CONTENTS

1. LIBRARY SETUP

To get started using the library, you first need to add the headers in the *Physics2D* folder to your project. Once that is done, you need to link the library *2D_Physics_Library.lib* to your project.

The last step is to include *Physics2D.h* into your project, after which the library is set up.

2. DOCUMENTATION

Documentation of the user accessible classes can be found below. These classes also have in code documentation. (Documentation is created using Doxygen: www.doxygen.org)

2.1. WORLD

The world is the main interface used to create objects and update the physics system.

FUNCTIONS

void P2D::World::ClearBodyForces ()

Manually clear all forces on bodies

Body * P2D::World::CreateBody (const BodyDef & *def*)

Create a body

Parameters:

in	*def*	Body definition

Returns:

Body created from the definition

Constraint * P2D::World::CreateConstraint (const ConstraintDef & *def*)

Create a constraint

Parameters:

in	*def*	Constraint definition

Returns:

Constraint created from the definition

RevoluteJoint * P2D::World::CreateJoint (const RevoluteJointDef & *def*)

Create a revolute joint

Parameters:

in	*def*	Revolute joint def

Returns:

Revolute joint created from the definition

DistanceJoint * P2D::World::CreateJoint (const DistanceJointDef & *def*)

Create a distance joint

Parameters:

in	*def*	Distance joint def

Returns:

Distance joint created from the definition

FixedJoint * P2D::World::CreateJoint (const FixedJointDef & *def*)

Create a fixed joint

Parameters:

in	*def*	Fixed joint def

Returns:

Fixed joint created from the definition

PrismaticJoint * P2D::World::CreateJoint (const PrismaticJointDef & *def*)

Create a prismatic joint

Parameters:

in	*def*	Prismatic joint def

Returns:

Prismatic joint created from the definition

CircleShape * P2D::World::CreateShape (const CircleShapeDef & *def*)

Create a circle shape

Parameters:

in	*def*	Circle shape definition

Returns:

Circle shape created from the definition

EdgeShape * P2D::World::CreateShape (const EdgeShapeDef & *def*)

Create an edge shape

Parameters:

in	*def*	Edge shape definition

Returns:

Edge shape created from the definition

ChainShape * P2D::World::CreateShape (const ChainShapeDef & *def*)

Create a chain shape

Parameters:

in	*def*	Chain shape definition

Returns:

Chain shape created from the definition

PolygonShape * P2D::World::CreateShape (const PolygonShapeDef & *def*)

Create a polygon shape

Parameters:

in	*def*	Polygon shape definition

Returns:

Polygon shape created from the definition

void P2D::World::DestroyBody (Body * *pBody*)

Destroy a body

Parameters:

in	*pBody*	Body to destroy

void P2D::World::DestroyConstraint (Constraint * *pConstraint*)

Destroy a constraint

Parameters:

in	*pConstraint*	Constraint to destroy

void P2D::World::DestroyJoint (RevoluteJoint * *pJoint*)

Destroy a revolute joint

Parameters:

in	*pJoint*	Revolute joint to destory

void P2D::World::DestroyJoint (DistanceJoint * *pJoint*)

Destroy a distance joint

Parameters:

in	*pJoint*	Distance joint to destory

void P2D::World::DestroyJoint (FixedJoint * *pJoint*)

Destroy a fixed joint

Parameters:

in	*pJoint*	Fixed joint to destory

void P2D::World::DestroyJoint (PrismaticJoint * *pJoint*)

Destroy a prismatic joint

Parameters:

in	*pJoint*	Prismatic joint to destory

void P2D::World::DestroyShape (CircleShape * *pShape*)

Destroy a circle shape

Parameters:

in	*pShape*	Circle shape to destroy

void P2D::World::DestroyShape (EdgeShape * *pShape*)

Destroy an edge shape

Parameters:

in	*pShape*	Edge shape to destroy

void P2D::World::DestroyShape (ChainShape * *pShape*)

Destroy a chain shape

Parameters:

in	*pShape*	Chain shape to destroy

void P2D::World::DestroyShape (PolygonShape * *pShape*)

Destroy a polygon shape

Parameters:

in	*pShape*	Polygon shape to destroy

P2D_FORCE_INL u32 P2D::World::GetBodyCount () const[inline]

Get the amount of bodies in the world

Returns:

Amount of bodies in the world

P2D_FORCE_INL Body* P2D::World::GetBodyList ()[inline]

Get the list of bodies in the world

Returns:

First body in the list

P2D_FORCE_INL u32 P2D::World::GetConstraintCount () const[inline]

Get the amount of constraints in the world

Returns:

Amount of constraints in the world

P2D_FORCE_INL Constraint* P2D::World::GetConstraintList ()[inline]

Get the list of constraints in the world

Returns:

First constraint in the list

P2D_FORCE_INL u32 P2D::World::GetContactCount () const[inline]

Get the amount of contacts in the world

Returns:

Amount of contacts in the world

P2D_FORCE_INL const Contact* P2D::World::GetContactList () const[inline]

Get the list of contacts in the world

Returns:

First contact in the list

P2D_FORCE_INL EventListener& P2D::World::GetEventListener ()[inline]

Get a reference to the event listener

Returns:

Reference to the event listener

P2D_FORCE_INL f32v2 P2D::World::GetGravity () const[inline]

Get the global gravity

Returns:

Global gravity

P2D_FORCE_INL u32 P2D::World::GetJointCount () const[inline]

Get the amount of joints in the world

Returns:

Amount of joints in the world

P2D_FORCE_INL Joint* P2D::World::GetJointList ()[inline]

Get the list of joints in the world

Note:

To get the specific joint type, the user needs to store the joint manually, which can be done with the return of the joint create functions

Returns:

First constraint in the list

P2D_FORCE_INL u32 P2D::World::GetShapeCount () const[inline]

Get the amount of shapes in the world

Returns:

Amount of shapes in the world

P2D_FORCE_INL f32 P2D::World::GetTimeStep () const[inline]

Get the current timestep used during physics updates

Returns:

Timestep used during physics updates

P2D_FORCE_INL u32 P2D::World::GetTouchingContactCount () const[inline]

Get the amount of touching contacts in the world

Returns:

Amount of touching contacts in the world

P2D_FORCE_INL const Contact* P2D::World::GetTouchingContactList () const[inline]

Get the list of touching contacts in the world

Returns:

First touching contact in the list

void P2D::World::Raycast (const RaycastInput & *input*, RaycastOutput & *output*)

Raycast the world

Parameters:

in	*input*	Raycast input
out	*output*	Raycast output

P2D_FORCE_INL void P2D::World::ResetContactFilter ()[inline]

Reset the contact filter to the default filter

P2D_FORCE_INL void P2D::World::SetAutoClearForces (bool *autoClear*)[inline]

Set whether to automatically clear forces during update

Note:

When disabled, it is expected that the user clears the forces manually, using ClearBodyForces();

Parameters:

in	*autoClear*	Whether to automatically clear forces during update

P2D_FORCE_INL void P2D::World::SetContactFilter (const ContactFilter & *filter*)[inline]

Set the contact filter used during collision filtering

Parameters:

in	*filter*	Contact filter

P2D_FORCE_INL void P2D::World::SetGravity (const f32v2 & *gravity*)[inline]

Set the global gravity

Parameters:

in	*gravity*	Gravity

P2D_FORCE_INL void P2D::World::SetTimeStep (f32 *timestep*)[inline]

Set the timestep used during physics updates

Parameters:

in	*timestep*	Timestep to be used during physics updates

void P2D::World::Step (f32 *timestep*)

Manually step through 1 iteration of the simulation, you can repeat this until enough time has passed for your current frame

Parameters:

in	*timestep*	Fixed timestep (e.g. 1/30 or 1/60), try to refrain from using bigger timesteps then 1/30, this could cause incorrect results

void P2D::World::Update (f32 *dt*)

Update all physics over a variable timestep, uses fixed timestep during steps, ...

Parameters:

in	*dt*	Timestep

2.2. BODY

The body is the main object, containing shapes and physics properties

BodyDef

bool P2D::BodyDef::active

Whether the body is active

f32 P2D::BodyDef::angle

Angle

bool P2D::BodyDef::awake

Whether the body is awake

f32v2 P2D::BodyDef::position

Position

BodyType P2D::BodyDef::type

Body type

BodyMassData

f32v2 P2D::BodyMassData::centerOfMass

Center of mass of the body

f32 P2D::BodyMassData::inertia

Angular inertia of the body

f32 P2D::BodyMassData::invInertia

Inverse of the angular inertia of the body

f32 P2D::BodyMassData::invMass

Inverse of the mass of the body

f32 P2D::BodyMassData::mass

Mass of the body

FUNCTIONS

void P2D::Body::AddShape (Shape * *pShape*)

Add a shape to the body

Parameters:

in	*pShape*	Shape to add

void P2D::Body::ApplyAngularImpulse (f32 *impulse*, bool *wake* = true)

Apply an angular impulse to the body

Parameters:

in	*impulse*	Angular impulse to apply on the body
in	*wake*	Whether to wake the body

void P2D::Body::ApplyForce (f32v2 *force*, bool *wake* = true)

Apply a force to the body

Parameters:

in	*force*	Force to apply on the body
in	*wake*	Whether to wake the body

void P2D::Body::ApplyForce (f32v2 *force*, f32v2 *point*, bool *wake* = true)

Apply a force to a point on the body

Parameters:

in	*force*	Force to apply on the body
in	*point*	Point where force is applied
in	*wake*	Whether to wake the body

void P2D::Body::ApplyImpulse (f32v2 *impulse*, bool *wake* = true)

Apply a linear impulse to the body

Parameters:

in	*impulse*	Linear impulse to apply on the body
in	*wake*	Whether to wake the body

void P2D::Body::ApplyImpulse (f32v2 *impulse*, f32v2 *point*, bool *wake* = true)

Apply a linear impulse to a point on the body

Parameters:

in	*impulse*	Linear impulse to apply on the body
in	*point*	Point where impulse is applied
in	*wake*	Whether to wake the body

void P2D::Body::ApplyTorque (f32 *torque*, bool *wake* = true)

Apply a torque to the body

Parameters:

in	*torque*	Torque to apply on the body
in	*wake*	Whether to wake the body

P2D_FORCE_INL AABB P2D::Body::GetAABB () const[inline]

Get the AABB of the body

Returns:

AABB of the body

P2D_FORCE_INL f32 P2D::Body::GetAngle () const[inline]

Get the angle of the body

Returns:

Angle of the body

P2D_FORCE_INL f32 P2D::Body::GetAngularVelocity () const[inline]

Get the angular velocity of the body

Returns:

Angular velocity of the body

P2D_FORCE_INL BodyType P2D::Body::GetBodyType () const[inline]

Get the body type

Returns:

Body type

P2D_FORCE_INL ContactNode* P2D::Body::GetContacts ()[inline]

Get the list of contact nodes in the body

Returns:

First contact node in the list

P2D_FORCE_INL JointNode* P2D::Body::GetJoints ()[inline]

Get the list of joints nodes in the body

Returns:

First joints node in the list

P2D_FORCE_INL f32v2 P2D::Body::GetLinearVelocity () const[inline]

Get the linear velocity of the body

Returns:

Linear velocity of the body

P2D_FORCE_INL const BodyMassData& P2D::Body::GetMassData () const[inline]

Get the mass data of the body

Returns:

Mass data of the body

P2D_FORCE_INL Body* P2D::Body::GetNext ()[inline]

Get the next body in the list

Returns:

Next body in the list

P2D_FORCE_INL f32v2 P2D::Body::GetPosition () const[inline]

Get the position of the body

Returns:

Position of the body

P2D_FORCE_INL Body* P2D::Body::GetPrev ()[inline]

Get the previous body in the list

Returns:

Previous body in the list

P2D_FORCE_INL u32 P2D::Body::GetShapeCount () const[inline]

Get the amount of shapes in the body

Returns:

Amount of shapes in the body

P2D_FORCE_INL Shape* P2D::Body::GetShapes ()[inline]

Get the list of shapes in the body

Returns:

First shape in the list

P2D_FORCE_INL i32 P2D::Body::GetSolverIndex () const[inline]

Get the solver id of the body (mostly for internal use)

Returns:

Solver id of the body

P2D_FORCE_INL Transform& P2D::Body::GetTransform ()[inline]

Get the transform of the body

Returns:

Transform of the body

P2D_FORCE_INL const Velocity& P2D::Body::GetVelocity () const[inline]

Get the velocity of the body

Returns:

Velocity of the body

P2D_FORCE_INL World* P2D::Body::GetWorld ()[inline]

Get the world

Returns:

World

P2D_FORCE_INL bool P2D::Body::IsActive () const[inline]

Check whether the body is active

Returns:

Whether the body is active

P2D_FORCE_INL bool P2D::Body::IsAwake () const[inline]

Check whether the body is awake

Returns:

Whether the body is awake

void P2D::Body::SetActive (bool *active*)

Set whether the body is awake

Parameters:

in	*active*	Whether the doby should be awake

void P2D::Body::SetAwake (bool *awake*)

Set whether the body is awake

Parameters:

in	*awake*	Whether the doby should be awake

void P2D::Body::UpdateAABB ()

Update the AABB of the body (mostly for internal use)

2.3. SHAPE

Shapes represent the part of a body that interacts with the world. All shapes and definitions extend from the base types, so underlying types have same functionality as base classes

MassData

f32 P2D::MassData::area

Area

f32v2 P2D::MassData::centerOfMass

Center of mass

f32 P2D::MassData::inertia

Rotational inertia

f32 P2D::MassData::mass

Mass

f32 P2D::MassData::shapeInertia

Inertia of the shape, independent of relative position

Material

f32 P2D::Material::density

density

f32 P2D::Material::dynamicFriction

dynamic friction

f32 P2D::Material::restitution

Restitution/bounciness

f32 P2D::Material::staticFriction

static friction

ShapeDef

CollisionFilter P2D::ShapeDef::collisionFilter

Contact filter

bool P2D::ShapeDef::isSensor

Whether the shape is a sensor

Material P2D::ShapeDef::material

Physics material

f32v2 P2D::ShapeDef::relpos

Relative position to body

FUNCTIONS

P2D_FORCE_INL AABB P2D::Shape::GetAABB () const[inline]

Get the AABB of the shape

Returns:

AABB of the shape

P2D_FORCE_INL Body* P2D::Shape::GetBody ()[inline]

Get the parent body

Returns:

Parent body

P2D_FORCE_INL const CollisionFilter& P2D::Shape::GetFilterData () const[inline]

Get the collision filter of the shape

Returns:

Collision filter of the shape

P2D_FORCE_INL const MassData& P2D::Shape::GetMassData () const[inline]

Get the mass data of the shape

Returns:

Mass data of the shape

P2D_FORCE_INL Material& P2D::Shape::GetMaterial ()[inline]

Get the material of the shape

Returns:

Material of the shape

P2D_FORCE_INL Shape* P2D::Shape::GetNext ()[inline]

Get the next shape in the list

Returns:

Next shape in the list

P2D_FORCE_INL Type P2D::Shape::GetType () const[inline]

Get the shape type

Returns:

Shape type

P2D_FORCE_INL bool P2D::Shape::IsSensor () const[inline]

Check whether the shape is a sensor

Returns:

Whether the shape is a sensor

void P2D::Shape::SetMass (f32 *mass*)[virtual]

Set the mass of the shape

Note:

This function can cause issues when the shape is already added to a body, only use before adding to a body

Parameters:

in	*mass*	Mass

virtual void P2D::Shape::SetRelPosition (const f32v2 & *relPos*)[inline], [virtual]

Set the relative position of the shape

Parameters:

in	*relPos*	Relative positition

void P2D::Shape::UpdateAABB ()[virtual]

Update the AABB of the shape (mostly for internal use)

void P2D::Shape::UpdateInertia ()[virtual]

Update the inertia of the shape (mostly for internal use)

void P2D::Shape::UpdateMass ()[virtual]

Update the mass of the body

Note:

Should be called after changing the density of the material
This function can cause issues when the shape is already added to a body, only use before adding to a body

2.3.1. CIRCLE SHAPE

The shapes represent a circle

CircleShapeDef

f32 P2D::CircleShapeDef::radius

Radius of the circle

FUNCTIONS

f32 P2D::CircleShape::GetRadius () const[inline]

Get the radius of the circle

Returns:

Radius of the circle

2.3.2. EDGE SHAPE

The shape represents an edge

EdgeShapeDef

f32v2 P2D::EdgeShapeDef::v0

Vertex 0

f32v2 P2D::EdgeShapeDef::v1

Vertex 1

FUNCTIONS

P2D_FORCE_INL const f32v2& P2D::EdgeShape::GetNormal () const[inline]

Get the normal of the edge

Returns:

Normal of the edge

P2D_FORCE_INL const f32v2& P2D::EdgeShape::GetV0 () const[inline]

Get vertex 0 of the edge

Returns:

Vertex 0 of the edge

P2D_FORCE_INL const f32v2& P2D::EdgeShape::GetV1 () const[inline]

Get vertex 1 of the edge

Returns:

Vertex 1 of the edge

2.3.3. CHAIN SHAPE

The shape represents a chain of edges

ChainShapeDef

u32 P2D::ChainShapeDef::numPoints

Number of points in the chain

f32v2* P2D::ChainShapeDef::points

Points in the chain

FUNCTIONS

void P2D::ChainShape::GetChildEdge (EdgeShape * *pEdge*, u32 *childIndex*)

Get an edge in the chain

Parameters:

in,out	*pEdge*	Edge to set values to
in	*childIndex*	Index of the edge in the chain

P2D_FORCE_INL u32 P2D::ChainShape::GetNumPoints () const[inline]

Get the number of points in the chain

Returns:

Number of points in the chain

P2D_FORCE_INL const f32v2* P2D::ChainShape::GetPoints () const[inline]

Get the points in the chain

Returns:

Points in the chain

2.3.4. POLYGON SHAPE

The shape represents a convex polygon

PolygonShapeDef

u32 P2D::PolygonShapeDef::numPoints

Number of points in the polygon

f32v2* P2D::PolygonShapeDef::points

Points in the polygon

FUNCTIONS

bool P2D::PolygonShape::CheckWinding () const

Check whether the winding is in the correct order (CCW)

Returns:

Whether the winding is correct

P2D_FORCE_INL u32 P2D::PolygonShape::GetNumPoints () const[inline]

Get the points in the polygon

Returns:

Points in the polygon

P2D_FORCE_INL const f32v2* P2D::PolygonShape::GetPoints () const[inline]

Get the number of points in the polygon

Returns:

Number of points in the polygon

void P2D::PolygonShape::SetAsBox (f32 *width*, f32 *height*)

Set the polygon as a box

Parameters:

in	*width*	Width of the box
in	*height*	Height of the box

void P2D::PolygonShape::SetAsRegularPolygon (u32 *numSides*, f32 *radius*)

Set the polygon as a regular polygon

Parameters:

in	*numSides*	Number of sides of the polygon
in	*radius*	Radius of the polygon

2.4. CONSTRAINT

Constrains the movement of a body

ConstraintDef

f32 P2D::ConstraintDef::axisMaxValue

max value on the axis

f32 P2D::ConstraintDef::axisMinValue

Min value on the axis

f32v2 P2D::ConstraintDef::axisPosition

< Axis to constrain the position to (constraining axis) Position of the axis

f32 P2D::ConstraintDef::axisTolerance

Tolerance perpendicular to the axis, large tolerances can constrain the axis to a box

bool P2D::ConstraintDef::constrainPosition

Whether to constrain position

bool P2D::ConstraintDef::constrainRotation

Whether to constraint rotation

f32 P2D::ConstraintDef::maxAngle

Max angle

f32 P2D::ConstraintDef::minAngle

Min angle

Body* P2D::ConstraintDef::pBody

Body to constrain

FUNCTIONS

const f32v2& P2D::Constraint::GetAxisPosition () const[inline]

Get the axis position

Returns:

Axis position

f32 P2D::Constraint::GetAxisTolerance () const[inline]

Get the axis tolerance

Returns:

Axis tolerance

const f32v2& P2D::Constraint::GetContrainingAxis () const[inline]

Get the constraining axis

Returns:

Constraining axis

f32 P2D::Constraint::GetMaxAngle () const[inline]

Get the max angle

Returns:

Max angle

f32 P2D::Constraint::GetMaxAxisLimit () const[inline]

Get the axis min limit

Returns:

Axis max limit

f32 P2D::Constraint::GetMinAngle () const[inline]

Get the min angle

Returns:

Min angle

f32 P2D::Constraint::GetMinAxisLimit () const[inline]

Get the axis min limit

Returns:

Axis min limit

bool P2D::Constraint::IsPositionConstrained () const[inline]

Check whether position is constrained

Returns:

Whether position is constrained

bool P2D::Constraint::IsRotationConstrained () const[inline]

Check whether rotation is constrained

Returns:

Whether rotation is constrained

void P2D::Constraint::SetAngleLimits (f32 *min*, f32 *max*)[inline]

Set the angle limits

Parameters:

in	*min*	Min angle
in	*max*	Max angle

void P2D::Constraint::SetAxisLimits (f32 *min*, f32 *max*)[inline]

Set the axis limits

Parameters:

in	*min*	Min limit
in	*max*	Max limit

void P2D::Constraint::SetAxisPosition (const f32v2 & *position*)[inline]

Set the axis position

Parameters:

in	*position*	Axis position

void P2D::Constraint::SetAxisTolerance (f32 *tolerance*)[inline]

Set the axis tolerance

Parameters:

in	*tolerance*	Tolerance

void P2D::Constraint::SetConstrainPosition (bool *constrain*)[inline]

Set whether position should be constrained

Parameters:

in	*constrain*	Whether position should be constrained

void P2D::Constraint::SetConstrainRotation (bool *constrain*)[inline]

Set whether rotation should be constrained

Parameters:

in	*constrain*	Whether rotation should be constrained

void P2D::Constraint::SetConstrainingAxis (const f32v2 & *axis*)[inline]

Set the constraining axis

Parameters:

in	*axis*	Constraining axis

void P2D::Constraint::SetMaxAngle (f32 *angle*)[inline]

Set the max angle

Parameters:

in	*angle*	Angle

void P2D::Constraint::SetMaxAxisLimit (f32 *max*)[inline]

Set the max axis limit

Parameters:

in	*max*	Max limit

void P2D::Constraint::SetMinAngle (f32 *angle*)[inline]

Set the min angle

Parameters:

in	*angle*	Angle

void P2D::Constraint::SetMinAxisLimit (f32 *min*)[inline]

Set the min axis limit

Parameters:

in	*min*	Min limit

void P2D::Constraint::Update ()

Update the constraint (for internal use)

2.5. JOINT

Joint between 2 shapes

JointDef

Body* P2D::JointDef::pBody0

Body 0, if nullptr, the joint is connected to the world

Body* P2D::JointDef::pBody1

Body 1

f32v2 P2D::JointDef::pos0

Relative position to body 0/world

f32v2 P2D::JointDef::pos1

Relative position to body 1

JointNode

Joint* P2D::JointNode::pJoint

Joint

JointNode* P2D::JointNode::pNext

Next node

Body* P2D::JointNode::pOther

Other body, nullptr if connected to world

JointNode* P2D::JointNode::pPrev

Previous node

FUNCTIONS

bool P2D::Joint::DoShapesCollide ()[virtual]

Check whether shapes of the parent bodies should collide

Returns:

Whether shape should collide

const f32v2& P2D::Joint::GetPos0 () const[inline]

Get relative position to body 0/world

Returns:

Relative position to body 0/world

const f32v2& P2D::Joint::GetPos1 () const[inline]

Get relative position to body 1

Returns:

Relative position to body 1

void P2D::Joint::Update (f32 *dt*)[virtual]

Update the joint (used internally)

2.5.1. REVOLUTE JOINT

The revolute joint or rotational joint fixes the movement of the points, but allows rotation. Rotation can be limited and a motor speed can be applied.

RevoluteJointDef

bool P2D::RevoluteJointDef::limitAngle

Whether to limit the angle

f32 P2D:: RevoluteJointDef::minAngle

Min angle limit

f32 P2D:: RevoluteJointDef::maxAngle

Max angle limit

bool P2D::RevoluteJointDef::hasMotor

Whether the joint has a motor

f32 P2D:: RevoluteJointDef::motorSpeed

Motor speed

FUNCTIONS

P2D_FORCE_INL f32 P2D::RevoluteJoint::GetMaxAngle () const[inline]

Get the max angle

Returns:

Max angle

P2D_FORCE_INL f32 P2D::RevoluteJoint::GetMinAngle () const[inline]

Get the min angle

Returns:

Min angle

P2D_FORCE_INL f32 P2D::RevoluteJoint::GetMotorSpeed () const[inline]

Get the motor speed

Returns:

Motor speed

P2D_FORCE_INL bool P2D::RevoluteJoint::HasMotor () const[inline]

Check whether a motor speed is applied

Returns:

Whether a motor speed is applied

P2D_FORCE_INL bool P2D::RevoluteJoint::IsAngleLimited () const[inline]

Check whether the angle is limited

Returns:

Whether the angle is limited

P2D_FORCE_INL void P2D::RevoluteJoint::LimitAngle (bool *limit*)[inline]

Set whether to limit the angle

Parameters:

in	*limit*	Whether to limit the angle

P2D_FORCE_INL void P2D::RevoluteJoint::SetAngleLimits (f32 *min*, f32 *max*)[inline]

Set the angle limits

Parameters:

in	*min*	Min angle
In	*max*	Max angle

P2D_FORCE_INL void P2D::RevoluteJoint::SetMaxAngle (f32 *angle*)[inline]

Set the min angle limit

Parameters:

in	*angle*	Max angle

P2D_FORCE_INL void P2D::RevoluteJoint::SetMinAngle (f32 *angle*)[inline]

Set the min angle limit

Parameters:

in	*angle*	Min angle

P2D_FORCE_INL void P2D::RevoluteJoint::SetMotor (bool *hasMotor*)[inline]

Set whether a motor speed is applied

Parameters:

in	*hasMotor*	Whether a motor speed is applied

P2D_FORCE_INL void P2D::RevoluteJoint::SetMotorSpeed (f32 *speed*)[inline]

Set the motor speed

Parameters:

in	*speed*	Motor speed

2.5.2. DISTANCE JOINT

Joint that keeps 2 points a certain distance from each other

DistanceJointDef

f32 P2D::DistanceJointDef::distance

Distance between points

f32 P2D:: DistanceJointDef::tolerance

Distance tolerance

FUNCTIONS

P2D_FORCE_INL f32 P2D::DistanceJoint::GetDistance () const[inline]

Get the distance of the joint

Returns:

Joint distance

P2D_FORCE_INL f32 P2D::DistanceJoint::GetTolerance () const[inline]

Get the tolerance of the joint

Returns:

Joint tolerance

P2D_FORCE_INL void P2D::DistanceJoint::SetDistance (f32 *distance*)[inline]

Set the distance of the joint

Parameters:

in	*distance*	Distance

P2D_FORCE_INL void P2D::DistanceJoint::SetTolerance (f32 *tolerance*)[inline]

Set the tolerance of the joint

Parameters:

in	*tolerance*	Tolerance

2.5.3. FIXED JOINT

The fixed joint keeps the points at each other location and locks movement

FixedJointDef

f32 P2D::FixedJointDef::angle

Angle between 2 bodies

2.5.4. PRISMATIC JOINT

The prismatic joint only allows movement similar to a hydraulic cylinder (with a simple revolute joint on it)

PrismaticJointDef

f32 P2D::FixedJointDef::axis

Sliding axis

f32 P2D::FixedJointDef::tolerance

Axis tolerance

f32 P2D::FixedJointDef::minValue

Min limit

f32 P2D::FixedJointDef::maxValue

Max limit

FUNCTIONS

P2D_FORCE_INL const f32v2& P2D::PrismaticJoint::GetAxis () const[inline]

Get the sliding axis of the joint

Returns:

Sliding axis

P2D_FORCE_INL f32 P2D::PrismaticJoint::GetMaxlimit () const[inline]

Get the max limit of the joint

Returns:

Max limit

P2D_FORCE_INL f32 P2D::PrismaticJoint::GetMinLimit () const[inline]

Get the min limit of the joint

Returns:

Min limit

P2D_FORCE_INL f32 P2D::PrismaticJoint::GetTolerance () const[inline]

Get the tolerance of the joint

Returns:

Tolerance

P2D_FORCE_INL void P2D::PrismaticJoint::SetAxis (const f32v2 & *axis*)[inline]

Set the sliding axis of the joint

Parameters:

in	*axis*	Sliding axis

P2D_FORCE_INL void P2D::PrismaticJoint::SetLimits (f32 *min*, f32 *max*)[inline]

Set the tolerance of the joint

Parameters:

in	*min*	Min limit
in	*max*	max limit

P2D_FORCE_INL void P2D::PrismaticJoint::SetMaxLimit (f32 *max*)[inline]

Set the max limit of the joint

Parameters:

in	*max*	max limit

P2D_FORCE_INL void P2D::PrismaticJoint::SetMinLimit (f32 *min*)[inline]

Set the min limit of the joint

Parameters:

in	*min*	Min limit

P2D_FORCE_INL void P2D::PrismaticJoint::SetTolerance (f32 *tolerance*)[inline]

Set the tolerance of the joint

Parameters:

in	*tolerance*	Tolerance

2.6. CONTACT

Contact between 2 shapes

ContactNode

Connection between the contact and a body, stored as a linked list

Contact* P2D::ContactNode::pContact

Contact

Body* P2D:: ContactNode::pOther

Other body in contact

ContactNode* P2D:: ContactNode::pNext

Next node

ContactNode* P2D:: ContactNode::pPrev

Previous node

FUNCTIONS

Contact * P2D::Contact::Create (Shape * *pShape0*, Shape * *pShape1*, BlockAllocator * *pAlloc*)[static]

Create a contact (internal use only)

void P2D::Contact::Destroy (Contact * *pContact*, BlockAllocator * *pAlloc*)[static]

Destroy a contact (internal use only)

void P2D::Contact::Evaluate (Manifold & *manifold*)[virtual]

Evaluate the contact (internal use only)

P2D_FORCE_INL const Manifold& P2D::Contact::GetManifold () const[inline]

Get the contact manifold

Returns:

Contact manifold

P2D_FORCE_INL const Contact* P2D::Contact::GetNext () const[inline]

Get the next contact

Returns:

Next contact

P2D_FORCE_INL const Contact* P2D::Contact::GetNextTouching () const[inline]

Get the next touching contact

Returns:

Next touching contact

P2D_FORCE_INL const Contact* P2D::Contact::GetPrev () const[inline]

Get the previous contact

Returns:

Previous contact

P2D_FORCE_INL Shape* P2D::Contact::GetShape0 ()[inline]

Get shape 0 of the contact

Returns:

Shape 0

P2D_FORCE_INL Shape* P2D::Contact::GetShape1 ()[inline]

Get shape 1 of the contact

Returns:

Shape 1

P2D_FORCE_INL bool P2D::Contact::IsActive () const[inline]

Check whether the contact is active

Returns:

Whether the contact is active

P2D_FORCE_INL bool P2D::Contact::IsTouching () const[inline]

Check whether the contact is touching

Returns:

Whether the contact is touching

P2D_FORCE_INL void P2D::Contact::SetCheckFilter (bool *check*)[inline]

Set whether to recheck the contact filter

Parameters:

in	*Check*	Whether to recheck the contact filter

2.7. OTHER

2.7.1. AABB

The AABB (Axis Aligned Bounding Box) encapsulates the shapes

FUNCTIONS

P2D_INL P2D::AABB::AABB ()

Create an empty AABB

P2D_INL P2D::AABB::AABB (const f32v2 & *min*, const f32v2 & *max*)

Create a AABB from a min and max

Parameters:

in	*min*	Min
in	*max*	Max

P2D_INL P2D::AABB::AABB (f32 *left*, f32 *bottom*, f32 *right*, f32 *top*)

Create an AABB from values

Parameters:

in	*left*	Left
in	*bottom*	Bottom
in	*right*	Right
in	*top*	Top

P2D_INL void P2D::AABB::Combine (const AABB & *aabb*)

Combine 2 AABBs

Parameters:

in	*aabb*	AABB to combine with

P2D_INL bool P2D::AABB::Contains (const AABB & *aabb*) const

Check whether another AABB is completely in the AABB

Parameters:

in	*aabb*	AABB to check

P2D_INL f32 P2D::AABB::GetPerimeter () const

Get the perimeter of the AABB

Returns:

Perimeter

P2D_INL void P2D::AABB::Move (const f32v2 & *v*)

Move an AABB

Parameters:

in	*v*	Displacement

P2D_INL bool P2D::AABB::Overlaps (const AABB & *aabb*) const

Check whether 2 AABBs overlap

Parameters:

in	*aabb*	AABB to check overlap with

P2D_INL void P2D::AABB::Pad (f32 *value*)

Pad the AABB (extend size)

Parameters:

in	*value*	Value to pad with

2.7.2. COLLISION FILTER

Structure containing data used for filtering

u16 P2D::CollisionFilter::category

Collision category

u16 P2D::CollisionFilter::collisionMask

Collision mask, with which groups to collide

i16 P2D::CollisionFilter::group

Collision group, always wins over mask
0: No collision group
pos: Always collides with same group
neg: Never collides with same group

2.7.3. CONTACT FILTER

User overridable filter to have finer control over filtering

FUNCTIONS

bool P2D::ContactFilter::ShouldCollide (Shape * *pShape0*, Shape * *pShape1*)[virtual]

Check whether 2 shape collide

Parameters:

in	*pShape0*	First shape
in	*pShape1*	Second shape

2.7.4. EVENT LISTENER

Manages and calls events

EVENTS INFO

OnCollisionEnter: Called when the shapes in a contact start to overlap

OnCollisionStay: Called when the shapes in a contact are overlapping and have been in the previous step

OnCollisionLeave: Called when the shapes in a contact stop to overlap

OnContactCreate: Called when a contact is created

OnContactDestroy: Called when a contact is destroyed

PreSolve: Called every step a contact exists, allows the user to control contact and even modify it. Returning false, makes the contact's collision skip a step.

FUNCTIONS

P2D_FORCE_INL void P2D::EventListener::OnCollisionEnter (Contact * *pContact*) const[inline]

Run the OnCollisionEnter callback

P2D_FORCE_INL void P2D::EventListener::OnCollisionLeave (Contact * *pContact*) const[inline]

Run the OnCollisionLeave callback

P2D_FORCE_INL void P2D::EventListener::OnCollisionStay (Contact * *pContact*) const[inline]

Run the OnCollisionStay callback

P2D_FORCE_INL void P2D::EventListener::OnContactCreate (Contact * *pContact*) const[inline]

Run the OnContactCreate callback

P2D_FORCE_INL void P2D::EventListener::OnContactDestroy (Contact * *pContact*) const[inline]

Run the OnContactDestroy callback

P2D_FORCE_INL bool P2D::EventListener::PreSolve (Contact * *pContact*) const[inline]

Run the PreSolve callback

Returns:

PreSolve callback resuly

void P2D::EventListener::SetOnCollisionEnterCallback (OnCollisionEnterFunc *onCollisionEnter*)[inline]

Set the OnCollisionEnter callback

Parameters:

in	*onCollisionEnter*	OnCollisionEnter callback

void P2D::EventListener::SetOnCollisionLeaveCallback (OnCollisionLeaveFunc *onCollisionLeave*)[inline]

Set the OnCollisionLeave callback

Parameters:

in	*onCollisionLeave*	OnCollisionLeave callback

void P2D::EventListener::SetOnCollisionStayCallback (OnCollisionStayFunc *onCollisionStay*)[inline]

Set the OnCollisionStay callback

Parameters:

in	*onCollisionStay*	OnCollisionStay callback

void P2D::EventListener::SetOnContactCreateCallback (OnCollisionLeaveFunc *onContactCreate*)[inline]

Set the OnContactCreate callback

Parameters:

in	*onContactCreate*	OnContactCreate callback

void P2D::EventListener::SetOnContactDestroyCallback (OnCollisionLeaveFunc *onContactDestroy*)[inline]

Set the OnContactnDestroy callback

Parameters:

in	*onContactDestroy*	OnContactnDestroy callback

void P2D::EventListener::SetPreSolveCallback (PreSolveFunc *preSolve*)[inline]

Set the PreSolve callback

Parameters:

in	*preSolve*	PreSolv callback

2.7.5. MANIFOLD

Contact manifold

u32 P2D::Manifold::numPairs

Amount of manifold points

ManifoldPair P2D::Manifold::pairs[g_MaxManifoldPairs]

Manifold points

ManifoldPair

f32v2 P2D::ManifoldPair::normal

Normal

f32v2 P2D::ManifoldPair::position0

Position 0

f32v2 P2D::ManifoldPair::position1

Position 1

f32 P2D::ManifoldPair::separation

Separation

2.7.6. RAYCAST

RaycastInput

f32v2 P2D::RaycastInput::direction

Ray direction

f32v2 P2D:: RaycastInput::length

Ray length

f32v2 P2D:: RaycastInput::position

Ray starting position

RaycastOutput

f32v2 P2D::ManifoldPair::fraction

Fraction of ray length to hit

f32v2 P2D::ManifoldPair::hit

Whether the ray hit

f32v2 P2D::ManifoldPair::normal

Normal at hit

f32 P2D::ManifoldPair::pShape

Shape the ray hit

2.7.7. TRANSFORM AND VELOCITY

TRANSFORM

f32v2 P2D::Transform::position

Position

f32 P2D:: Transform::rotation

Rotation/angle

P2D_FORCE_INL void P2D::Transform::Move (const f32v2& *relpos*)[inline]

Set whether to recheck the contact filter

Parameters:

in	*relpos*	Relative position

VELOCITY

f32v2 P2D::Velocity::linearVelocity

Linear velocity

f32 P2D:: Velocity::angularVelocity

Angular velocity

2.7.8. VEC2

2D vector with x and y coordinates. f32v2 is a specialized type for f32 (32-bit float).

CONSTANTS

- static const Vec2 Zero = Vec2<T>(0, 0)
- static const Vec2 One = Vec2<T>(1, 1)
- static const Vec2 AxisX = Vec2<T>(1, 0)
- static const Vec2 AxisY = Vec2<T>(0, 1)
- static const Vec2 Left = Vec2<T>(-1, 0)
- static const Vec2 Right = Vec2<T>(1, 0)
- static const Vec2 Up = Vec2<T>(0, 1)
- static const Vec2 Down = Vec2<T>(0, -1)

FUNCTIONS

template<typename T > P2D::Vec2< T >::Vec2 ()

Create a vec2

template<typename T > template<typename U > P2D::Vec2< T >::Vec2 (U *val*)[explicit]

Create a vec2

Parameters:

in	*val*	Value

template<typename T > template<typename X , typename Y > P2D::Vec2< T >::Vec2 (X *x*, Y *y*)

Create a vec2

Parameters:

in	*x*	X-value
in	*y*	Y-value

template<typename T > template<typename U > P2D::Vec2< T >::Vec2 (const Vec2< U > & *v*)

Create a vec2 from another vec2

Parameters:

in	*v*	Vec2

template<typename T > T P2D::Vec2< T >::Angle () const

Get the angle of a vector

Return:

Angle

template<typename T > T P2D::Vec2< T >::Angle (const Vec2< T > & *v*) const

Get the angle between 2 vectors

Parameters:

in	*v*	Vec2

Return:

Angle between 2 vectors

template<typename T > T P2D::Vec2< T >::Cross (const Vec2< T > & *v*) const

Get the 2D cross product of 2 vector

Return:

Cross product

template<typename T> Vec2< T > P2D::Vec2< T >::Cross (T *val*) const

Get the 2D cross product of a vector and a scalar (Z-axis)

Return:

Cross product

template<typename T > T P2D::Vec2< T >::Distance (const Vec2< T > & *v*)

Get the distance between 2 vectors

Return:

Distance between vectors

template<typename T > T P2D::Vec2< T >::Dot (const Vec2< T > & *v*) const

Get the dot product of 2 vectors

Parameters:

in	*v*	Vec2

Return:

Dot product

template<typename T > bool P2D::Vec2< T >::Equals (const Vec2< T > & *v*) const

Check if 2 vectors are the same

Parameters:

in	*v*	Vec2

Return:

Whether 2 vectors are the same

template<typename T> bool P2D::Vec2< T >::Equals (const Vec2< T > & *v*, T *epsilon*) const

Check if 2 vectors are the same, with an epsilon

Parameters:

in	*v*	Vec2
in	*epsilon*	Epsilon

Return:

Whether 2 vectors are the same

template<typename T > T P2D::Vec2< T >::Length () const

Get the length of the vector

Return:

Length of the vector

template<typename T> Vec2< T > P2D::Vec2< T >::Lerp (const Vec2< T > & *v*, T *factor*) const

Lerp between 2 vectors

Parameters:

in	*v*	Other vector
in	*factor*	Lerp factor

Return:

Lerped vector

template<typename T > Vec2< T > & P2D::Vec2< T >::Normalize ()

Normalize the vector

template<typename T > Vec2< T > P2D::Vec2< T >::Normalized () const

Get the normalized version of the vector

template<typename T> Vec2< T > & P2D::Vec2< T >::Rotate (T *angle*)

Rotate the vector (changes vector)

Parameters:

in	*angle*	Angle to rotate by

Return:

Reference to the vector

template<typename T> Vec2< T > & P2D::Vec2< T >::RotateAroundPoint (const Vec2< T > & point, T angle)

Rotate the vector around a point

Parameters:

in	*point*	Point to rotate around
in	*angle*	Angle to rotate by

Return:

Reference to the vector

template<typename T> Vec2< T > P2D::Vec2< T >::Rotated (T *angle*) const

Get the rotated vector

Parameters:

in	*angle*	Angle to rotate by

Return:

Rotated vector

Parameters:

in	*angle*	Angle to rotate by

template<typename T > T P2D::Vec2< T >::SqDistance (const Vec2< T > & *v*)

Get the square distance between 2 vectors

Return:

Square distance between vectors

template<typename T > T P2D::Vec2< T >::SqLength () const

Get the square length of the vector

Return:

Square length of the vector