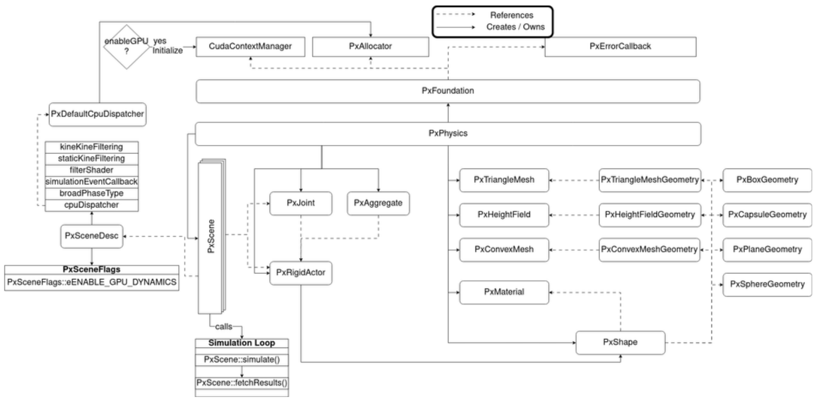
Collision Checking with PhysX

Epic Link: RBP-1838: HACC: MVP using PhysX DONE

PhysX Concepts

Rigid Body Overview

The following diagram provides an overview of the different components that participate in running a PhysX simulation



Memory Management and Error Reporting

> PhysX performs all heap allocations via the PxAllocatorCallback interface. This is required to instantiate all the other components in PhysX. A simple implementation of the allocator callback class is implemented in the PxDefaultAllocator.

Throughout this PoC, the default allocator has been used.

> PhysX logs all error messages through the PxErrorCallback interface. This must also be implemented in order to initialize PhysX. A simple implementation of the allocator callback class is implemented in the PxDefaultErrorCallback.

Code Snippet:

```
PxDefaultAllocator mAllocator;
PxDefaultErrorCallback mErrorCallback;
```

`PxFoundation` and `PxPhysics`

> Every PhysX module requires a PxFoundation instance to be available. The required parameters are a version ID, an allocator callback and an error callback. `PX_PHYSICS_VERSION`, is a macro predefined in our headers to enable PhysX to check for a version mismatch between the headers and the corresponding SDK DLLs.

> PxPhysics is the module that is responsible for instantiating PxScene and creating and maintaining the Actors within the scene.

Code Snippet:

```
PxFoundation* mFoundation = PxCreateFoundation(PX_PHYSICS_VERSION, mAllocator, mErrorCallback);
PxPhysics* mPhysics = PxCreatePhysics(PX_PHYSICS_VERSION, *mFoundation, PxTolerancesScale(), true, mPvd);
```

`PxScene`

The PxScene module is primarily used to run simulation loops (through `PxScene::simulate()`) or for scene queries (such as Sweeps and Overlaps). This module is the main interface for running collision checks in PhysX.

It needs a reference to a physx::PxSceneDesc object during instantiation. The `PxSceneDesc` requires two mandatory variables to be set:

- `PxSceneDesc::cpuDispatcher` -- for spawning worker threads
- `PxSceneDesc::fitlerShader` -- for collision filtering

A simple implementation of the cpu dispatcher is provided in PxDefaultCpuDispatcher. This is used as the cpu dispatcher for all the scenes in the PoC. A default filter shader PxDefaultSimulationFilterShader is also available if we don't wish to implement our own collision filtering shader.

Code Snippet:

```
PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
PxDefaultCpuDispatcher* mDispatcher = PxDefaultCpuDispatcherCreate(mNumThreads);  // exports the scene on the CPU

sceneDesc.cpuDispatcher = mDispatcher;
sceneDesc.filterShader = PxDefaultSimulationFilterShader;

```

```
// the physics creates the scene from the scene description
PxScene* mScene = mPhysics->createScene(sceneDesc);
```

Creating Rigid Actors

> Rigid actors along with their shapes can be created through the *PxPhysics* instance
>
> PxRigidActor represents a base class shared between dynamic and static rigid bodies in the physics SDK.
>
> PxRigidActor objects specify the geometry of the object by defining a set of attached shapes (see PxShape).

The base class is inherited by the PxRigidBody (used for creating PxArticulationLinks and PxRigidDynamic actors) and PxRigidStatic (for creating static actors) subclasses

Two important methods provided by the base class are:

- attachShape - Attach a shape to an actor
- setGlobalPose - Method for setting an actor’s pose in the world. *Instantaneously* changes the actor space to world space transformation

Code Snippet:

```
//plane rigid static
PxRigidStatic* rigidStatic = mPhysics->createRigidStatic(PxTransformFromPlaneEquation(PxPlane(PxVec3(0.f, 1.f, 0.f), 0.f)));
PxRigidActorExt::createExclusiveShape(*rigidStatic, PxPlaneGeometry(), material);

//single shape rigid dynamic
PxRigidDynamic* rigidDynamic = mPhysics->createRigidDynamic(PxTransform(PxVec3(0.f, 2.5f, 0.f)));
PxRigidActorExt::createExclusiveShape(*rigidDynamic, PxBoxGeometry(2.f, 2.f, 2.f), material);

mScene->addActor(*rigidStatic);
mScene->addActor(*rigidDynamic);
```

Simulation

The PxScene::simulate() method is used to advance the world forward in time. In our case, for trajectory planning and inverse kinematics, we only need a single step simulation where we just teleport the object to the target pose and make a single call to `simulate()` to perform a collision check

Code Snippet:

```
PxF32 mStepSize = 1.0f / 60.0f;
PxTransform targetPose; // some target pose
PxRigidDynamic* queryObject; // some RigidDynamic actor

queryObject->setGlobalPose(targetPose); // teleport to the target pose
queryObject->setKinematicTarget(targetPose); // wakes up the actor so that it participates in collision checking

mScene->simulate(mTimeStep);
mScene->fetchResults(true);
```

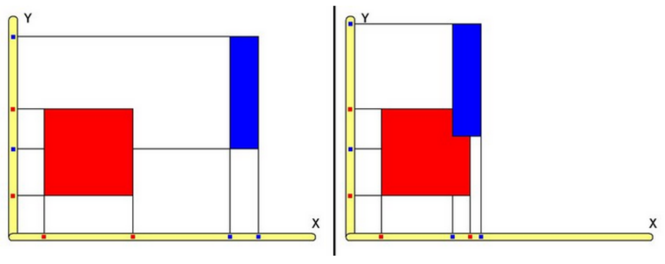
Rigid Body Collision

Broad-phase

The broad phase is the first part of the collision pipeline. It is called broad phase because it detects overlaps between axis-aligned bounding boxes, i.e it only reports potential collisions rather than actual collisions. The actual collisions are detected by the next phase in the physics pipeline, named narrow phase.

Algorithms implemented in PhysX

Sweep-and-Prune (SAP)



Sweep and prune for non-intersection(left) and intersecting(right) objects in 2D

Sweep-and-Prune is a method often used in collision detection to efficiently find overlapping bounding boxes in a simulation.

- *Sweeping Phase:* All the bounding boxes are sorted along an arbitrary number of axes. This operation is done by "sweeping" a line along chosen axes while keeping track of the beginning and end of each box.
- *Pruning Phase*: As the line sweeps across, it is checked whether the bounding boxes are overlapping on the chosen axis. If so, further checks are performed on other axes to verify if a collision is occurring. This "pruning" step discards non-colliding pairs quickly.

This algorithm is widely appreciated for its efficiency in scenarios where objects are moving slowly or are mostly static.

Multibox Pruning (MBP)

Multibox Pruning is an extension of the SAP algorithm. The key difference is that it uses several bounding volumes instead of one, creating a hierarchy of bounding boxes.

- *Box Hierarchy:* Objects are enclosed in several bounding boxes, forming a hierarchy.
- *Pruning Phase:* Similar to SAP, a sweeping process is used, but it is applied to each level of the hierarchy. This approach can reduce the number of pairs to be checked, making the process more efficient.

Automatic box Pruning (ABP)

> Revisited implementation of *PxBroadPhaseType::eMBP* introduced in PhysX 4. **It automatically manages world bounds and broad-phase regions**, thus offering the convenience of *PxBroadPhaseType::eSAP* coupled to the performance of *PxBroadPhaseType::eMBP*.

Parallel Automatic box Pruning (PABP)

PABP builds upon ABP by parallelizing the computation. This can significantly speed up the collision detection process, especially in environments with many objects.

- *Parallel Sorting:* The sorting phase is parallelized, dividing the task among multiple processors or threads. (Parallelized Radix Sort [1])
- *Parallel Pruning:* The pruning phase is also parallelized, allowing for simultaneous processing of multiple object pairs.

By exploiting parallel computing resources, PABP can handle more complex scenarios with numerous objects or very detailed geometries.

GPU Broadphase

> It is a GPU implementation of the incremental sweep and prune [2] algorithm. It not only has the advantage of traditional SAP approach which is good for when many objects are sleeping, but due to being fully parallel, it also is great when lots of shapes are moving or for runtime pair insertion and removal.

Collision Filtering

> Collision filtering is the mechanism used to discard some overlapping pairs returned by the broadphase.

PhysX implements collision filtering in several stages of the physics pipeline. From cheapest & least flexible to most expensive & most flexible, filtering happens:

- during the broadphase, using `PxPairFilteringMode`.
  - Enable kinematic-vs-kinematic interactions
    - `PxSceneDesc::kineKineFilteringMode = PxPairFilteringMode::eKEEP`
  - Enable kinematic-vs-static interactions
    - `PxSceneDesc::staticKineFilteringMode = PxPairFilteringMode::eKEEP`
- during or after the broadphase, using `PxSimulationFilterShader`.

> A *filter shader* is a standalone user-defined C function called for all pairs of shapes whose axis aligned bounding boxes in world space are found to intersect for the first time. All behavior beyond that is determined by what the shader returns.
>
> In theory such a shader can run on the GPU and thus it should not reference any memory other than arguments of the function and its own local stack variables

A custom filter shader was implemented and tested in the PoC to report contacts only between specific FilterGroups

The following PairFlags are necessary to

- after the broadphase, using `PxSimulationFilterCallback`.
  - It is the most flexible because contrary to the filter shader, the callback can access any memory and do anything it needs to.
  - It is the slowest because it happens after the broadphase, at the last stage of the filtering pipeline, and it is implemented with regular virtual calls.
  - This is not used in the PoC

Narrow-phase

The narrow phase involves processing a set of pairs produced by the broad phase to determine if the geometries are actually interacting and generate contacts for the pairs that do intersect / overlap.

Persistent Contact Manifold (PCM)

> PCM is a fully distance-based collision detection system

When two objects collide, they generally do so across multiple points. The set of these collision points can form a manifold, and the Persistent Contact Manifold algorithm aims to manage this manifold over time.

Rather than recalculating the entire manifold from scratch every frame, the algorithm updates the existing manifold, removing irrelevant points and adding new ones. This approach saves computational resources and provides smoother collision responses.

Under the hood, it makes use of the Expanded Polytope Algorithm (EPA) [3] for generating contact information

Contact reporting

Once the relevant flags for the colliding pairs are raised through the filter shader, the corresponding contact information is captured in the *ContactReportCallback*. This callback is a subclass of PxSimulationEventCallback. Whenever pair triggers a contact event (such as `PxPairFlag::eNOTIFY_TOUCH_FOUND` or `PxPairFlag::eNOTIFY_TOUCH_PERSISTS`), the `PxSimulationEventCallback::onContact()` method is invoked. This is where we can obtain information about the contacts generated for the triggering pairs.

Penetration depth

Penetration depth can be computed using PxGeometryQuery::computePenetration. This however is a separate query and not part of the rigid body collision pipeline. ***This is also not GPU accelerated.***

Another way would be to write our own function to compute penetration depth from the contact information received in the *ContactReportCallback.* The naive implementation of computing distances between every pair of contact points and then choosing the maximum value could be parallelized on the CPU.

GPU Acceleration

The GPU rigid body feature provides GPU-accelerated implementations of:

- Broad Phase
- Narrow Phase (contact generation)
- Shape and body management
- Constraint solver

In our case of motion planning we are mostly interested in the first three.

In order to enable the GPU accelerated features, we need to do the following:

- Create a CudaContextManager

```
PxCudaContextManagerDesc cudaContextManagerDesc;
PxCudaContextManager* mCudaContextManager = PxCreateCudaContextManager(*gFoundation,
                                                                        cudaContextManagerDesc,
                                                                        PxGetProfilerCallback());
```

- Pass the context manager to `physx::PxSceneDesc`

```
sceneDesc.cudaContextManager = mCudaContextManager;
```

- Enable GPU narrowphase

```
sceneDesc.flags |= PxSceneFlag::eENABLE_GPU_DYNAMICS;
```

- Enable the GPU broadphase

```
sceneDesc.broadPhaseType = PxBroadPhaseType::eGPU;
```

The GPU narrow phase supports boxes, convex hulls, meshes and heightfields. Meshes and convex hulls require PxCookingParams::buildGPUData to be set during cooking. Contacts generated on GPU are automatically transferred back to the CPU as needed

Modifying the state of actors forces data to be re-synced to the GPU, e.g. transforms for actors must be updated if the application adjusts global pose, velocities must be updated if the application modifies the bodies' velocities etc. This is managed by the GPU Simulation Controller (github)

Pros and Cons

Pros:

1. Parallelized CPU collision pipeline (optimized over several versions)
2. Optimized implementation for GPU collision pipeline. Specifically the broadphase and narrowphase stages which is what we care about during trajectory planning and inverse kinematics
3. API provides several easy-to-use ways to execute the rigid body pipeline for specific scenarios: simulate(), collide() + advance() and scene queries

Cons:

1. GPU acceleration provides performance speedups for large scenes with multiple moving objects ideal for video games. Therefore, we might not get the most optimal performance benefits from the GPU acceleration

Proof-of-Concept

Repository Details

Link: https://gitlab.ocado.tech/advanced-technology/pre-production/hacc_physx_mvp

Aliases

There are two alias files in order to ease building and running different components of the project

`ws-aliases.sh` is used to setup workspace environment for O3DE and for building images and running containers. For more details refer: https://gitlab.ocado.tech/advanced-technology/pre-production/hacc_physx_mvp/-/blob/main/workspace_config/ws-aliases.sh

`container-aliases.sh` is used for running collision_tests or opening the O3DE editor, etc inside their respective containers. For more details refer: https://gitlab.ocado.tech/advanced-technology/pre-production/hacc_physx_mvp/-/blob/main/workspace_config/container-aliases.sh

STL files

The UR10e link STLs as well as the gripper STL are take from the `robot_control_platform` repo along with the STL files for `simple_ur_test_scene`. The STL files part of this project are listed below:

```
kindred_arms
└── ur10e
    ├── meshes
    │   ├── collision
    │   │   ├── 550-00260REV02.STL
    │   │   ├── base.stl
    │   │   ├── forearm.stl
    │   │   ├── shoulder.stl
    │   │   ├── upperarm.stl
    │   │   ├── wrist1.stl
    │   │   ├── wrist2.stl
    │   │   └── wrist3.stl
    │   └── visual
    │       ├── 550-00260REV02.STL
    │       ├── base.dae
    │       ├── forearm.dae
    │       ├── shoulder.dae
    │       ├── upperarm.dae
    │       ├── wrist1.dae
    │       ├── wrist2.dae
    │       └── wrist3.dae
    └── mp2_ur10e.urdf
kindred_scenes
└── simple_ur_test_scene
    └── meshes
        ├── 550-00214REV05.STL
        ├── 550-00215REV04.STL
        ├── 550-00218REV03.STL
        ├── 550-00219REV02.STL
        ├── 550-00220REV03.STL
        ├── 550-00221REV03.STL
        └── 550-00227REV05.STL

```

UR10ePoC - O3DE stuff

TODO: Gather relevant bits from the early work with O3DE

`collision_tests`

This directory contains the following:

1. Tests for FCL
2. Tests for PhysX
3. Processed STL files that are required by FCL and PhysX in a particular format
4. Python scripts to process the STL files and generate frame data from the tests in csv format
5. `CMakeLists.txt` to build the `collision_tests` project

PoC Source: `collision_tests`

Building and running the docker container

Follow these steps to build and run the `collision_tests` docker container

- Ensure that you have the nvidia-container-runtime installed

```
sudo apt-get install nvidia-container-runtime
```

- Add the runtime to the docker daemon configuration file

```
sudo tee /etc/docker/daemon.json <<EOF
{
    "runtimes": {
        "nvidia": {
            "path": "/usr/bin/nvidia-container-runtime",
            "runtimeArgs": []
        }
    }
}
EOF
sudo pkill -SIGHUP dockerd
```

- Clone the repo and run `source workspace_config/ws-aliases.sh`
- Build the `collision_tests` image by running `ws-build collision_tests`
- Once build is complete, run `ws-up collision_tests` to start the container
- Exec into the container shell by running `ws-exec collision_tests`

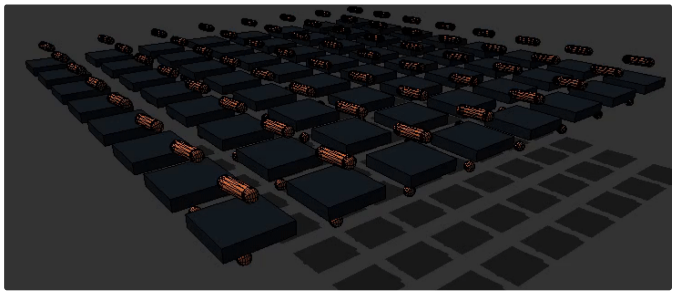
Existing tests:

1. FCL
   a. `fcl_primitive_collider` - Performs collision checking by randomly spawning box, spheres and cylinder primitives and kinematically moving a subset of the objects in each step
   b. `fcl_mesh_collider` - Collision test that loads the gripper and tote mesh by reading from their corresponding OBJ files and performs a iterative collision check by kinematically moving the gripper towards the tote
   c. `fcl_mesh_geom_compare` - Loads the original gripper mesh and the convex hull approximation to compare collision checking performance
2. PhysX
   a. snippets: (*dir: collision_tests/physx/snippets*)

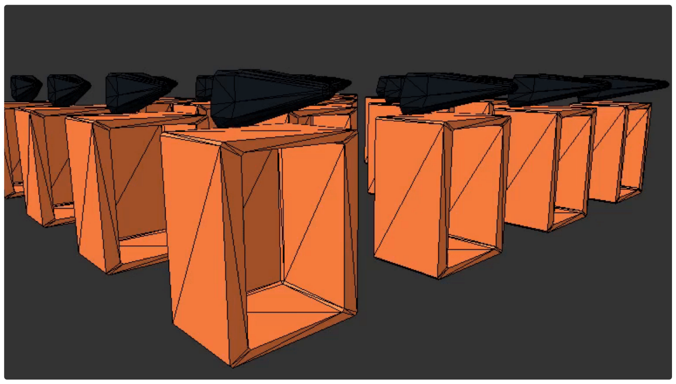
i. `SnippetKinematicContactReport` - Attempt to generate contacts using kinematic and static box geometries
ii. `SnippetKinematicContactReportGPU` - GPU version of `SnippetKinematicContactReport`
iii. `SnippetKinematicMeshContactReport` - Attempt to generate contacts using kinematic and static triangle mesh geometries
iv. `SnippetKinematicHullContactReport` - Attempt to generate contacts using kinematic and static convex hulls
v. `SnippetKinematicHullContactReportGPU` - GPU version of `SnippetKinematicHullContactReport`

b. `physx_collision_test`
Contains the following scene

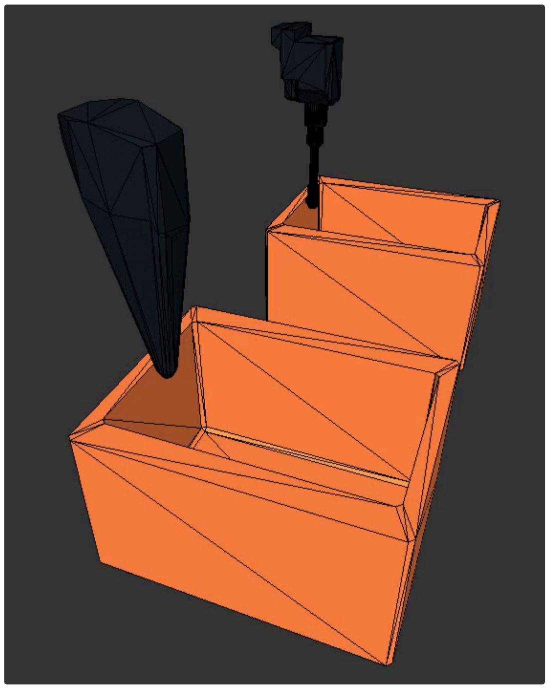
PrimitivesScene



MeshColliderScene



MeshGeomCompareScene



Building and running the tests:

- Ensure that the `collision_tests` container is running
- Exec into the container: `ws-exec collision_tests`
- Build all the tests by running `build-collision-tests`
- FCL's mesh based tests can be configured through a config file at: `/data/workspace/collision_tests/fcl/config/config.ini`

```
[parameters]
# Environment object filepath (multiple instances of the same mesh)
envObjPath = /data/workspace/collision_tests/meshes/tote/tote.obj

# Query object filepath (multiple instances of the same mesh)
queryObjPath = /data/workspace/collision_tests/meshes/gripper/gripper.obj

# Size of the environment grid (Default: 1)
envSize = 1

# Size of the query grid (Default: 1)
querySize = 1

# Query object offset to control clutter index (Default: 0.0)
# Offset(0.0) -> ClutterIndex(0.98f)
# Offset(41.0) -> ClutterIndex(0.51f)
# Offset(85.3) -> ClutterIndex(0.042f)
# Offset(85.7) -> ClutterIndex(0.f)
queryCenterOffset = 0.0

# Enable or disable exhaustive search (Default: false)
exhaustive = false
```

- PhysX test `physx_collision_test`, can be configured through a config file at: `/data/workspace/collision_tests/physx/config/config.ini`

```
[parameters]
# Index of the scene to be loaded (Default: 1)
# Index 2: Primitives Collider
# Index 4: Mesh Collider
# Index 8: Mesh Geometry Compare
sceneIndex = 4

# Frame rate (Default: 60Hz)
timeStep = 60

# Number of threads to be used for parallel computation (Default: 1)
numThreads = 1

# Number of environment objects (Default: 1)
envSize = 1

# Number of query objects (Default: 1)
querySize = 1

# Scale factor for the environment (Default: 1.0)
envScale = 1.0

# Scale factor for the query (Default: 1.0)
queryScale = 1.0

# Spacing for the Signed Distance Function (Default: 0.0)
sdfSpacing = 0.0

# Query object offset to control clutter index (Default: 0.0)
# Offset(0.0) -> ClutterIndex(0.99f)
# Offset(0.41) -> ClutterIndex(0.51f)
# Offset(0.853) -> ClutterIndex(0.042f)
# Offset(0.857) -> ClutterIndex(0.f)
queryCenterOffset = 0.0

# Save frame data to file
saveFrameData = false

# Enable or disable GPU acceleration (Default: false)
enableGPU = false

# Enable or disable rendering (Default: false)
enableRender = true
```

- A particular test can be run as follows: `run-test <TEST-NAME>`
- In order the list all the tests that are available, you can TAB-complete after typing `run-test`. Example

```
root@17c337b68ecd:/data/workspace# run-test
fcl/fcl_mesh_collider                               physx/snippets/SnippetKinematicContactReportGPU
fcl/fcl_mesh_geom_compare                           physx/snippets/SnippetKinematicHullContactReport
fcl/fcl_primitive_collider                          physx/snippets/SnippetKinematicHullContactReportGPU
fcl/test_render                                     physx/snippets/SnippetKinematicMeshContactReport
physx/physx_collision_test                          physx/snippets/SnippetSaveConvexHullAsOBJ
physx/snippets/SnippetKinematicContactReport
```

Loading the assets into FCL and PhysX:

Mesh geometries stored as STL files need to be converted to OBJ format before loading into FCL. OBJ files are loaded into FCL using this function: https://gitlab.ocado.tech/advanced-technology/pre-production/hacc_physx_mvp/-/blob/main/collision_tests/fcl/include/utils.h#L215-278. It extracts the vertices and triangle indices from the OBJ file

In order to load the STL into PhysX, we first need to extract the triangle and vertex data from the STL file. This is done using a python script: https://gitlab.ocado.tech/advanced-technology/pre-production/hacc_physx_mvp/-/blob/main/collision_tests/python/process_stl.py. This creates two files: `vertices.csv` and `indices.csv` for each STL file. Then, its only a matter of reading the vertex and index data from these files into PhysX which is made available during the creation of mesh geometries within PhysX. The files are read with the help of this function: https://gitlab.ocado.tech/advanced-technology/pre-production/hacc_physx_mvp/-/blob/main/collision_tests/physx/source/CollisionTestUtils.cpp#L322-359

Evaluation

Evaluation Metrics:

Step Timings (ms)

Step timing is defined as the time taken to complete collision checking for a single iteration of the simulation / planning algorithm. It is measured in milliseconds

For FCL it represents the time spent in all the calls to `manager->collide()`:

```
double totalTime(0);
for (size_t i = 0; i < scene.query.size(); ++i) {
    DefaultCollisionData<double> query_data;
    timer.start();
    scene.manager->collide(scene.query[i], &query_data, DefaultCollisionFunction);
    timer.stop();
    totalTime += timer.getElapsedTime();
}
```

For PhysX it represents the time spent in call to `PxScene::simulate()`:

```
PxU64 startTime = SnippetUtils::getCurrentTimeCounterValue();
mScene->simulate(mTimeStep);
mScene->fetchResults(true);
mStepTime = SnippetUtils::getElapsedTimeInMilliseconds(SnippetUtils::getCurrentTimeCounterValue() - startTime);
```

Clutter Index (ρ)

Let the number of frames/iterations for a simulation / planning algorithm $= N$. Let the number of contacts generated in the i^{th} frame/iteration be denoted by $C(i)$. Then the clutter index ρ can be defined as follows:

$$\rho = \frac{\sum_{i=1}^{N} \mathbb{I}(C(i) > 0)}{N}$$

Where $\mathbb{I}$ is the indicator function. A value of ρ can range between 0 and 1. Higher value implies more clutter

Collision Precision

Since PhysX cannot generate contacts with two concave meshes (triangle mesh geometries in our case), we have to approximate the mesh geometry as a convex hull. The approximation leads to false positives being generated during collision checking. The collision precision quantifies how much does the approximation affect the libraries' ability to correctly identify collisions. With the results from FCL (triangle mesh geometries) as groundtruth, the collision accuracy for other libraries is measured as follows:

$$\text{Collision Precision} = \frac{TP}{TP + FP}$$

where,

$TP \rightarrow$ Number of colliding frames / iterations reported as colliding

$FP \rightarrow$ Number of non-colliding frames / iterations reported as colliding

The convex hull approximation overestimates the total number of colliding frames and will never actually miss an actual collision (i.e the Collision Recall is always 1 as it never produces false negatives)

Results

Primitive Colliders:

Total Number Of Collision Objects vs Mean Step Timings (ms):

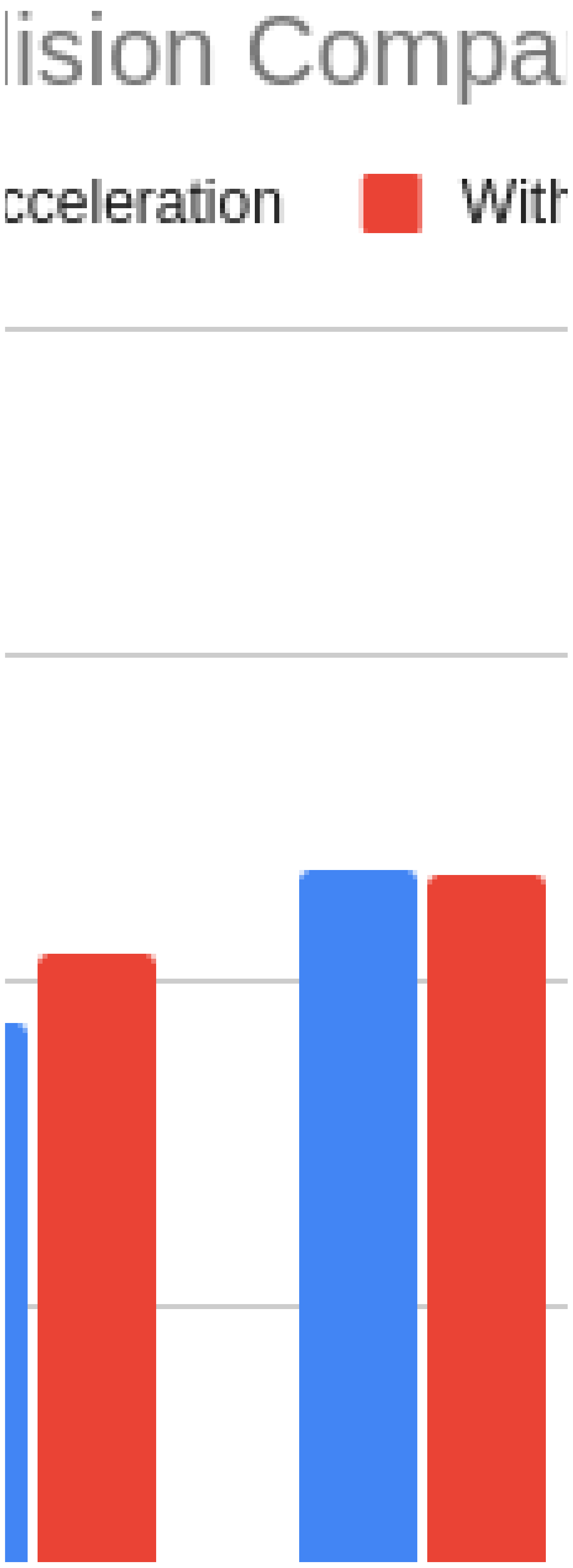


Figure 1: Comparing collision checking timings between PhysX CPU and GPU for primitive collision objects

Mesh Colliders:

Total Number Of Collision Objects vs Mean Step Timings (ms):

High Clutter Index (0.98)

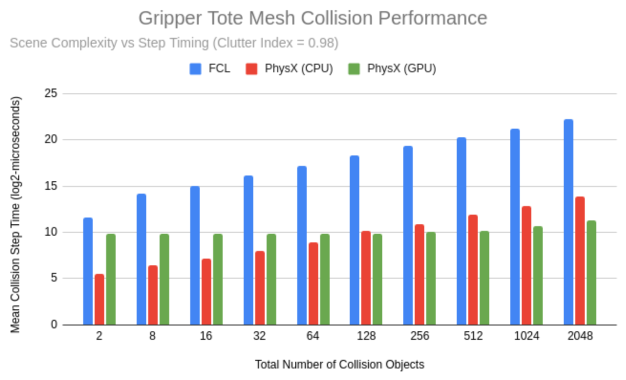


Figure 2: Comparing collision checking timings between FCL, PhyX (CPU) and PhysX (GPU) for mesh geometries with high clutter index

Low Clutter Index (0.04)

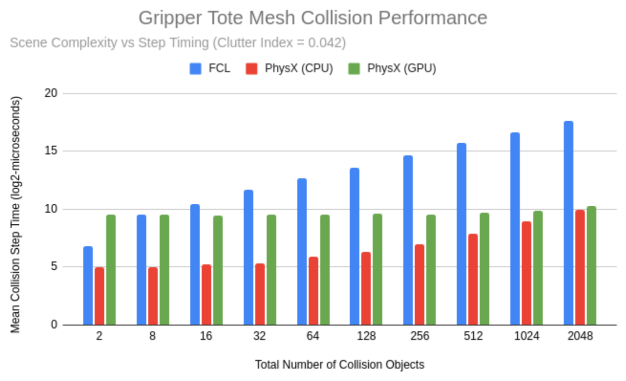
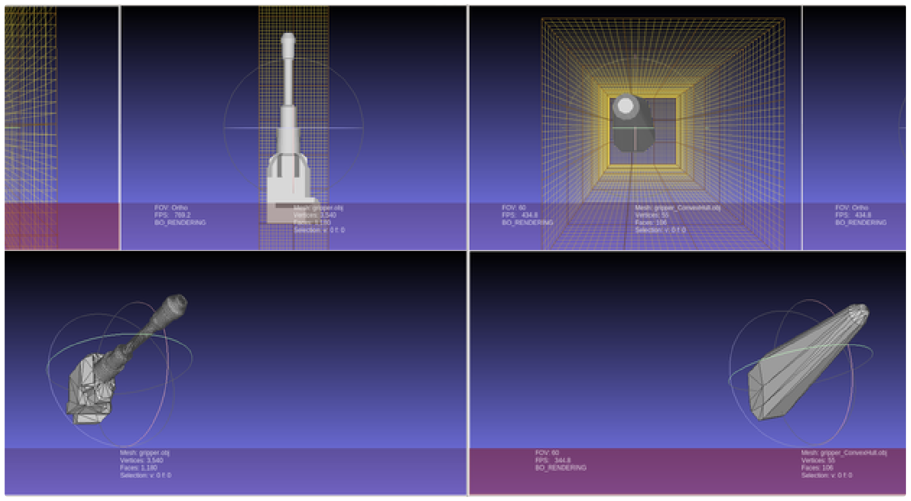


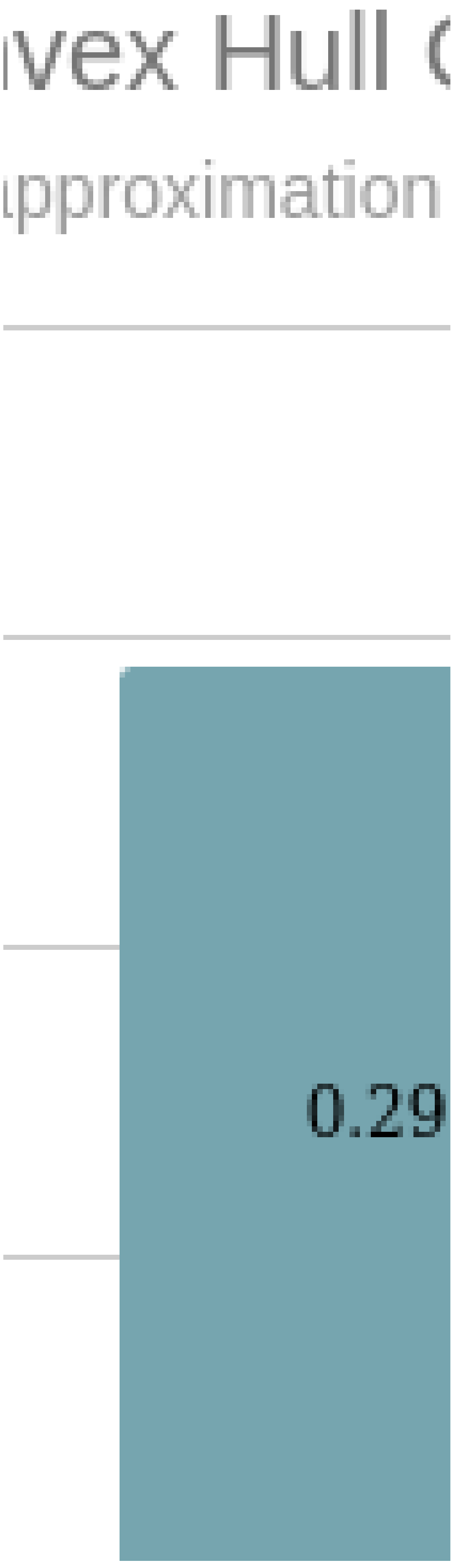
Figure 3: Comparing collision checking timings between FCL, Physx (CPU) and PhysX (GPU) for mesh geometries with low clutter index

Convex Hull approximation:



```
FOV: Ortho
FPS:   769.2
BO_RENDERING

Mesh: gripper.obj
Vertices: 3,540
Faces: 1,180
Selection: v: 0 f: 0

FOV: 60
FPS:   434.8
BO_RENDERING

Mesh: gripper_ConvexHull.obj
Vertices: 55
Faces: 106
Selection: v: 0 f: 0

FOV: Ortho
FPS:   434.8
BO_RENDERING

Mesh: gripper.obj
Vertices: 3,540
Faces: 1,180
Selection: v: 0 f: 0

FOV: 60
FPS:   344.8
BO_RENDERING

Mesh: gripper_ConvexHull.obj
Vertices: 55
Faces: 106
Selection: v: 0 f: 0
```

Visual comparison of the original gripper geometry and convex hull approximation (MeshLab)


ıvex Hull (
ıpproximation
0.29

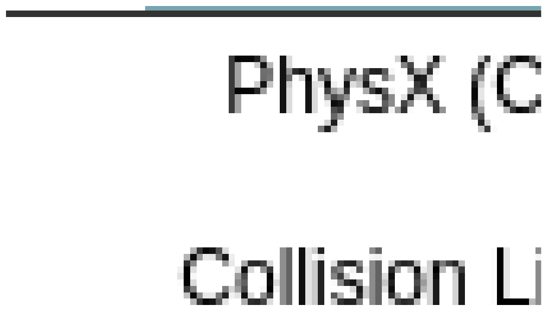


Figure 4: Collision precision with convex hull approximation using FCL, PhysX (CPU and GPU)

Boolean Collision Result Per Frame

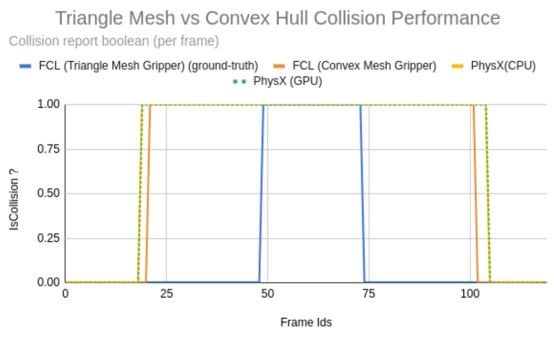


Figure 5: For the pick action in the MeshGeomCompareScene, this graph displays the frames where the different libraries incorrectly report collision due to the convex hull approximation

Convex Hull approximation: FCL's collision timing

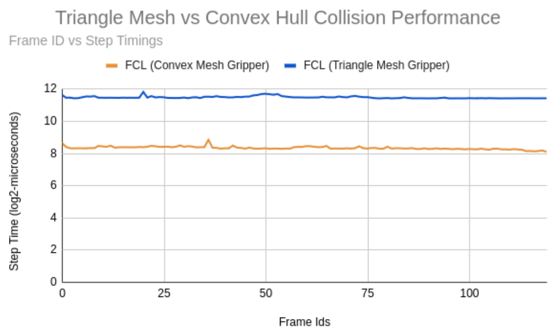


Figure 6: Comparing FCL's collision checking performance when using convex hull approximation over the triangle mesh geometry

Single Contact Generation: FCL's collision timing

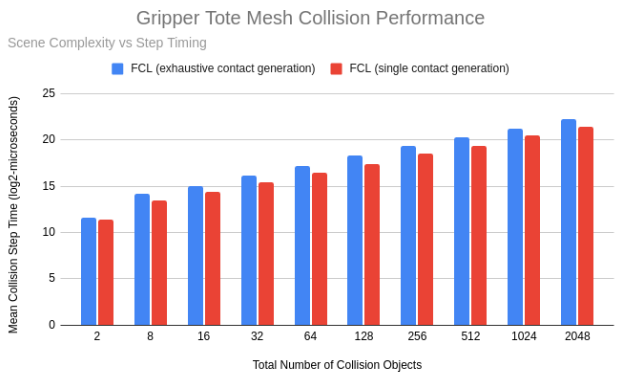


Figure 7: FCL collision checking timing with exhaustive contact generation (blue) and single contact generation (red)

Discussion

The analysis of the primitive colliders (with kinematic boxes and static spheres and capsules) revealed a margin after which the PhysX GPU acceleration starts to gain performance benefits over the CPU implementation. This margin was around 12,000 primitive collision objects. There are three main reasons for the lack of performance from the GPU acceleration with relatively fewer collision objects:

1. *Memory Transfer Overheads:* Everytime there is a change to an object's transform, the new data has to be copied over from the host(CPU) to the device (GPU)
2. *Kernel Launch Overheads*: Stages from the collision checking pipeline that are implemented on the GPU, such as the *broadphase* and *narrowphase*, require a kernel launch from the host in order to run the computations on the device. This involves just-in-time compilation of the kernel code, resource allocation for the kernel, loading the kernel in the GPU's instruction memory amongst other things which build up the launch overhead
3. *Synchronization between kernel launches*: As an example, data for the narrowphase kernel launch depends on the results from the broadphase kernel and therefore cannot be launched asynchronously. In this case, the contact pair information needs to copied back to the host, pair filtering is performed on the host and the updated pairs are then copied to the appropriate device memory space before the narrowphase kernel can be launched

With fewer objects in the scene / smaller "batch-size", the speedup from the GPU parallelization is diminished by the overheads mentioned above. As the objects / batch-size is increased the relative-speedup from the CPU implementation becomes evident.

Taking this result forward, a similar threshold for mesh geometries is observed in Figures 2 and 3 at around 128 collision objects / "batch-size". It is also evident that with fewer objects and low clutter index, FCL performs better than PhysX (GPU). From the trends, FCL seems to be more sensitive to clutter index than PhysX and FCL would asymptotically perform better than both PhysX CPU and GPU as the clutter index goes to zero. In general, for high clutter index simulations, PhysX CPU consistently provides about 65-70X speedup over traditional FCL while the speedup from GPU version ranges from 4-4000X depending on the number of collision objects in the scene (or batch-size in single query).

In order to enable contact generation for collisions in PhysX, the gripper mesh had to be converted to a convex hull approximation. This approximation speeds up the collision checking process in FCL as can be seen in Figure 6 by about 16X. However, this leads to a lower collision precision (Figure 4) as certain transforms are reported as colliding with the hull approximation while the true geometry is in fact in free-space. The convex hull approximation is also partially responsible for the speedups from PhysX.

Since we set the num_max_contacts to 1 in the request for a call to checkCollision, it was interesting to see how much does FCL's collision checking performance degrade when requesting exhaustive contact generation. From Figure 7, it is evident that as the number of collision obejcts in the scene increase, requesting more contacts slows FCL's collision checking performance. With ~2000 mesh geometries requesting a single contact generation is around 2X faster than an exhaustive contact generation. However, with a single tote and gripper, there is not alot of difference. Exhaustive generation of contacts might have an impact on penetration depth computation.

Complete results with the data can be found here:

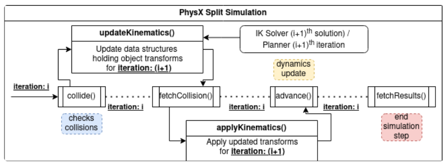
1. Primitive Collider:
   Sign in to access Google Drive Spreadsheet
2. Mesh Collider (clutter index: 0.98):
   Sign in to access Google Drive Spreadsheet
3. Mesh Collider (clutter index: 0.042):
   Sign in to access Google Drive Spreadsheet
4. Convex Hull Performance comparison:
   Sign in to access Google Drive Spreadsheet
5. FCL - Single Contact Generation performance:
   Sign in to access Google Drive Spreadsheet

Integration Plan

The plan will closely follow the architecture described here: Architecture Diagram | Sub Option 1: Implement the collision environment on the CPU

Questions

- The way we currently handle IK (from what I saw in `URKinematics.cpp`) is that we take in a bunch of poses and call `calculateIK`. Then, we call `solveIK`, sequentially, for each of those poses. Finally, once we get the calibrated and ranked IK solutions, we update `robotState` sequentially and pass a reference to that in a call to `isStateValid()`. **So the question is, how can we send a batch of calibrated solutions when making a call to `isStateValid()` ?**
  Answer:
- **Why do we need to do convex hull approximation ?**
  Answer: Because PhysX does not directly support contact generation for kinematic triangle mesh <> static triangle mesh interactions. It is however supported with an additional computation of signed-distance-fields for kinematic triangle meshes on the GPU
- **Is there a better (more precise) way of representing a concave mesh instead of a single convex hull ?**
  - *Answer: A concave mesh can be represented as an approximation using smaller decomposed convex meshes (V-HACD : Volumetric Hierarchical Approximate Convex Decomposition) [4]. However, there is no native support for v-hacd in PhysX. There is an example to integrate v-hacd into PhysX here: https://github.com/mikedh/v-hacd-1/blob/master/DebugView/TestHACD.cpp. PhysX 5 also introduces collision checking using SDFs (only supported on the GPU) which does not require performing a convex decomposition. However, they work for kinematic and dynamic actors only (not supported for static actors) and are generally less performant than non-SDF convex hull approximation with GPU*
- **Does FCL perform collisions serially ? Is this an implementation issue or an algorithmic issue ?**
  - *Answer: Yes FCL performs collisions serially at least for the broadphase which involves a sequential BVH tree traversal (code reference). I think this is just an implementation issue as the results from different parts of the trees are independent of each other, this could be converted into a multi-threaded query. The PABP algorithm implemented in PhysX (this is the default BP algorithm) is a multi-threaded version of the automatic-box-pruning algorithm (similar to the one implemented in FCL)*
- **Can we use split-simulation (`PxScene::collide() + PxScene::advance()`) instead of the standard sub-stepping (`PxScene::simulate()`) ?** Split-simulation splits the simulation into separate collision query and dynamics update steps. So essentially, we could potentially omit the dynamics update step as it is not required in motion planning.
  - *Answer: Yes it is possible to use split-simulation. However, based on a comment from one of NVIDIA's developers, the call to `PxScene::advance()` is required even with only kinematic and static actors. So we cannot really omit the dynamics even if we don't need it. This is in effect equivalent to calling `PxScene::simulate()`. However, we can leverage some parallelism for updating the kinematic targets for the next simulation step using split-simulation as such:*



-

References

[1] Parallelized Radix Sort - Google Code Archive - Long-term storage for Google Code Project Hosting. (*Merrill, D., & Grimshaw, A. (2011). High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters, 21(02), 245-272.*)

[2] Incremental Sweep and Prune - http://www.codercorner.com/SAP.pdf

[3] Expanded Polytope Algorithm - EPA (Expanding Polytope Algorithm)

[4] V-HACD -
Sign in to access Google Drive File

(*Mamou, Khaled, and Faouzi Ghorbel. "A simple and efficient approach for 3D mesh approximate convex decomposition." In 2009 16th IEEE international conference on image processing (ICIP), pp. 3501-3504. IEEE, 2009.*)