Physics Effects High Level API Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

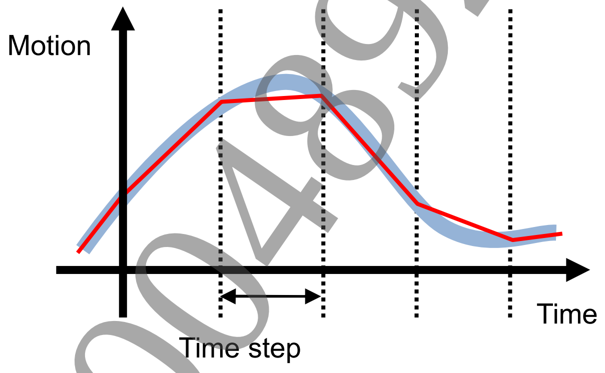
1 Physics Engine

Physics Engine Overview

A physics engine can be used as a tool to create in real time animations of various objects that appear mainly in games. By appropriately setting beforehand physical attributes including mass and velocity for individual objects, a physics engine automatically generates behavior that follows physical laws through computations. The objects input to the physics engine are treated as rigid bodies, collisions between rigid bodies are detected, and once the interaction between rigid bodies has been solved, the computation results are output after the specified time. By repeating these computations for each frame, a system for reproducing behavior according to physical laws is achieved.

Physics Engine System

Physics engines treat time not as a continuous flow but as a series of discrete steps. In other words, as time flows, physics operations are executed at certain time step intervals, and smooth motion changes between frames are output as linear changes, as shown in the figure below. As can be seen from the nature of this system, the accuracy of the physics operations is higher for shorter time steps, and vice versa, lower for larger time steps. Degradation of operation accuracy manifests itself in the form of object interpenetration, joint elongations, vibrations, and so on. In order to obtain the desired effect without loss of stability, it is important to appropriately set the parameters of the physics engine.



Simulation Pipeline

The physics operations executed in the physics engine consist of multiple successive processing stages, and this series of stages is called the simulation pipeline. The general flow is as follows.

First, rigid bodies that are highly likely to collide are searched for (broad phase). Pairs of detected rigid bodies highly likely to collide are checked for collision, and if a collision has actually occurred, the collision coordinates are calculated next (collision detection). Next, in order to solve the detected collision, the restitution force of the rigid bodies is calculated (by a solver). At this stage, the constraint force from the joint(s) is simultaneously calculated. The solver outputs the results as velocity changes, and lastly, the positions of the rigid bodies after the time steps are calculated using these velocity changes (integration).

2 Physics Effects High Level API Overview

Purpose and Features

The Physics Effects library provided as part of the SDK is a group of parts that brings together functions related to physics simulation and is designed so that functions equivalent to physics engines are built at an application level. For that reason, on its own, it cannot be used as a physics engine. Therefore, a class library that incorporates the required minimum of functions to facilitate the use of the Physics Effects library as a physics engine is provided as high level APIs.

The Physics Effects high level APIs conceal the lower level APIs of the Physics Effects library and provide higher level functions. Further, as it comes with the extended functions required for general game development, it can be used as a base when embedding it in a game engine.

Main Functions

The Physics Effects High Level APIs provide the following extended functions in addition to the functions provided by the Physics Effects library.

- Buffer management
- Fixed pipeline construction
- Sleep mechanism
- Area raycast
- AABB overlap
- Registration of non-colliding pairs
- Multithread
- Debug support
- Sub-step simulation
- Collision query

Resource Consumption

The Physics Effects High Level APIs are designed so that the required buffers are provided externally. Therefore, dynamic memory allocation is not done internally.

Name Space

Like the Physics Effects library, the Physics Effects High Level APIs are defined by the `sce::PhysicsEffects` namespace.

Embedding in Program

(1) Embedding the Physics Effects Library

Link the libScePhysicsEffects.a static link library to the project. The Physics Effects library uses the libult and Vector Math libraries. For details, refer to the documents of these respective libraries.

(2) Embedding the Physics Effects High Level APIs

The source code of the Physics Effects High Level APIs is located in the following folder.

- samples/sample_code/engines/tutorial_physics_effects_high_level/high_level/

Add the source code in the folder to all the project, and include pfx_high_level_include.h.

Tutorial

Tutorial build

Load samples/sample_code/engines/tutorial_physics_effects_high_level/tutorial_physics_effects_high_level.psp2.sln to Visual Studio and execute build.

Tutorial structure

The physics simulation execution code using the Physics Effects High Level APIs is described in the following source.

- samples/sample_code/engines/tutorial_physics_effects_high_level/physics_func.cpp

The debug support execution code is described in the following source.

- samples/sample_code/engines/tutorial_physics_effects_high_level/main.cpp

Physics-related functions of physics_func.cpp

Function	Contents	Description
`physicsInit()`	Initialization processing	Allocates the required buffers and creates the rigid body world.
`physicsRelease()`	End processing	Executes the rigid body world end processing and releases the buffers.
`physicsCreateScene()`	Scene creation	Creates rigid bodies and joints, and registers them to the rigid body world.
`physicsSimulate()`	Simulation execution	Executes physics simulation.
`physicsPickStart()`	Picking start	Executes raycasting from the coordinates of the touched screen to a deeper section of the scene and connects hit coordinates and the rigid body with joints.
`physicsPickUpdate()`	Picking update	Updates the connection coordinates while the screen is touched.
`physicsPickEnd()`	Picking end	The touch ends and the joint becomes inactive.
`updateRayCastObject()`	Ray update	Assigns coordinates to the rays used in a raycast scene.
`castRayCastObject()`	Raycast execution	Executes raycasting and detects the intersections of rays and rigid bodies.

Scenes included in tutorial

The tutorial includes five scenes that represent the features of the physics engine.

- Scene 1: Basic rigid body
- Scene 2: Stacking
- Scene 3: Ragdoll
- Scene 4: Large mesh
- Scene 5: Raycast

References

For details on the Physics Effects library, refer to the following document:

- Physics Effects Overview, Physics Effects Reference

3 Usage Procedure

Outline of Usage Procedure of Physics Effects High Level APIs

The processing flow for using the Physics Effects High Level APIs is as follows.

(1) Create a `PfxRigidBodyWorld` instance and initialize the rigid body world.

(2) Create physics objects such as rigid bodies and joints, and register them to the rigid body world.

(3) Execute simulation steps at the specified time.

(4) Release the rigid body world.

Rigid body world creation procedure

(1) Setting of rigid body world parameters

Specify the appropriate parameters to configure the rigid body world to `PfxRigidBodyWorldParam`.

Rigid body world parameters

Parameter	Default Value	Description
`maxRigidBodies`	500	Maximum number of rigid bodies
`maxJoints`	500	Maximum number of joints
`maxShapes`	100	Maximum number of shapes
`maxContacts`	4000	Maximum number of contacts
`maxNonContactPairs`	100	Maximum number of non-contact pairs
`worldCenter`	0.0,0.0,0.0	Center of world
`worldExtent`	500.0,500.0,500.0	Half of world size
`gravity`	0.0,-9.8,0.0	Gravity
`timeStep`	0.016	Time step
`separateBias`	0.1	Bias value for adjusting restitution force during collision
`iteration`	5	Number of solver iterations. The greater the value, the higher the accuracy of the physics operations, but at the cost of reduced performance.
`sleepCount`	180	Number of steps to enter sleep
`sleepVelocity`	0.1	Threshold velocity to enter sleep
`multiThreadFlag`	0	Parallel processing flag
`numTasks`	1	Number of tasks processed in parallel
`simulationFlag`	0	Active function flag for simulation

Values specified for multiThreadFlag (can be combined)

Value	Description
`SCE_PFX_ENABLE_MULTITHREAD_BROADPHASE`	Executes broad phase in parallel
`SCE_PFX_ENABLE_MULTITHREAD_COLLISION`	Executes collision judgment in parallel.
`SCE_PFX_ENABLE_MULTITHREAD_SOLVER`	Executes solver in parallel
`SCE_PFX_ENABLE_MULTITHREAD_INTEGRATE`	Executes integration in parallel
`SCE_PFX_ENABLE_MULTITHREAD_RAYCAST`	Executes raycast in parallel
`SCE_PFX_ENABLE_MULTITHREAD_ALL`	Executes all stages in parallel

Value specified for simulationFlag

Value	Description
`SCE_PFX_ENABLE_SLEEP`	Uses sleep function
`SCE_PFX_ENABLE_CONTACT_CACHE`	Caches collision data for collision query

(2) Allocation of buffer for rigid body world parameters

Call `PfxRigidBodyWorld::getRigidBodyWorldBytes()` using `PfxRigidBodyWorldParam` as the argument and get the buffer size according to the size of the scene. Allocate the memory according to this size and specify the size to *`poolBytes`* and the buffer to *`poolBuff`* of `PfxRigidBodyWorldParam`.

```
PfxRigidBodyWorldParam worldParam;
// ...
// set values to world parameters
// ...
PfxUInt32 poolBytes = PfxRigidBodyWorld::getRigidBodyWorldBytes(worldParam);
void *poolBuff = new unsigned char [poolBytes];
worldParam.poolBytes = poolBytes;
worldParam.poolBuff = poolBuff;
```

(3) Creation of rigid body world

Specify the rigid body world parameters created in step (1) to the constructor argument and create an instance of the `PfxRigidBodyWorld` class.

```
PfxRigidBodyWorld *world = new PfxRigidBodyWorld(worldParam);
```

(4) Initialization of rigid body world

Call the `initialize()` method of the created rigid body world and complete initialization.

```
world->initialize();
```

Note
Create the scenes so as not to exceed the maximum scene size set to the rigid body world parameters. Insufficient buffer size will cause an assertion or an error code to be returned.

Rigid Body Creation Procedure

(1) Creation of shape

First, create a single `PfxShape` shape.

```
PfxBox box(1.0f,1.0f,1.0f);
PfxShape shape;
shape.reset();
shape.setBox(box);
```

Set this shape to `PfxCollidable`.

```
PfxCollidable collidable;
collidable.reset();
collidable.addShape(shape);
collidable.finish();
```

`PfxCollidable` can hold one shape internally, but when creating a rigid body consisting of two or more shapes, a shape buffer allocated in the world is used. Therefore, when creating `PfxCollidable` holding multiple `PfxShape`, perform registration after allocating the required number of shapes beforehand from the rigid body world. Registration exceeding the set number of shapes is not possible.

```
world->setupCollidable(collidable,3); // holds 3 shapes
collidable.addShape(shape1);
collidable.addShape(shape2);
collidable.addShape(shape3);
collidable.finish();
```

(2) Creation of rigid body

Once the shapes have been created, set the values expressing the physical attributes of the rigid body to `PfxRigidState` and `PfxRigidBody`. For details on the physical attributes, refer to the "Physics Effects Overview" and "Physics Effects Reference" documents.

```
PfxRigidState state;
PfxRigidBody body;
body.reset();
body.setMass(1.0f);
body.setInertia(pfxCalcInertiaBox(PfxVector3(1.0f),1.0f));
state.reset();
state.setPosition(PfxVector3(0.0f,5.0f,5.0f));
state.setMotionType(kPfxMotionTypeActive);
state.setUseSleep(1);
```

(3) Registration of rigid body to world

Lastly, specify the three created data for the argument and call `PfxRigidBodyWorld::addRigidBody()` to register them to the world. Since the data are immediately copied to the internal buffer in the world, they can be discarded following registration. `PfxRigidBodyWorld::addRigidBody()` returns a unique ID that expresses the rigid body as an integer. Thereafter, access the rigid body information using this ID.

```
PfxUInt32 rigidbodyId = world->addRigidBody(state,body,collidable);
```

Note
By calling `PfxRigidBodyWorld::removeRigidBody()` using the unique ID as an argument, the rigid body can be deleted from the scene. `PfxRigidState`, `PfxRigidBody`, and `PfxCollidable` (including `PfxShape`) that configure the rigid body are stored in an internal pool buffer and are reused when creating the next rigid body. To check whether a rigid body with a unique ID is valid, use `PfxRigidBodyWorld::isRemovedRigidBody()`. This operation also applies to joints.

Joint Creation Procedure

(1) Creation of joint

Create and initialize the joint using the APIs of the Physics Effects library.

```
PfxRigidState &stateA = world->getRigidState(rigidbodyIdA);
PfxRigidState &stateB = world->getRigidState(rigidbodyIdB);

PfxHingeJointInitParam jparam;
jparam.anchorPoint = 0.5f * (stateA.getPosition() + stateB.getPosition());
jparam.axis = PfxVector3(1.0f,0.0f,0.0f);
jparam.lowerAngle = 0.0f;
jparam.upperAngle = 1.57f;

PfxJoint joint;
pfxInitializeHingeJoint(joint,stateA,stateB,jparam);
```

(2) Registration of joint to world

Specify `PfxJoint` for the argument and call `PfxRigidBodyWorld::addJoint()` to register the joint to the rigid body world. Since the data are immediately copied to the internal buffer in the rigid body world, they can be discarded following registration. `PfxRigidBodyWorld::addJoint()` returns the unique ID expressing the joint as an integer. Thereafter, access the joint information using this ID.

```
PfxUInt32 jointId = world->addJoint(joint);
```

(3) Registration of non-contact pair

When connecting two rigid bodies with a joint, one often may want to disable the detection of collisions between these two rigid bodies. In such a case, call `PfxRigidBodyWorld::appendNonContactPair()` with the unique IDs of the two rigid bodies in question as the argument, and register them as a non-contact pair to the rigid body world.

```
world->appendNonContactPair(rigidbodyIdA,rigidbodyIdB);
```

To release a non-contact pair, call `PfxRigidBodyWorld::removeNonContactPair()` for that pair.

```
world->removeNonContactPair(rigidbodyIdA,rigidbodyIdB);
```

Simulation Procedure

(1) Execution of simulation

Call `PfxRigidBodyWorld::simulate()` to execute simulation.

```
world->simulate();
```

(2) Sub-step of a simulation

If you want to execute simulation multiple times within 1 frame (sub-step execution), call `PfxRigidBodyWorld::simulateSubStep()` instead of `PfxRigidBodyWorld::simulate()` and specify the number of sub-steps as an argument. Because the world time step will be directly used multiple times in a simulation; when simulation is carried out twice with 30 fps (for example), specify 1/60 second ≒ 0.016 for time step and 2 for the number of sub-steps.

```
world->simulateSubStep(2);
```

Rigid Body World Release Procedure

(1) Notification of end to rigid body world

Call `PfxRigidBodyWorld::finalize()` to execute the end processing.

```
world->finalize();
```

(2) Release of buffer

Lastly, release the rigid world and buffer.

```
delete world;
delete [] poolBuff;
```

Raycast Procedure

(1) Specification of area

Before executing raycasting, call `PfxRigidBodyWorld::setCastArea()` to set the area of influence. The raycast performance can be increased by limiting the area of influence. `setCastArea()` creates the proxy array of the rigid bodies included in the area and saves it to the internal buffer in the rigid body world. If the area of influence is not specified, the current world size is used instead, but in order to create the proxy array, be sure to execute simulation once.

```
// limit area to within 5x5x5 from the origin
PfxVector3 areaCenter(0.0f,0.0f,0.0f);
PfxVector3 areaExtent(5.0f);
world->setCastArea(areaCenter,areaExtent);
```

(2) Execution of raycasting

Prepare respective arrays for ray input and output, and set the ray information to the ray input buffer.

Execute raycasting by specifying the input and output arrays to `PfxRigidBodyWorld::castRays()`. When ray intersection is detected, the intersection coordinates closest to the start point are saved to the ray output array.

```
PfxRayInput rayInputs[100];
PfxRayOutput rayOutputs[100];
for(int i=0;i<100;i++) {
        rayInputs[i].reset();
        // ...
        // Set start point of ray and direction vector
        // ...
}
world->castRays(rayInputs,rayOutputs,100);
```

AABB Overlap Detection Procedure

AABB overlap provides a function for detecting rigid bodies that intersect the specified area in the world. It is used, for example, to collect the rigid bodies around a particular object.

(1) Specification of area

Similarly to raycasting, call `PfxRigidBodyWorld::setCastArea()` and specify the area of influence.

(2) Execution of AABB overlap detection

First, set a bounding box for detecting intersections to `PfxAabbInput`.

```
PfxAabbInput aabb;
aabb.reset();
aabb.center = PfxVector3(0.0f);
aabb.extent = PfxVector3(3.0f);
```

Call `PfxRigidBodyWorld::findAabbOverlap()` by specifying the bounding box, the buffer for storing the result, and the maximum number of items that can be received. Following execution of this function, the intersections with the bounding box of the rigid body are detected, the index of the intersected rigid bodies is stored to *`intersectBuff`*, and their number is returned to *`numIntesection`*.

```
PfxUInt16 intersectBuff[10]
PfxUInt32 numIntesection;
findAabbOverlap(aabb,intersectBuff,numIntesection,10);
```

Collision Query Procedure

An API is provided to search for the collision buffer from the rigid body index and obtain the related `PfxQueryContactInfo` collision data. Even when sub-step executing a simulation, accumulate the collisions detected per sub-step and avoid missing any undetected collisions.

```
struct PfxQueryContactPoint {
        PfxVector3 pointA; // Collision point of rigid body A in a world
coordinate system
        PfxVector3 pointB; // Collision point of rigid body B in a world
coordinate system
        PfxVector3 normal; // Normal line of the collision point in a world
coordinate system
        PfxFloat distance; // Penetration depth of the collision
};

struct PfxQueryContactInfo {
        PfxQueryContactPoint contactPoints[4];
        PfxUInt32 numContactPoints;
        PfxUInt16 rigidbodyIdA;
        PfxUInt16 rigidbodyIdB;
        PfxUInt32 pairId;
};
```

(1) Allocate a collision buffer for query

To cache collision data used for a query, specify `SCE_PFX_ENABLE_CONTACT_CACHE` to the `PfxRigidBodyWorldParam::simulationFlag` rigid body world creation parameter.

```
PfxRigidBodyWorldParam worldParam;
…
worldParam.simulationFlag |= SCE_PFX_ENABLE_CONTACT_CACHE;
```

(2) Get collision data from the two rigid body indices

Specify the two rigid body indices as arguments and call `PfxRigidBodyWorld::queryContact()` to obtain collision data. If applicable collision data does not exist, the function will return NULL.

```
// Get collision data of rigid body index 10 and index 30
const PfxQueryContactInfo *contactInfo = world->queryContact(10,30);
if(contactInfo) {
        // Processing regarding this collision
}
```

(3) Repeatedly obtain collision data relating to one rigid body index

Specify the rigid body index as an argument and call `PfxRigidBodyWorld::queryFirstContact()` to obtain collision data. Because collision data is stored in a list structure, the above function can be called repeatedly until it returns NULL to obtain all related collision data of the specific rigid body.

```
const PfxQueryContactInfo *contactInfo = world->queryFirstContact(10);
while(contactInfo) {
        // Processing regarding this collision
        contactInfo = world->queryNextContact(contactInfo,10);
}
```