

}

if (contactArray[i].particle[1])

{

if (contactArray[i].particle[1] == contactArray[maxIndex].particle[0])

{

contactArray[i].penetration += move[0] \* contactArray[i].contactNormal;

}

else if (contactArray[i].particle[1] == contactArray[maxIndex].particle[1])

{

contactArray[i].penetration += move[1] \* contactArray[i].contactNormal;

}

}

}

iterationsUsed++;

}

}

ПРИЛОЖЕНИЕ  Д

(*обязательное*)

Код программы. Модуль particle

#ifndef CYCLONE\_PARTICLE\_H

#define CYCLONE\_PARTICLE\_H

#include "core.h"

namespace cyclone {

class Particle

{

public:

// ... Other Particle code as before ...

protected:

real inverseMass;

real damping;

Vector3 position;

/\*\*

\* Holds the linear velocity of the particle in

\* world space.

\*/

Vector3 velocity;

Vector3 forceAccum;

/\*\*

\* Holds the acceleration of the particle. This value

\* can be used to set acceleration due to gravity (its primary

\* use), or any other constant acceleration.

\*/

Vector3 acceleration;

/\*@}\*/

public:

void integrate(real duration);

void setMass(const real mass);

/\*\*

\* Gets the mass of the particle.

\*

\* @return The current mass of the particle.

\*/

real getMass() const;

void setInverseMass(const real inverseMass);

/\*\*

\* Gets the inverse mass of the particle.

\*

\* @return The current inverse mass of the particle.

\*/

real getInverseMass() const;

/\*\*

\* Returns true if the mass of the particle is not-infinite.

\*/

bool hasFiniteMass() const;

/\*\*

\* Sets both the damping of the particle.

\*/

void setDamping(const real damping);

real getDamping() const;

void setPosition(const Vector3 &position);

void setPosition(const real x, const real y, const real z);

void getPosition(Vector3 \*position) const;

Vector3 getPosition() const;

void setVelocity(const Vector3 &velocity);

void setVelocity(const real x, const real y, const real z);

void getVelocity(Vector3 \*velocity) const;

Vector3 getVelocity() const;

void setAcceleration(const Vector3 &acceleration);

void setAcceleration(const real x, const real y, const real z);

void getAcceleration(Vector3 \*acceleration) const;

Vector3 getAcceleration() const;

void clearAccumulator();

void addForce(const Vector3 &force);

};

}

#endif // CYCLONE\_BODY\_H

#include <assert.h>

#include <cyclone/particle.h>

using namespace cyclone;

/\*

\* --------------------------------------------------------------------------

\* FUNCTIONS DECLARED IN HEADER:

\* --------------------------------------------------------------------------

\*/

void Particle::integrate(real duration)

{

// We don't integrate things with zero mass.

if (inverseMass <= 0.0f) return;

assert(duration > 0.0);

// Update linear position.

position.addScaledVector(velocity, duration);

// Work out the acceleration from the force

Vector3 resultingAcc = acceleration;

resultingAcc.addScaledVector(forceAccum, inverseMass);

// Update linear velocity from the acceleration.

velocity.addScaledVector(resultingAcc, duration);

// Impose drag.

velocity \*= real\_pow(damping, duration);

// Clear the forces.

clearAccumulator();

}

void Particle::setMass(const real mass)

{

assert(mass != 0);

Particle::inverseMass = ((real)1.0)/mass;

}

real Particle::getMass() const

{

if (inverseMass == 0) {

return REAL\_MAX;

} else {

return ((real)1.0)/inverseMass;

}

}

void Particle::setInverseMass(const real inverseMass)

{

Particle::inverseMass = inverseMass;

}

real Particle::getInverseMass() const

{

return inverseMass;

}

bool Particle::hasFiniteMass() const

{

return inverseMass >= 0.0f;

}

void Particle::setDamping(const real damping)

{

Particle::damping = damping;

}

real Particle::getDamping() const

{

return damping;

}

void Particle::setPosition(const Vector3 &position)

{

Particle::position = position;

}

void Particle::setPosition(const real x, const real y, const real z)

{

position.x = x;

position.y = y;

position.z = z;

}

void Particle::getPosition(Vector3 \*position) const

{

\*position = Particle::position;

}

Vector3 Particle::getPosition() const

{

return position;

}

void Particle::setVelocity(const Vector3 &velocity)

{

Particle::velocity = velocity;

}

void Particle::setVelocity(const real x, const real y, const real z)

{

velocity.x = x;

velocity.y = y;

velocity.z = z;

}

void Particle::getVelocity(Vector3 \*velocity) const

{

\*velocity = Particle::velocity;

}

Vector3 Particle::getVelocity() const

{

return velocity;

}

void Particle::setAcceleration(const Vector3 &acceleration)

{

Particle::acceleration = acceleration;

}

void Particle::setAcceleration(const real x, const real y, const real z)

{

acceleration.x = x;

acceleration.y = y;

acceleration.z = z;

}

void Particle::getAcceleration(Vector3 \*acceleration) const

{

\*acceleration = Particle::acceleration;

}

Vector3 Particle::getAcceleration() const

{

return acceleration;

}

void Particle::clearAccumulator()

{

forceAccum.clear();

}

void Particle::addForce(const Vector3 &force)

{

forceAccum += force;

}

ПРИЛОЖЕНИЕ Е

(*обязательное*)

Спецификация

ПРИЛОЖЕНИЕ Ж

(*обязательное*)

Ведомость документов