Physics Effects Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Library Overview 4
Scope of This Document 4
Purpose and Features 4
Embedding in Program 4
Name Space 4
Main Functions 4
Resource Consumption 5
Embedding in Program 5
Sample Programs 5
2 Architecture 8
Representing Objects 8
Motion Type 10
Contact Filter 11
Type of Shape 11
Type of Joint 13
3 Usage Procedure 15
About Types 15
Initialize the Task Manager 15
Create a Rigid Body 16
Create a Joint 17
4 Stages 19
Simulation Stages 19
Apply an External Force 20
Broad Phase 20
Collision Detection 23
Constraint Calculation 23
Position Calculation 25
5 Raycast 26
Prepare a Raycast 26
Raycast 27
6 Sort 28
7 Serialization Input/Output 29
Overview of Serialization Processing 29
Callback Functions Called When Performing Serialization Processing 29
Serialization Processing Procedure 30
Precautions for Serialization Processing 31
8 Debug Rendering 32
Debug Rendering 32
Outline of Usage Procedure of Debug Rendering 32
Creation of Debug Rendering 32
Registration of Rendering Callback Function 32
Visualization of Debug Information 33
Release of Debug Rendering 33

Rendering of Islands and Triangles of Large Mesh 34

9 FAQ/TIPS 35

10 Reference Information 36

A List of the Behavior Change by Type 36

Compose Contact Points....36

Mechanism of a Joint Constraint....36

1 Library Overview

Scope of This Document

This is a document for physics simulation libraries, which reproduce a behavior of an object. This document explains object representation methods of the Physics Effects library, guidance for implementing processing stages such as a broad phase and collision detection, and an overview of a raycast and sort processing, etc.

Purpose and Features

Physics Effects library is a real time physics simulation library, which reproduces a behavior of an object in accordance with the laws of physics. The Physics Effects library compactly implements functions required for an application development.

Each individual processing unit, functions such as collision detection and a raycast that are included in a simulation pipeline are independent for each component so that the unit and functions can be used partially.

API consists of three levels: base level (base_level), low level (low_level) and high level (high_level). The upper level API uses the lower level API. base_level, which is the lowest level, only processes a single data. low_level, which is the upper level of base_level, processes multiple data assuming that the processing is performed in parallel. high_level fixes simulation pipelines and conceals complicated processing such as buffer management and sleep to provide higher level functions such as an animation collaboration and file input/output. Instead of providing such higher level functions, high_level has certain limitations in terms of customization and internal accessibility.

Note
This document mainly explains the low level APIs. High level APIs will be provided as tutorials. For details, refer to the "Physics Effects High Level API Tutorial" document.

Embedding in Program

The following files are required to use the Physics Effects library.

File Name	Description
physics_effects.h	Header file for Physics Effects API
libScePhysicsEffects.a	Static link library

Include physics_effects.h into a source program. When building a program, link libScePhysicsEffects.a The Physics Effects library uses libult and vectormath libraries.

Name Space

All of the APIs of the Physics Effects library are contained in the `sce::PhysicsEffects` name space.

Main Functions

The following are the functions that the Physics Effects library provides.

- Rigid Body Simulation
- Raycast

The shapes handled as a rigid body are geometric shapes which can be represented by parameters (sphere, box, capsule and cylinder), convex mesh, large mesh for landscapes and combinations of these shapes (compound shapes). To represent an object with joints, connect multiple rigid bodies by joints.

Resource Consumption

In general, the Physics Effects library does not allocate memory internally. The library always utilizes a buffer allocated externally. An application must prepare required data and work buffer for the library in advance and pass them to the API upon execution.

Embedding in Program

Include physics_effects.h into a source program.

For multithread mode, libult is used to manage a thread. Refer to "libult Overview" and "libult Reference" documents in order to link required libraries.

Sample Programs

Building a Simple Rigid Body Simulation Pipeline

samples/sample_code/engines/api_physics_effects/1_simple/

This simulation pipeline focuses more on performance than on stability. The pipeline is simplified by omitting several stages and not maintaining data cached between frames so that the performance is ensured while the stability is sacrificed. This pipeline isn't suitable for stacking rigid bodies or a scene with complex gimmick.

Building a Stable Rigid Body Simulation Pipeline

samples/sample_code/engines/api_physics_effects/2_stable/

This pipeline realizes high stability by caching the result of collisions and solvers in the previous frame to utilize them for the calculations in the current frame. The precision can be further increased by making a time step smaller and increasing the number of solver iterations. This pipeline provides high versatility and can be applied to a wide range of use.

Incorporating the Sleep Mechanism Using Simulation Islands

samples/sample_code/engines/api_physics_effects/3_sleep/

For an application handling a large world, it is often desirable to stop physics calculations for an area never come into a player's view. In this case, the sleep mechanism is useful. In this sample program, a function to transition the rigid bodies that do not move for a certain period to sleep mode and to cancel calculations of collision detection and constraint solver, which require an extremely high calculation load, is added to a stable pipeline.

Differences in Behavior by Motion Type

samples/sample_code/engines/api_physics_effects/4_motion_type/

In this sample program, you can see the differences in behavior of various actions which can be given to a rigid body. In the first scene, a rigid body having behavior of `kPfxMotionTypeKeyframe` is arranged, and the application feed the orientation per frame. The changes in the orientation are held as velocity, so it is possible to see that the natural interference with other rigid bodies is reproduced. In the second scene, the differences between two types of rigid bodies, `kPfxMotionTypeOneWay` and `kPfxMotionTypeActive`, can be seen. When a collision occurs, a rigid body whose motion type is `kPfxMotionTypeOneWay` cannot give any impact to other rigid bodies in contrast with `kPfxMotionTypeActive`. In the third scene, a rigid body having behavior of `kPfxMotionTypeTrigger` is arranged in the center. The trigger rigid body is only used for checking the interference and never shows physical behavior.

Raycast

samples/sample_code/engines/api_physics_effects/5_raycast/

In this sample program, raycasts are executed to a scene. A ray detects the intersection closest to the start point and returns the coordinates and the normal vector of the intersection point. In the third scene, a method is provided for specifying contact filters to the ray and then identifying the rigid bodies that intersect with the ray.

Joint

samples/sample_code/engines/api_physics_effects/6_joint/

In the first scene, provided basic joints are introduced. You can move the rigid bodies by actually dragging them and can see the behavior. In the second scene, ragdolls often used in games are reproduced. Each joint of rigid bodies that make up the ragdoll has a restricted range of movement in order to prevent joints from being bent in the reverse direction. Therefore, it is possible to make sure that no unnatural orientations are produced.

Serialization

samples/sample_code/engines/api_physics_effects/7_serialize/

This sample program illustrates the way to write/read physics simulation instances into/from a file.

Dynamic Mesh Deformation

samples/sample_code/engines/api_physics_effects/8_deform_mesh

This sample realizes dynamic mesh deformation by moving the vertex coordinates of a large mesh. The mesh is deformed upon completion of the simulation, and the contact points locations on the deformation triangle at the next frame are calculated using the barycentric coordinates. To prevent collision tunneling, a largish thickness is set for the triangle, and if as the result of the deformation the contact points move in a direction that results in deeper penetration, the position of the rigid body is newly set again on the triangle.

Particle Physics

samples/sample_code/engines/api_physics_effects/9_particle_physics

This sample simulates particles. Distance and angle constraints can be specified between particles, and various soft bodies such as cloth and human hair can be simulated through their combination. Simple implementation that resolves constraint on a location base is achieved. For particle related implementation details, refer to pfx_position_based_physics.cpp/.h.

Character Controller

samples/sample_code/engines/api_physics_effects/10_character_ctrl

This is a sample of a character controller that combines rigid body physics and a finite state machine. In a world inhabited by many different types of rigid bodies, characters can be freely moved and made to physically interfere with one another.

XML File Import

samples/sample_code/engines/api_physics_effects/11_xml_import

This sample loads XML files to build scenes.

Tutorial

samples/sample_code/engines/tutorial_physics_effects_high_level/

This is a tutorial that includes the source code of high level APIs. It groups into a class library the general-purpose functions handled as a stand-alone physics engine based on the Physics Effects library. For details, refer to the "Physics Effects High Level API Tutorial" document.

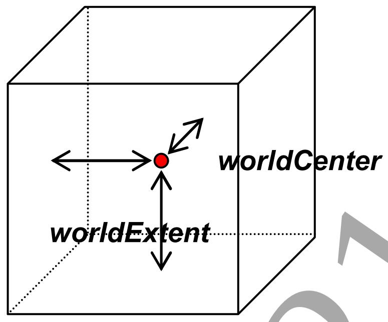
2 Architecture

Representing Objects

Methods for representing objects provided by the Physics Effects library are described below.

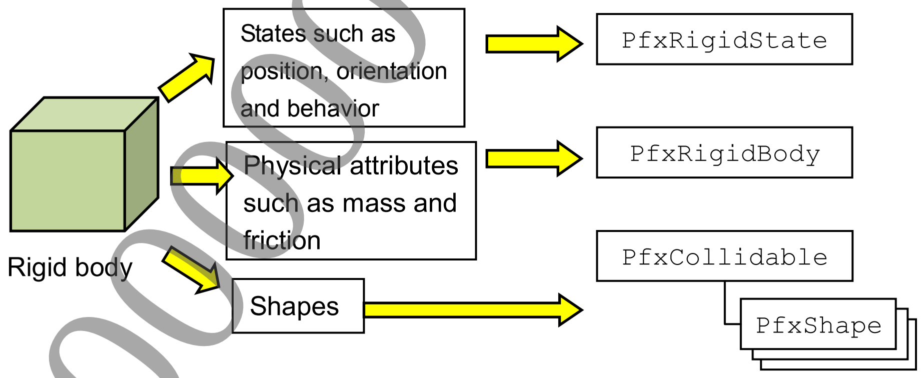
World

To determine the world size, specify the center point and the extent. The calculation precision will deteriorate if the world size is too large.



Rigid Body

Three data structures, `PfxRigidState`, `PfxRigidBody`, and `PfxCollidable`, are used to represent a rigid body. One index is allocated to these three data structures, and the structures are handled as one rigid body. All of `PfxRigidState`, `PfxRigidBody`, and `PfxCollidable` are containers. `PfxRigidBody` contains physical attributes such as friction, restitution and mass, `PfxRigidState` contains states such as position, orientation and velocity, and `PfxCollidable` contains collision shapes. The stages that comprise a simulation pipeline process only the required data among these three structures.


States such as position, orientation and behavior
Physical attributes such as mass and friction
Shapes
Rigid body
PfxRigidState
PfxRigidBody
PfxCollidable
PfxShape

Shape

`PfxShape` is a data structure that contains a single shape. Geometric shapes (sphere, box, etc.) do not need any additional data because they can be represented using parameters. Regarding convex mesh or large mesh, however, the size is too large because they are made up by multiple vertices and triangles, so the `PfxShape` contains only the pointers to the mesh structure data. Thus, the application side is responsible for the memory management of the mesh.

Joint

A complex mechanism such as a rag doll and a gear can be created by combining multiple joints (each joint connects two rigid bodies). A joint has six constrains (`PfxJointConstraint`), and various kinds of joints can be represented by the combinations of constraining axes.

Collision Information

When a collision between two rigid bodies is detected, the collision information is stored in `PfxContactManifold`. The constraint solver calculates an appropriate restitution force based on the collision information and separates the two collided rigid bodies to prevent interpenetration. One `PfxContactManifold` can contain up to four contact points (`PfxContactPoint`). Each contact point is made up by coordinates the collision has occurred, the depth of the penetration and restitution vectors. If five or more contact points are detected, four contract points including the contact point whose depth of the penetration is the deepest among those contact points are selected so that the area generated by the four contact points is maximized.

Ray

Raycast is mainly utilized for grasping the positions of rigid bodies in the world. A ray is represented as a direction vector having the start point and length. `PfxRayInput` is passed as an input parameter of raycast, and the result is returned to `PfxRayOutput`. When intersections detecting, the information on the coordinates closest to the start point among the intersections and the relevant rigid bodies are set to `PfxRayOutput`.

Pair

As for the Physics Effects libraries, the relation between two rigid bodies is important for many processing. To represent such relation efficiently, data structures (`PfxBroadphasePair`, `PfxConstraintPair`) representing a pair are used. `PfxBroadphasePair` represents two rigid bodies the bounding volumes are intersecting, and `PfxConstraintPair` represents two rigid bodies constrained by collisions or joints, etc. The processing can be performed efficiently without referring to any actual rigid body data by storing required information in a compact structure. In addition, the data of `PfxBroadphasePair` and `PfxConstraintPair` are compatible, so any of the APIs can share the data.

Broad Phase Proxy

For the broad phase, a simplified data structure, `PfxBroadphaseProxy`, is used to detect an intersection between two rigid bodies. As for `PfxBroadphaseProxy`, one proxy corresponds to one rigid body, and the data structure contains information such as bounding box, contact filter and motion type. The detected result is output as `PfxBroadphasePair`, and, in general, it is used as an input parameter to calculate further detailed information on the collision at the next collision detection stage.

Solver Body

A solver body (`PfxSolverBody`) is the dedicated data structure for a constraint solver which corresponds to one rigid body. The constraint solver repeatedly executes the calculation loop. The calculation efficiency can be improved by copying only the required data for the calculation from the rigid body to the solver body in advance.

Simulation Island

Simulation island (`PfxIsland`) manages a group of rigid bodies connected by collisions and joints and is mainly used for realizing the sleep function.

Motion Type

A rigid body has five kinds of adaptable behaviors and can change the conditions used for determining the broad phase and constraint solver.

Note
The behaviors can be changed dynamically; however, the settings must be done so that a contradiction between the rigid bodies never arises.

Fixed (kPfxMotionTypeFixed)

A rigid body with a fixed behavior is always fixed in a space and is represented as a rigid body that has infinite mass. The fixed rigid body is never affected by the other rigid bodies.

Active (kPfxMotionTypeActive)

By default, a normal rigid body is treated as being active and is affected by collisions and restitution from other rigid bodies. The active rigid body is able to transition to the sleep mode.

Key Frame (kPfxMotionTypeKeyframe)

A rigid body having the key frame behavior is given a position and an orientation after the time step externally. A key frame rigid body can affect the other rigid bodies, but, on the other hand, the key frame rigid body itself is never affected by the other rigid bodies. It is mainly used for representing a rigid body driven by the animation, etc. The key frame rigid body is able to transition to the sleep mode.

One Way (kPfxMotionTypeOneWay)

The basic properties of this rigid body are the same as those of an active rigid body. This rigid body never affects the other rigid bodies irrespective of their behavior while it is affected by the other rigid bodies. In other words, when a collision with the other rigid bodies occurs, only the one way rigid body receives the restitution force and rebounds off. It is mainly used for representing a camera, etc. and is able to transition to the sleep mode.

Trigger (kPfxMotionTypeTrigger)

The trigger rigid body is never affected by or never affects the other rigid bodies, but it can acquire the collision information. That is, although the collision detection is performed, there is no restitution given by a constraint solver. By placing the trigger rigid body in a space, for example, it can be used as a trigger to make an event occur when a specified object coming into contact.

Sleep Mode

Sleep serves as a supplementary behavior. Several behaviors can transition to the sleep mode so that they can cancel the processing which requires a high load. For example, if two active rigid bodies transitioned to the sleep mode after a collision, the collision detection and constraint solver processing between the pair will not be performed until the two rigid bodies return to the active type again.

Note
The low level API does not include the function that automatically moves a rigid body to the sleep mode. Please move a rigid body to the sleep mode by judging the conditions or timing on the application side. Refer to the sample program that describes an example for implementing sleep function using a simulation island.

Contact Filter

Two 32-bit masks, `ContactFilterSelf` (own contact filter) and `ContactFilterTarget` (target contact filter), can be set to a rigid body state and a ray. The result of the bit mask logic operation can switch between valid and invalid settings of collision or intersection detection.

For example, in the case that the contact filters of rigid body A are *`SelfA`* and *`TargetA`*, and the those of rigid body B are *`SelfB`* and *`TargetB`*, if the value of `(SelfA&TargetB)&&(TargetA&SelfB)` is false, the detection will not be performed. (This pair will be excluded in the broad phase.)

Type of Shape

Geometric Shapes

Geometric shapes can be represented using several parameters. Both the calculation costs and data consumption volumes are low. When handling a complex shape, represent it approximately by combining multiple geometric shapes.

Sphere (PfxSphere)

Sphere shape is represented only by radius.

Box (PfxBox)

Box shape is represented by the size in the XYZ axes direction.

Cylinder (PfxCylinder)

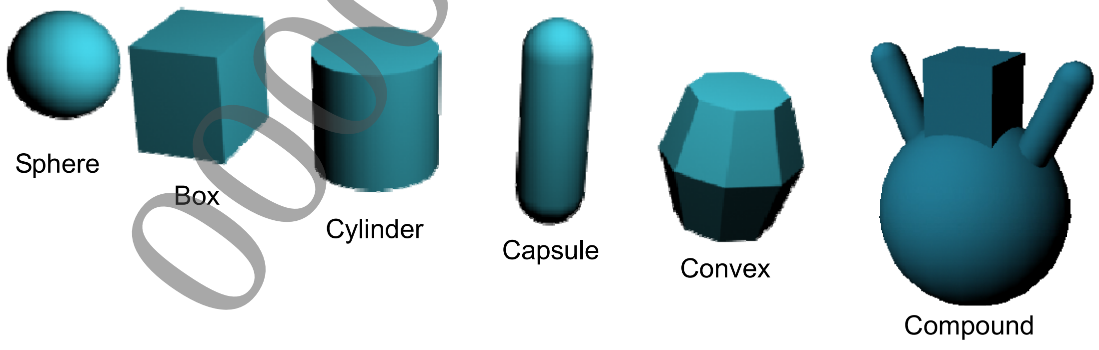
Cylinder shape is represented by the length in the X axis direction and radius.

Capsule (PfxCapsule)

Capsule shape is represented by the length in the X axis direction and radius.

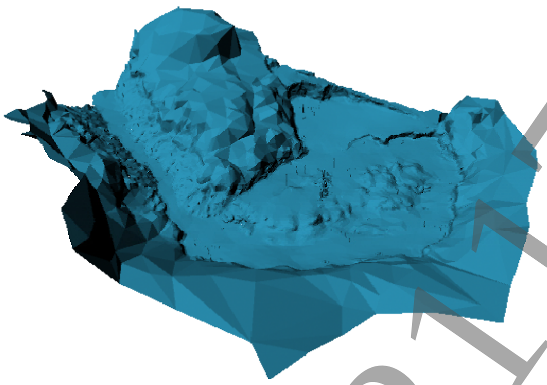
Convex Mesh (PfxConvexMesh)

A convex mesh consists of vertices and triangles. Make sure to create the mesh so that there is no concaved part. Pass the buffer of vertices and triangles to a utility function, `pfxCreateConvexMesh()`, to create a convex mesh.



Large Mesh (PfxLargeTriMesh)

A large mesh consists of vertices and triangles as in the case of a convex mesh, but it is not necessary to be convex-shaped, and the large mesh can be applied to a static and large-sized shape. The large mesh is represented as a collection of small divided meshes (islands), and for an actual detection, searching is performed hierarchically, like large mesh -> island -> triangle. Use a utility function, `pfxCreateLargeTriMesh()`, to create a large mesh. `pfxCreateLargeTriMesh()` allocates buffers to store mesh information internally. When the buffers become unnecessary, please make sure to free the allocated buffers by calling `pfxReleaseLargeTriMesh()`.



Large mesh

Note
There is a limitation on the number of triangles a mesh can hold. The maximum number of triangles in a convex mesh is 64. A large mesh can hold up to 512 islands, while an island can hold up to 64 triangles, so the maximum number of triangles in a large mesh is512*64=32768. If you handle a larger mesh, divide the mesh into multiple rigid bodies and then create a large mesh.

Whether to use the array or BVH (Bounding Volume Hierarchy) for the storage structure of large mesh islands can be selected. By default, the array is selected, but BVH can also be selected by specifying `SCE_PFX_MESH_FLAG_USE_BVH` for *`flag`* of large mesh creation parameter `PfxCreateLargeTriMeshParam`. BVH is used for efficiently searching a large number of islands, but an additional buffer is required to some extent in order to maintain the BVH structure. When additionally specifying `SCE_PFX_MESH_FLAG_HIGH_QUALITY`, an algorithm to build a more efficient array or BVH can be used; however, note that this will entail a longer building time than normal.

Moreover, the size can be reduced by half by compressing the elements (vertex, normal, triangle) that constitute large meshes. By specifying `SCE_PFX_MESH_FLAG_USE_QUANTIZED` for *`flag`* of large mesh creation parameter `PfxCreateLargeTriMeshParam`, compression is done by converting the floating-point values of the various elements into 16-bit integers through quantization. Because the compressed data is expanded in the ultimate phase of collision detection, the additional processing cost overhead is minimized.

Large meshes are classified into the following four types according to the island storage structure and the data compression method.

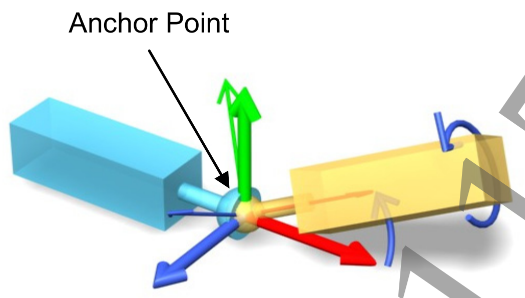
	No compression	Compression
Array	`SCE_PFX_LARGE_MESH_TYPE_EXPANDED_ARRAY`	`SCE_PFX_LARGE_MESH_TYPE_QUANTIZED_ARRAY`
BVH	`SCE_PFX_LARGE_MESH_TYPE_EXPANDED_BVH`	`SCE_PFX_LARGE_MESH_TYPE_QUANTIZED_BVH`

Type of Joint

All joints are represented using the same data structure, `PfxJoint`. Various types of joints can be created by adjusting six constraint parameters included in `PfxJoint`. An application can create joint types or change the joints dynamically just using the joint initialization function. To adjust the joints in detail, refer to the "Mechanism of a Joint Constraint" section in the "10 Reference Information" chapter.

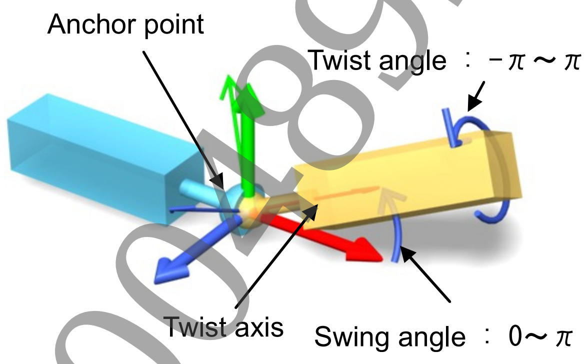
Ball Joint

A ball joint connects two rigid bodies at one specified point. The rigid bodies are able to rotate freely.



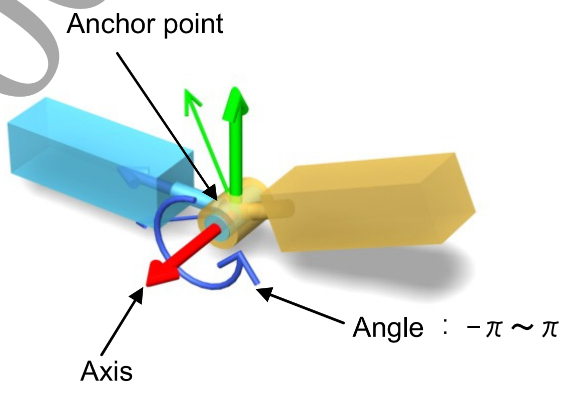
Swing Twist Joint

This joint has a cone-shaped movement range.



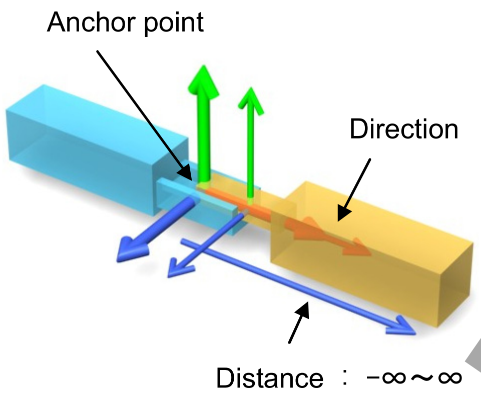
Hinge Joint

A hinge joint reproduces the behavior like a door hinge.



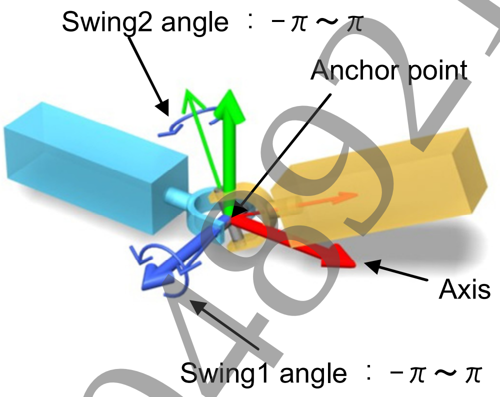
Slider Joint

A slider joint allows movement only on the specified axis.



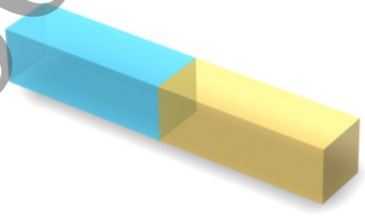
Universal Joint

A universal joint has two swing rotation axes.



Fixed Joint

A fixed joint is used to fix two rigid bodies.



3 Usage Procedure

About Types

The unique types for the Physics Effects library are redefined as follows.

Redefined Types	Original Types	Description
`PfxInt8`	`int8_t`	Standard type
`PfxUInt8`	`uint8_t`	
`PfxInt16`	`int16_t`	
`PfxUInt16`	`uint16_t`	
`PfxInt32`	`int32_t`	
`PfxUInt32`	`uint32_t`	
`PfxInt64`	`int64_t`	
`PfxUInt64`	`uint64_t`	
`PfxBool`	`bool`	
`PfxFloat`	`float`	
`PfxPoint3`	`sce::Vectormath::Scalar::Aos::Point3`	Type of Vector Math library
`PfxVector3`	`sce::Vectormath::Scalar::Aos::Vector3`	
`PfxVector4`	`sce::Vectormath::Scalar::Aos::Vector4`	
`PfxQuat`	`sce::Vectormath::Scalar::Aos::Quat`	
`PfxMatrix3`	`sce::Vectormath::Scalar::Aos::Matrix3`	
`PfxMatrix4`	`sce::Vectormath::Scalar::Aos::Matrix4`	
`PfxTransform3`	`sce::Vectormath::Scalar::Aos::Transform3`	
`PfxFloatInVec`	`sce::Vectormath::Scalar::floatInVec`	
`PfxBoolInVec`	`sce::Vectormath::Scalar::boolInVec`	

Initialize the Task Manager

The task manager divides processing into multiple tasks and provides a function for executing the tasks in parallel. Specify the task manager as an argument of the API that supports parallel processing upon execution. This initialization is not necessary if the multithread mode is not used.

(1) Preparation

First, load and initialize the PRX of libult.

```
sceSysmoduleLoadModule( SCE_SYSMODULE_ULT );
```

(2) Create a Task Manager

Use `PfxPsp2TaskManager::getWorkBytes()` to obtain the number of bytes of a work buffer required for a task manager and allocate the work buffer. Then, create a task manager by specifying the work buffer and the maximum number of tasks to a constructor. Do not destroy the work buffer until after using the task manager.

```
#define MAXTASKS 3
PfxUInt32 workBytes = PfxPsp2TaskManager::getWorkBytes(MAXTASKS);
void *workBuff = malloc(workBytes);
PfxTaskManager *taskManager = new PfxPsp2TaskManager(
        MAXTASKS,MAXTASKS,workBuff,workBytes);
taskManager->initialize();
```

(3) Destroy the Task Manager

Call the task manager termination method to free the task manager and the allocated work buffer.

```
taskManager->finalize();
delete taskManager;
free(workBuff);
```

Finally, unload the PRX of libult.

```
sceSysmoduleUnloadModule( SCE_SYSMODULE_ULT );
```

Create a Rigid Body

Prepare an array for storing the rigid body data. Initialize each object and relate the objects.

(1) Prepare Rigid Body Arrays

Prepare the arrays of three data structure required for representing a rigid body (`PfxRigidState`, `PfxRigidBody` and `PfxCollidable`) as many as the number of the rigid bodies.

```
#define NUM_RIGIDBODIES 10
PfxRigidState states[NUM_RIGIDBODIES];
PfxRigidBody  bodies[NUM_RIGIDBODIES];
PfxCollidable collidables[NUM_RIGIDBODIES];
```

(2) Create Rigid Body Shapes

An example of creating a rigid body, a sphere of radius 1, is shown below. First, create `PfxShape` to represent a sphere and register it to `PfxCollidable`. After the registration, call `PfxCollidable::finish()`, and notify that the registration has completed. The contents of `PfxSphere` and `PfxShape` used here are copied to the internal data of `PfxCollidable` immediately.

```
PfxSphere sphere(1.0f); // Create a sphere of radius 1
PfxShape shape;
shape.reset();
shape.setSphere(sphere);// Allocate the sphere to PfxShape
collidables[id].reset();
collidables[id].addShape(shape);
collidables[id].finish(); // Complete the registration of the shape
```

Next, an explanation on complex shape is provided below. `PfxCollidable` can contain references for up to 64 shapes. When the number of the shapes is 1, a copy is created internally. If the number of the shapes is 2 or more, the external references are used. Thus, an array of shape must be prepared before creating a complex shape.

```
PfxShape shapes[100]; // shape array used for a complex shape
```

Then, among the shape arrays, prepare an array to store the index to be used and initialize `PfxCollidable`.

```
PfxUInt16 shapeIds[3]={0,1,2}; // 0, 1 and 2 of the shape arrays are to be used
collidables[id].reset(shapes,shapeIds,3); // This PfxCollidable can contain up
to three shapes
```

Register shapes multiple times in the same way as described above, and finally, call `PfxCollidable::finish()` to notify that the registrations have completed.

```
{
        PfxBox box(0.5f,0.5f,1.5f);
        PfxShape shape;
        shape.reset();
        shape.setBox(box);
```

```
                shape.setOffsetPosition(PfxVector3(-2.0f,0.0f,0.0f));
                collidables[id].addShape(shape);
        }
        {
                PfxBox box(0.5f,1.5f,0.5f);
                PfxShape shape;
                shape.reset();
                shape.setBox(box);
                shape.setOffsetPosition(PfxVector3(2.0f,0.0f,0.0f));
                collidables[id].addShape(shape);
        }
        {
                PfxCapsule cap(1.5f,0.5f);
                PfxShape shape;
                shape.reset();
                shape.setCapsule(cap);
                collidables[id].addShape(shape);
        }
        collidables[id].finish();
```

(3) Create Rigid Body Attributes

Specify physical attributes to a rigid body. Calculate the mass and inertia tensor using utility functions.

```
bodies[id].reset();
bodies[id].setMass(1.0f); // Set the mass to 1kg
bodies[id].setInertia(pfxCalcInertiaSphere(1.0f,1.0f)); // Calculate the
inertia tensor of a sphere with a mass of 1kg
```

(4) Create Rigid Body States

Specify the position, velocity, behavior, etc. of the rigid body to the rigid body states. Finally, set the index of the rigid body.

```
states[id].reset();
states[id].setPosition(PfxVector3(-5.0f,5.0f,0.0f));
states[id].setMotionType(kPfxMotionTypeActive);
states[id].setRigidBodyId(id); // Set the index of the rigid body
```

Note
The same index manages all of the shapes, attributes, and states of a rigid body.

Create a Joint

(1) Prepare a joint array

Prepare an array to store joint data.

```
#define NUM_JOINTS 5
PfxJoint joints[NUM_JOINTS];
```

(2) Create a Joint

An example of procedures to create a ball joint, which connects two rigid bodies at one specified point, is described below.

The rigid body states of the already created two rigid bodies to be connected are *`stateA`* and *`stateB`*, respectively.

Specify the center point of each rigid body to the joint initialization parameter `PfxBallJointInitParam` as the connection points.

```
PfxVector3 anchor = ( stateA.getPosition() + stateB.getPosition() ) * 0.5f;

PfxBallJointInitParam jointParam;
jointParam.anchorPoint = anchor;
```

Call the joint initialization function to initialize the joint structure as a ball joint.

Note
The initialization parameter and function are provided for each type of joint.

```
pfxInitializeBallJoint(joints[id],stateA,stateB,jointParam);
```

Create a pair structure corresponds to the joints in advance. This pair structure will be used for a constraint solver later.

```
pfxUpdateJointPairs(jointPairs[id],id,joints[id],stateA,stateB);
```

How to More Precisely Control Joints

Upon initializing a joint, the behavior of a joint can be adjusted by specifying additional parameters to the joint initialization parameter.

- *damping*
  This parameter is used to add friction to the movable direction and provide resistance to the movement. Specify within a range of 0.0 - 1.0.
- *bias*
  This parameter adjusts the rigidity of a joint (strength of constraint required to return the orientation to within the movable range). Specify within a range of 0.0 - 1.0.
- *jointFrame*
  This parameter specifies the joint coordinate system directly in a 3 x 3 matrix. To enable this parameter, set `true` to *enableJointFrame*. The use of *jointFrame* will be prioritized and the axis specification parameter of a joint (*direction*, *axis*, *twistAxis*, and *upVec*) will be ignored.

4 Stages

Simulation Stages

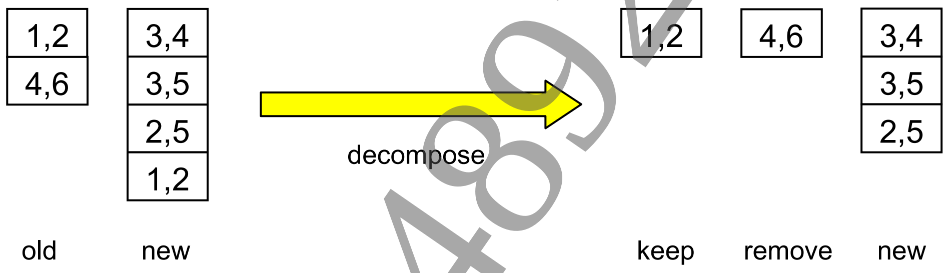
In the Physics Effects library, a simulation pipeline composes of multiple processing stages as described below. The library is designed to efficiently reproduce dynamically natural behaviors by setting the data to this series of stages in an appropriate order. Also, a mechanism that the each individual stage can be independently handled is provided so that the user can build pipelines at will.

(1) Apply an External Force

Apply the force and gravity given by an application to a rigid body as an external force, and convert it to a velocity.

(2) Broad Phase

In this stage, to omit unnecessary collision calculation, the two rigid bodies whose AABBs (Axis Aligned Bounding Box) intersect each other are detected as a pair having a strong possibility to collide. Moreover, by comparing with the result of the previous frame, it is determined whether the pair continues or new pairs have been added or whether any pairs have been removed. The result will be returned as pair arrays.



(3) Collision Detection

The pair having a strong possibility to collide output from the broad phase is tested for collisions with actual shapes. If intersections are detected, more detailed collision information is calculated, and then the coordinates of the point of collision, the depth of the penetration and restitution vectors are set to `PfxContactManifold` structure.

(4) Constraint Solver

Constraint solver calculates the restitution force of the colliding two rigid bodies and the constraint force of the rigid bodies connected by joints. The calculation is repeated for the specified number of times to get close to the optimal solution and the constraint force is eventually reflected in the rigid bodies as a change of velocity.

(5) Position Calculation

The position of the rigid body is updated using the velocity calculated by the constraint calculation.

Apply an External Force

`pfxApplyExternalForce()` applies an external force to a rigid body and updates the velocity of the rigid body. This function does not affect a rigid body having behaviors that are never affected by the external force or a sleeping rigid body.

```
PfxVector3 gravityAccel = PfxVector3(0.0f,-9.8f,0.0f); // acceleration of
gravity
for(int id=0;id<NUM_RIGIDBODIES;id++) {
        PfxVector3 externalForce = bodies[id].getMass() * gravityAccel;
        PfxVector3 externalTorque = PfxVector3(0.0f);
        pfxApplyExternalForce(
             states[id],bodies[id],externalForce,externalTorque,0.016f);
}
```

Broad Phase

(1) Decide the Axis to be Tested

Select the axis from XYZ to perform intersection detection of AABBs. The selection method is not specified. The following example describes a method to select axes where the rigid bodies are distributed the most.

```
PfxVector3 s(0.0f),s2(0.0f);
for(int id=0;id<NUM_RIGIDBODIES;id++) {
        PfxVector3 c = states[id].getPosition();
        s += c;
        s2 += mulPerElem(c,c);
}
PfxVector3 v = s2 - mulPerElem(s,s) / (PfxFloat)NUM_RIGIDBODIES;
int axis = 0; // X
if(v[1] > v[0]) axis = 1; // Y
if(v[2] > v[axis]) axis = 2; // Z
```

(2) Update a Broad Phase Proxy

Allocate broad phase proxy arrays as many as the number of rigid bodies in advance. Determine the size of the world and update all arrays by `pfxUpdateBroadphaseProxy()`. The broad phase proxy stores information on AABB, ID, behaviors and contact filter of a rigid body. When the rigid body is placed outside the world, `SCE_PFX_ERR_OUT_OF_WORLD` will be returned.

Note
The broad phase proxy is created even when `SCE_PFX_ERR_OUT_OF_WORLD` is returned. The handling of the rigid body placed outside the world should be determined by the application side.

```
PfxVector3 worldCenter(0.0f);
PfxVector3 worldExtent(500.0f);
PfxBroadphaseProxy proxies[NUM_RIGIDBODIES];
for(int id=0;id<NUM_RIGIDBODIES;id++) {
        pfxUpdateBroadphaseProxy(
             proxies[id],states[id],collidables[id],
             worldCenter,worldExtent,axis);
}
```

Sort the updated broad phase proxy arrays along the selected axis. Specify the work buffer that is 16-byte aligned and is the same size as the sort target to `pfxParallelSort()`.

```
int workBytes = sizeof(PfxBroadphaseProxy)*NUM_RIGIDBODIES;
void *workBuff = memalign(16,workBytes);
pfxParallelSort(proxies,NUM_RIGIDBODIES,workBuff,workBytes);
free(workBuff);
```

Note
Proxy arrays can be created simultaneously for multiple axes by using the `pfxUpdateBroadphaseProxies()` function. For details, refer to the "(1) Update a Broad Phase Proxy" section in the chapter 5 "Raycast".

(3) Detect an Intersecting Pair

Test all broad phase proxies for intersection detection using the bounding box and detect intersecting pairs.

The required work buffer size can be acquired by specifying the maximum number of pairs to be detected, and, in the case of parallel processing, the number of the tasks to `pfxGetWorkBytesOfFindPairs()` and then calling the function. The size of the pair buffer can be acquired by specifying the maximum number of pairs to be detected to `pfxGetPairBytesOfFindPairs()`. Make sure to allocate the buffer so that the 16-byte alignment is guaranteed.

```
#define MAX_PAIRS 2000 // The maximum number of the pairs to be detected
int workBytes = pfxGetWorkBytesOfFindPairs(MAX_PAIRS);
void *workBuff = malloc(workBytes);
int pairBytes = pfxGetPairBytesOfFindPairs(MAX_PAIRS);
void *pairBuff = memalign(16,workBytes);
```

Set the work buffer, pair buffer and broad phase proxy arrays sorted along the detecting axis to `PfxFindPairsParam`, which is an input parameter passed to `pfxFindPairs()`. After executing `pfxFindPairs()`, the number of the detected intersecting pairs and the address of the arrays are returned to `PfxFindPairsResult`. If the number of the detected pairs exceeds the maximum number of the pairs, `SCE_PFX_ERR_OUT_OF_MAX_PAIRS` will be returned. In that case, execute `pfxFindPairs()` again after increasing the size of the pair buffer for output. Note that the intersecting pair arrays are sorted by a unique value generated from the indexes of the two rigid bodies.

```
PfxFindPairsParam findPairsParam;
findPairsParam.workBuff = workBuff;
findPairsParam.workBytes = workBytes;
findPairsParam.pairBuff = pairBuff;
findPairsParam.pairBytes = pairBytes;
findPairsParam.proxies = proxies;
findPairsParam.numProxies = NUM_RIGIDBODIES;
findPairsParam.maxPairs = MAX_PAIRS;
findPairsParam.axis = axis;

PfxFindPairsResult findPairsResult;

pfxFindPairs(findPairsParam,findPairsResult);
```

(4) Compose Intersecting Pairs

In order to maintain information of pairs between frames, compare the result of the previous frame with the newly detected pair arrays and then compose them. Calculate the required buffer size based on the number of pairs of the previous frame and the number of the newly detected pairs and allocate the buffer. Make sure that the pair buffer is 16-byte aligned.

```
int workBytes = pfxGetWorkBytesOfDecomposePairs(
        numPreviousPairs,findPairsResult.numPairs);
void *workBuff = malloc(workBytes);

int pairBytes = pfxGetPairBytesOfDecomposePairs(
        numPreviousPairs,findPairsResult.numPairs);
void *pairBuff = memalign(16,workBytes);
```

Set the required parameters to `PfxDecomposePairsParam` and call `pfxDecomposePairs()`. Set the pairs of the previous frame to *`previousPairs`* and newly detected pairs obtained by `pfxFindPairs()`

to *currentPairs*. Three pair arrays (arrays for pairs newly intersected, pairs the intersection is being kept, pairs not intersected) are returned to `PfxDecomposePairsResult` as the result. These arrays are built from the pair buffer of an input parameter. Do not destroy the arrays until the arrays become unnecessary.

```
PfxDecomposePairsParam decomposePairsParam;
decomposePairsParam.workBuff = workBuff;
decomposePairsParam.workBytes = workBytes;
decomposePairsParam.pairBuff = pairBuffer;
decomposePairsParam.pairBytes = pairBytes;
decomposePairsParam.previousPairs = previousPairs;
decomposePairsParam.numPreviousPairs = numPreviousPairs;
decomposePairsParam.currentPairs = findPairsResult.pairs;
decomposePairsParam.numCurrentPairs = findPairsResult.numPairs;

PfxDecomposePairsResult result;

pfxDecomposePairs(decomposePairsParam,decomposePairsResult);
```

Based on the result returned from `pfxDecomposePairs()`, establish a link (or destroy the link) between pairs and `PfxContactManifold` that the result of collision detection.

Set the index of collision information to the new pairs.

```
for(int i=0;i<result.numOutNewPairs;i++) {
        // Obtain the index of the new collision information
        // ...
        pfxSetContactId(result.outNewPairs[i],contactId);
}
```

Regarding the pairs to be destroyed, obtain the allocated index of collision information and destroy the information.

```
for(int i=0;i<result.numOutRemovePairs;i++) {
        PfxUInt32 contactId = pfxGetContactId(result.outRemovePairs[i]);
        // Destroy the collision information of the obtained index
        // ...
}
```

Finally, compose the new pairs and the continuing pairs and sort them. These pair arrays are used throughout the pipeline, so it is necessary to hold the arrays during the use of simulation.

Note
It is possible to easily manage these two arrays by using them alternately. Refer to the sample code.

```
PfxBroadphasePair *currentPairs = memalign(
        16,sizeof(PfxBroadphasePair)*
        (result.numOutKeepPairs+result.numOutNewPairs));

int numCurrentPairs = 0
for(int i=0;i<result.numOutKeepPairs;i++) {
        currentPairs[numCurrentPairs++] = result.outKeepPairs[i];
}
for(int i=0;i<result.numOutNewPairs;i++) {
        currentPairs[numCurrentPairs++] = result.outNewPairs[i];
}

int workBytes = sizeof(PfxBroadphasePair)*numCurrentPairs;
void *workBuff =  memalign(16,workBytes);
pfxParallelSort(currentPairs,numCurrentPairs,workBuff,workBytes);
free(workBuff);
```

Collision Detection

(1) Collision Detection

The required data for performing collision detection one time are the data of the two rigid bodies and the collision information. `pfxDetectCollision()` executes collision detection based on the pair arrays output by the broad phase. The function obtains the required data from the offset address and reference index of each data and executes collision detection in succession.

The type of pair in the input parameter is `PfxConstraintPair`, however `PfxBroadphasePair` can be directly passed to it. The result of collision detection is stored to `PfxContactManifold`.

```
PfxDetectCollisionParam param;
param.contactPairs = currentPairs;
param.numContactPairs = numCurrentPairs;
param.offsetContactManifolds = contacts;
param.offsetRigidStates = states;
param.offsetCollidables = collidables;
param.numRigidBodies = NUM_RIGIDBODIES;

pfxDetectCollision(param);
```

(2) Update the Collision Information

While the collision continues, the contact point must be held between the frames. If two colliding rigid bodies are quite separate from each other in the restitution direction, the contact point must be removed. In order to determine the validity of the contact point, call `pfxRefreshContacts()`. A contact point determined as not being valid will be invalidated immediately.

```
PfxRefreshContactsParam param;
param.contactPairs = currentPairs;
param.numContactPairs = numNewPairs;
param.offsetContactManifolds = contacts;
param.offsetRigidStates = states;
param.numRigidBodies = NUM_RIGIDBODIES;

pfxRefreshContacts(param);
```

Constraint Calculation

(1) Update a Solver Body

First, allocate solver body arrays for storing required data for a solver as many as the number of rigid bodies.

```
PfxSolverBody solverBodies[NUM_RIGIDBODIES];
```

Then, set the offset address of the required data to `PfxSetupSolverBodiesParam`, call `pfxSetupSolverBodies()`, and reflect the required attributes for the solver in the solver body. The solver body can be destroyed after the solver calculation has completed.

```
PfxSetupSolverBodiesParam param;
param.states = states;
param.bodies = bodies;
param.solverBodies = solverBodies;
param.numRigidBodies = NUM_RIGIDBODIES;

pfxSetupSolverBodies(param);
```

(2) Set up Constraints

Before the solver calculation, set up constraints for both a collision and a joint. Specify pair arrays and offset addresses of required data to the input parameters, and then call the set up function.

Prepare the pair arrays to pass the joint to the constraint solver. Whenever the parameters of joint or the behaviors of rigid body change, the pair is updated.

```
#define NUM_JOINTS 10
PfxJoint joints[NUM_JOINTS];
PfxConstraintPair jointPairs[NUM_JOINTS];

// Update the joint pair
for(int i=0;i<NUM_JOINTS;i++) {
        pfxUpdateJointPairs(
              jointPairs[i],i,joints[i],
              states[joints[i].m_rigidBodyIdA],
              states[joints[i].m_rigidBodyIdB]);
}

// Set up the constraints on collision
PfxSetupContactConstraintsParam param;
param.contactPairs = currentPairs;
param.numContactPairs = numCurrentPairs;
param.offsetContactManifolds = contacts;
param.offsetRigidStates = states;
param.offsetRigidBodies = bodies;
param.offsetSolverBodies = solverBodies;
param.numRigidBodies = NUM_RIGIDBODIES;
param.timeStep = 0.016f;
param.separateBias = 0.2f;

pfxSetupContactConstraints(param);

// Set up the constraints on joint
PfxSetupJointConstraintsParam param;
param.jointPairs = jointPairs;
param.numJointPairs = NUM_JOINTS;
param.offsetJoints = joints;
param.offsetRigidStates = states;
param.offsetRigidBodies = bodies;
param.offsetSolverBodies = solverBodies;
param.numRigidBodies = NUM_RIGIDBODIES;
param.timeStep = 0.016f;

pfxSetupJointConstraints(param);
```

(3) Constraint Solver

Allocate the work buffer of the size obtained with `pfxGetWorkBytesOfSolveConstraints()`.

```
int workBytes = pfxGetWorkBytesOfSolveConstraints(
        NUM_RIGIDBODIES,numCurrentPairs,NUM_JOINTS);
void *workBuff = malloc(workBytes);
```

Set the already set up data to the parameters and call `pfxSolveConstraints()`. The calculated constraint force updates the velocity of the rigid body. By updating the position of rigid body using the velocity, the positioning that satisfies the constraint conditions is maintained.

Note
The calculation precision will increase as the calculation iteration count is increased, however, performance will drop as well.

```
PfxSolveConstraintsParam param;
param.workBuff = workBuff;
param.workBytes = workBytes;
param.contactPairs = currentPairs;
param.numContactPairs = numCurrentPairs;
param.offsetContactManifolds = contacts;
param.jointPairs = jointPairs;
param.numJointPairs = NUM_JOINTS;
param.offsetJoints = joints;
param.offsetRigidStates = states;
param.offsetSolverBodies = solverBodies;
param.numRigidBodies = NUM_RIGIDBODIES;
param.iteration = 5;

pfxSolveConstraints(param);
```

Position Calculation

`pfxUpdateRigidStates()` calculates the position of the rigid body after the time step using the velocity of the rigid body. This function does not affect a static rigid body.

```
PfxUpdateRigidStatesParam param;
param.states = states;
param.bodies = bodies;
param.numRigidBodies = NUM_RIGIDBODIES;
param.timeStep = 0.016f;

pfxUpdateRigidStates(param);
```

5 Raycast

Prepare a Raycast

(1) Update a Broad Phase Proxy

Prepare broad phase proxy arrays just as you would for the broad phase of the rigid body simulation. The difference from the rigid body simulation is to create broad phase proxy arrays corresponding to + and - directions of XYZ axes. In other words, it is necessary to prepare broad phase proxy arrays as many as the number of the rigid bodies x 6. Once you create these arrays, they can be reused because the rigid body simulation and the raycast can share the arrays.

The processing for rigid bodies outside the world area can be changed with the content of the flag specified for the *outOfWorldBehavior* parameter of the PfxUpdateBroadphaseProxiesParam structure. SCE_PFX_OUT_OF_WORLD_BEHAVIOR_FIX_MOTION forcibly changes the motion type of a rigid body to kPfxMotionTypeFixed. SCE_PFX_OUT_OF_WORLD_BEHAVIOR_REMOVE_PROXY does not perform registration to proxy arrays.

The pfxUpdateBroadphaseProxies() function sets the number of out-of-world rigid bodies as the return value to *numOutOfWorldProxies* of the PfxUpdateBroadphaseProxiesResult structure.

```
PfxBroadphaseProxy proxies[6][NUM_RIGIDBODIES];

PfxUpdateBroadphaseProxiesParam param;
param.workBytes = pfxGetWorkBytesOfUpdateBroadphaseProxies(NUM_RIGIDBODIES);
param.workBuff = memalign(16,param.workBytes);
param.numRigidBodies = NUM_RIGIDBODIES;
param.offsetRigidStates = states;
param.offsetCollidables = collidables;
param.proxiesX = proxies[0];
param.proxiesY = proxies[1];
param.proxiesZ = proxies[2];
param.proxiesXb = proxies[3];
param.proxiesYb = proxies[4];
param.proxiesZb = proxies[5];
param.worldCenter = worldCenter;
param.worldExtent = worldExtent;

PfxUpdateBroadphaseProxiesResult result;

pfxUpdateBroadphaseProxies(param,result);

free(param.workBuff);
```

(2) Create a Ray

Represent a ray using the PfxRayInput structure. Set the start coordinates of the ray to *m_startPosition*, and set the length and direction vector to *m_direction*. For example, when a line between two points, *p1* and *p2*, is treated as a ray, specify the value as described below.

```
PfxRayInput rayIn;
rayIn.reset(); // Set to the default value
rayIn.m_startPosition = p1;
rayIn.m_direction = p2-p1;
```

Raycast

Single Raycast

Specify the rigid body, shape, offset address of broad phase proxy arrays and the size of the world to the PfxRayCastParam structure. Pass the PfxRayCastParam structure together with the PfxRayOutput structure, which receives the result for each of the ray, to pfxCastSingleRay() as arguments, and execute the raycast.

Note
When executing a raycast from multiple threads, make sure that the parameters set to the PfxRayCastParam structure will not be changed during the execution.

```
PfxRayCastParam param;
param.offsetRigidStates = states;
param.offsetCollidables = collidables;
param.proxiesX  = proxies[0];
param.proxiesY  = proxies[1];
param.proxiesZ  = proxies[2];
param.proxiesXb = proxies[3];
param.proxiesYb = proxies[4];
param.proxiesZb = proxies[5];
param.numProxies = numRigidBodies;
param.rangeCenter = worldCenter;
param.rangeExtent = worldExtent;

PfxRayOutput rayOut;
pfxCastSingleRay(rayIn,rayOut,param);
```

Batch Raycast

Use the batch raycast when multiple rays are casted.

As same as for the single raycast, set the parameters to the PfxRayCastParam structure. Then pass the arrays of PfxRayInput and PfxRayOutput to pfxCastRays() as arguments and execute the raycast.

```
PfxRayInput rayIns[10];
PfxRayOutput rayOuts[10];

// Initialize a ray
// ...

PfxRayCastParam param;
param.offsetRigidStates = states;
param.offsetCollidables = collidables;
param.proxiesX  = proxies[0];
param.proxiesY  = proxies[1];
param.proxiesZ  = proxies[2];
param.proxiesXb = proxies[3];
param.proxiesYb = proxies[4];
param.proxiesZb = proxies[5];
param.numProxies = numRigidBodies;
param.rangeCenter = worldCenter;
param.rangeExtent = worldExtent;

pfxCastRays(rayIns,rayOuts,10,param);
```

6 Sort

The sort function `pfxParallelSort()` can perform sort processing quickly for the array of the sort data (`PfxSortData16` or `PfxSortData32`). In the Physics Effects library, sort data is used to represent the data structure that requires sorting.

When executing sort processing, specify the work buffer that is 16-byte aligned and is the same size as the sort target. The work buffer can be destroyed after the sort processing has completed.

```
int workBytes = sizeof(PfxSortData16)*NUM_DATA;
void *workBuff =  memalign(16,workBytes);
pfxParallelSort(sortDataArray,NUM_DATA,workBuff,workBytes);
free(workBuff);
```

7 Serialization Input/Output

Overview of Serialization Processing

Physics Effects supports the functions to input/output instances executing simulation from/into a file stream through serialization. This function is mainly intended for debugging use, for example analyzing output files on a viewer that runs on Windows.

The serialized data in a file can be read on a different platform on which Physics Effects is operated.

Callback Functions Called When Performing Serialization Processing

It is necessary to create callback functions in advance to initialize the data and terminate the processing required for serialization processing. There are five kinds of callback functions, namely, "initialization callback", "termination callback", "buffer resizing callback", "update callback" and "error handling callback". These callback functions are called at the appropriate timing. Register callback functions to the arguments of the serialize functions as needed, describing the effective processing on the application side.

Types of callback functions	Registration to `pfxSerializeRead()`	Registration to `pfxSerializeWrite()`
Initialization	Required	Required
Termination	Optional	Optional
Buffer resizing	Optional	None
Update	Optional	None
Error handling	Optional	Optional

Initialization Callback Function Type

```
typedef void(*PfxSerializeInitFunc)(
        PfxSerializeCapacity *capacity,
        PfxSerializeBuffer *buffer
);;
```

This function is called during serialization initialization. Set the size of each buffer to the `PfxSerializeCapacity` type variable and the pointer to the buffer being held by an application to the `PfxSerializeBuffer` type variable. Make sure that all the pointers set in the variables must be valid. During serialization input, data is loaded to the set buffers in sequence. During serialization output, the data that was saved to the set buffers is written out in sequence.

Termination Callback Function Type

```
typedef void(*PfxSerializeTermFunc)(
        PfxSerializeCapacity *capacity,
        PfxSerializeBuffer *buffer
);
```

This function is called at the end of serialization. The number of data loaded during serialization input is set to the `PfxSerializeCapacity` type variable.

Buffer Resizing Callback Function Type

```
typedef PfxBool(*PfxSerializeResizeFunc)(
        PfxSerializeCapacity *capacity,
        PfxSerializeBuffer *buffer,
        const PfxSerializeCapacity *capacityLoaded
);
```

When performing serialization input, this function is called if loading of data that exceeds the specified buffer size is attempted. The values specified from the application with the initialization callback function are set to *`capacity`* and *`buffer`*, and the data size that is actually loaded is set to *`capacityLoaded`*. Compare the values of *`capacity`* and *`capacityLoaded`*, reallocate the buffer shortfall to *`buffer`*, and return `true`. If `false` is returned, or this callback function has not been registered, reallocation of the buffer is considered to have failed, and the error handling callback function is called.

Update Callback Function Type

```
typedef void(*PfxSerializeUpdateFunc)(
        PfxUInt32 progress,
        PfxUInt32 maxProgress
);
```

While serialization input is performed, this function is called periodically and reports the progress status to the application.

Error Handling Callback Function Type

```
typedef void(*PfxSerializeErrorFunc)(
        PfxInt32 errorCode,
        const char **tags,
        int numTags
);
```

This function will be called if an error arises when performing serialization input/output. If an error code is set to *`errorCode`* and an error occurs during file analysis, the array of the tag character string for identifying the error occurrence location is returned to *`tags`* and the number of tags is returned to *`numTags`*. For details on the error codes, refer to the "Physics Effects Reference" document.

Serialization Processing Procedure

(1) Open a file for output or input and prepare the `FILE*` type file pointer with the C standard library.

(2) Call `Sce::PhysicsEffects::pfxSerializeWrite()` for the output processing, and `Sce::PhysicsEffects::pfxSerializeRead()` for the input processing.

(3) Set the size of each physics instance for serialization input/output to the `PfxSerializeCapacity` type variable, and set the pointer to the buffer to the `PfxSerializeBuffer` type variable. Both the `PfxSerializeCapacity` type variable and the `PfxSerializeBuffer` type variable are passed as an argument in the initialization callback function.

(4) Call the termination callback function to perform the required serialization termination processing at the user application level.

Note
API samples include a sample implementation of serialization processing.

Precautions for Serialization Processing

- In the serialization processing, input and output processing are performed through the pointer set in `PfxSerializeBuffer`. Make sure that all the pointers set in the variable must be valid.
- Memory allocation will be done within the serialization processing if the data input from a file includes a large mesh or a convex mesh. Users can specify the memory allocator called for the memory allocation. Set a function pointer to the memory allocator in advance with the `pfxSetUtilityAllocator()` function. Implement the release processing for these meshes at the application level.
- Close processing for the input and output files is not performed internally. Files must be closed properly by the application that has opened the files.

8 Debug Rendering

Debug Rendering

Debug rendering is an API that provides functions for visualizing the information in the physics engine, such as simulation island, bounding volume, collision coordinates and their restitution direction. Debug rendering provides only functions that support rendering and does not include basic functions to render points , lines, boxes, etc., so as to prevent dependence on specific rendering APIs. At the application level, functions to execute basic rendering processing are provided, and they are called through the user callback functions when debug rendering is executed.

Note
API samples include a sample implementation of debug rendering.

Outline of Usage Procedure of Debug Rendering

The processing flow of debug rendering is as follows.

(1) Create a debug rendering class instance.

(2) Register the user rendering callback function to debug rendering.

(3) Visualize the debug information of the specified rigid body world.

(4) Release the debug rendering.

Creation of Debug Rendering

Create the debug rendering

```
PfxDebugRender *dbg_render = new PfxDebugRender;
```

Registration of Rendering Callback Function

(1) Preparation of rendering callback function

The required information for debug rendering is passed through the callback functions. The actual rendering codes must be implemented at the user level.

Prepare the callback functions to render the basic shapes.

```
Example
void render_debug_line(const PfxVector3 &position1,const PfxVector3
&position2,const PfxVector3 &color)
{
        glColor4f(color[0],color[1],color[2],1.0f);
        …
        glDrawArrays(GL_LINES,0,2);
        …
}
```

The types of rendering callback function are as follows.

Classification	Callback Function Type
Point	`typedef void (*PfxDebugRenderPointFunc)(const PfxVector3 &position,const PfxVector3 &color);`
Line	`typedef void (*PfxDebugRenderLineFunc)(const PfxVector3 &position1,const PfxVector3 &position2,const PfxVector3 &color);`
Arc	`typedef void (*PfxDebugRenderArcFunc)(const PfxVector3 &pos,const PfxVector3 &axis,const PfxVector3 &dir,const float radius,const float startRad,const float endRad,const PfxVector3 &color);`
AABB	`typedef void (*PfxDebugRenderAabbFunc)(const PfxVector3 &center,const PfxVector3 &halfExtent,const PfxVector3 &color);`
Box	`typedef void (*PfxDebugRenderBoxFunc)(const PfxTransform3 &transform,const PfxVector3 &halfExtent,const PfxVector3 &color);`

(2) Registration to debug rendering

Register the rendering callback function prepared through the procedure (1) to the debug rendering.

```
dbg_render->setDebugRenderLineFunc(render_debug_line);
```

Visualization of Debug Information

Reset the visibility flag of debug rendering of rigid bodies.

```
dbg_render->resetVisible(physicsGetMaxNumRigidbodies());
```

Call the actual debug rendering method for each simulation step.

```
dbg_render->renderLocalAxis(states,numRigidbodies);
```

The debug information is rendered after the function internally calls the callback function registered through the procedure (2) of the "Registration of Rendering Callback Function" section.

The types of debug rendering method are as follows.

Method	Description
`renderWorld`	Renders the bounding box in the world area
`renderAabb`	Renders the bounding box of the rigid body
`renderLocalAxis`	Renders the local coordinate axes
`renderIsland`	Renders the simulation island
`renderContact`	Renders the collision information
`renderLargeMesh`	Renders the debug information of large mesh
`renderLargeMeshInFrustum`	Renders the debug information of the large meshes in the view frustum
`renderLargeMeshByFlag`	Renders the debug information of large mesh according to the flag table
`renderJoint`	Renders the debug information of joint

Release of Debug Rendering

Release the debug rendering

```
delete dbg_render;
```

Rendering of Islands and Triangles of Large Mesh

For a large mesh, the debug rendering includes the functions to render the mesh islands and triangles individually.

Set the flag along with the rigid body index, island index and triangle index arguments with `setLargeMeshFlag()` in advance, and then call `renderLargeMeshByFlag()`.

Combination of the following types can be used as a flag.

Rendering Target	Type
Island mesh	`SCE_PFX_DRENDER_MESH_FLG_ISLAND`
Edge of the triangle	`SCE_PFX_DRENDER_MESH_FLG_EDGE`
Bounding box of the triangle (AABB)	`SCE_PFX_DRENDER_MESH_FLG_FACET_AABB`
Normal of the triangle	`SCE_PFX_DRENDER_MESH_FLG_NORMAL`
Thickness of the triangle	`SCE_PFX_DRENDER_MESH_FLG_THICKNESS`
All	`SCE_PFX_DRENDER_MESH_FLG_ALL`

9 FAQ/TIPS

Memory Management

If memory allocation and deallocation are repeatedly performed, memory fragmentation increases, and as a result, the performance may drop. As for the rigid body simulation pipeline, only the collision pair is the element that the numbers change dynamically if the number of objects included in the scene has been determined. Thus, the frequent memory allocation and deallocation can be avoided by preparing heap in advance. Also, the number of the collision pair can be roughly estimated from the number of the rigid bodies included in the scene, so if the adequate number of heap is prepared all memory can be handled statically.

Note
The sample program is written based on this mechanism.

Mass Ratio

If a rigid body interferes with another rigid body whose mass is extremely larger or smaller, the behavior of the rigid bodies will become unstable. Set the mass ratio to 1:10 at the maximum.

How to Specify an Inertia Tensor

Stated simply, it can be said that an inertia tensor is a parameter that indicates the difficulty of rotation of a rigid body. The larger inertia tensor stabilizes a rigid body, and smaller inertia tensor makes the rigid body easy to rotate. When the behavior of a rigid body is unstable (the jittering of the rigid body continues, etc.), it is possible to easily stabilize the rigid body by setting its inertia tensor to a larger value.

Constraint Conditions

When there is a contradiction between constraint conditions, the behavior of a rigid body will become unstable. The example situation is when a rigid body fixed with a joint is pushed and crashes into the wall. (The constraint force of the joint tries to get back the rigid body to the correct position; on the other hand, the restitution force tries to resolve the crash. The two forces are affected each other, so the both conditions cannot be satisfied at the same time.)

Improve the Precision Comprehensively

In order to improve the precision of a rigid body simulation comprehensively, try to shorten the time step and increase the number of the constraint solver iterations. In general, 60 frames per second and 5 to 10 iterations are considered as a sufficient precision. When reproducing a more complex physics mechanism, try to increase the numbers.

Unit System

Simulation is designed to run stably with a standard MKS unit system (length: m, weight: kg, time: sec). Note that handling a length of 1 as 1 cm or 1 km - in other words, handling very different unit systems - or using an extremely heavy (or light) mass in comparison to the size of a shape, will lead to instability of the simulation.

10 Reference Information

A List of the Behavior Change by Type

Collision Detection

	Fixed	Active	Key Frame	One Way	Trigger
Fixed	No	Yes	No	Yes	Yes
Active		Yes	Yes	Yes	Yes
Key Frame			No	Yes	Yes
One Way				No	Yes
Trigger					No

Constraint Solver

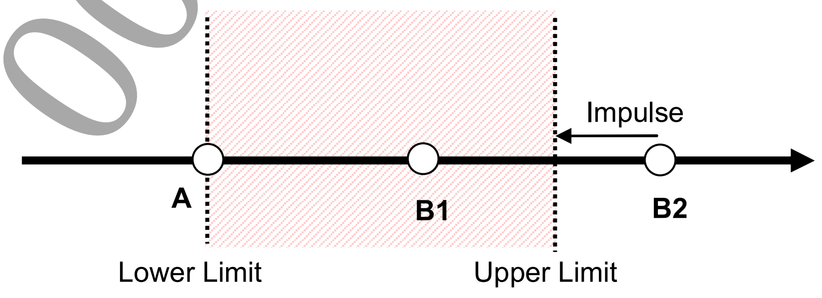
	Fixed	Active	Key Frame	One Way	Trigger
Fixed	No	Yes	No	Yes	No
Active		Yes	Yes	Yes	No
Key Frame			No	Yes	No
One Way				No	No
Trigger					No

Compose Contact Points

`PfxContactManifold`, which contains collision information, can store up to four contact points. If five or more contact points are detected, four contact points including the contact point having the deepest penetration among those contact points are selected so that the area generated by the four contact points is maximized.

Mechanism of a Joint Constraint

A joint is represented by the combinations of `PfxJointConstraint`, which represent constraints on the specified axis. The combination (or which constraint will be used) depends on the type of the joint. The constraint axis will be the direction and rotation axis for translation constraints and rotation constraints respectively. A constraint force is calculated based on the relativity between the velocity and position of two rigid bodies on the constraint axis. In the Physics Effects library, among six `PfxJointConstraint` of `PfxJoint`, the first three are considered as translation constraint, and the remaining three are considered as rotation constraints.



Suppose one axis of the joint has the following constraints. The parent's connection point on this axis is fixed at A, while the child's connection point B is allowed a range of motion between the Lower Limit and the Upper Limit. When the child's position is at B1, which is within the range of motion, there is no constraint (if the damping value is 0 or more, a dumping force will occur in a direction opposite to the velocity). On the other hand, when the position is at B2, which is outside the range of motion, there will be a constraint force pulling back the position (B2) to the Upper Limit.

The constraint force is calculated as an impulse which pulls back the position B to the Upper Limit (positional error correction) and makes the velocity of B to 0 or less (velocity error correction) after time step. This operation is repeated for the number of iterations, and the shift of the joint will eventually be corrected.

The impulse can be adjusted by using bias and weight values in the joint parameters. The bias value is a parameter that affects the positional error correction, and the weight value is a parameter that affects the whole calculated impulse. For stable behavior, specify the range of values between 0.0 and 1.0. A simplified formula for calculating the impulse is as follows.

Impulse = Weight value x Func(Bias value x Positional error correction value, Velocity error correction value)

Note
To fix the constraint position, specify the same value to the Lower Limit and the Upper Limit.