COPPERODE MIDDLEWARE for PANDA 3D

Version 1.2.1

WHAT IS THIS FRAMEWORK?

Long story short, it is my attempt to patch the missing features in ODE and ease the world management for the purposes of the project I'm working on. Thus you should be aware it is not meant to fit everyone's needs, so I can't guarantee it will fit yours.

IMPORTANT: Consider this framework a work in progress. It's something I extract from the larger code I'm working on, so I can't ensure the API stability nor regular updates.

ODE is a nice engine, but it lacks some features, which are crucial for certain computer game genres. Some of those features are available out-of-the-box in game-physics middlewares, such as PhysX, Bullet or Havok. However, since Havok is expensive, Physx's license is ugly and Bullet's Character Controller is not good enough, I have decided to stick with ODE. It gives me the most flexibility and it's available with Panda in one package, allowing me to avoid writing wrappers and messing around with C++.

The most important features you'll find here are:

1. World Manager, which makes it simpler to use ODE with Panda and Python.
2. Basic classes for Static, Kinematic and Dynamic objects
3. Kinematic Character Controller (KCC)
4. Continuous Collision Detection (CCD) support
5. Player controller (PCC) for FPP games using the KCC
6. Object picking and carrying mechanics
7. Firearm mechanics with OOTB support for pistols, automatic guns and shotguns
8. Explosion mechanics
9. Area Triggers
10. Door
11. Map loading example
12. Moving platform example
13. BitMask generator
14. Includes visualizations using FenrirWolf's code (Thanks!)

WHAT IS THIS DOCUMENT?

It's supposed to give you an overview of what is available in this framework, but it is not a comprehensive manual. If you want to use this code, you should really read the source (and you will probably want to make some changes to it to fit your needs). There are lots of comments included and it's not that long.

Also, I strongly encourage you to read the whole code for every new version (major and minor). It might be a pain, but there can be changes you do not want to miss.

I can't guarantee there are no mistakes in this document just like I can't guarantee there are no bugs in the code. If you find any, please inform me.

SETTING UP A WORLD

To utilize this framework you'll need to create an odeWorldManager, add objects to it and start the simulation.

There are many possible ways you can setup your map for simulation. The map.py file shows the approach that I use for my game, to make it easy for me to place objects around in Blender. It features a system for automatic object creation based on tags. The .blend file for the map is available in ./graphics/blender directory, so if you'd like to use and extend the code in the map.py file, I encourage you to take a look into that blend file.

The World Manager

The World Manager (odeWorldManager class) is the center of this Framework. It's purpose is to add, remove and update the objects in the simulation as well as handle collisions between the objects. It contains the OdeSpace and

the OdeWorld that the objects will use. It also provides mechanisms for getting from the geom to the object it is used by.

The World Manager doesn't use the Panda's AutoCollide mechanics. It uses a custom handleCollisions method callback. This callback is written in Python, so it might be slower than AutoCollide, but it's also much more flexible.

> ***Note:***
> *Panda 1.7.0 (and earlier) has a bug that causes the odeWorldManager's collision handling to leak memory. Please use this Framework with newer Panda versions or builds from the Build Bot.*

By default, the World Manager enables ODE's auto disable mechanics, which puts ODE bodies to sleep when they're not used (i.e. the forces applied to them are below a certain level) for a specific time (or number of simulation frames). It's possible to set when the bodies go to sleep. To do this, consult the odeWorldManager.py file.

It also, by default, uses the ODE's more precise friction model. Note, that this friction is still not an actual Coulomb friction, but just a simple approximation. However, it's still better than the ODE's default model and it should be enough for most uses.

Be aware that ODE doesn't support rolling friction, no matter the model set.

Basic Object Types

Every object contains a Node (self.nodePath variable), which is used for rendering, and an OdeGeom (self.geom), which is used for collision detection. Dynamic Objects also contain an OdeBody (self.body).

There are currently 5 basic object classes by default:

1. Physical Object (physicalObject class)

   The most primitive and not meant to be used in any way by itself. It's purpose is for other objects to inherit from it.

2. Static Object (staticObject class), inheriting from physicalObject.

   Meant for static environment, like buildings. It's not updated in any way because it doesn't move. Contains set*Geom methods (except for setRayGeom) for simple geom creation.

3. Kinematic Object (kinematicObject class), inheriting from staticObject.

   Meant for animating with Panda's Interval system. It's update method sets the position of it's ode geom relative to the position of it's panda node. In other words, in this case, the collision detection geom follows the rendered object, which can be animated by Panda's mechanisms and Python scripts. This type of object is the most useful for in-engine cut-scenes or scripted events.

   Additionally, the kinematic object calculates the velocity in every frame. This allows for kinematic objects to be used as moving platforms, carrying other objects (dynamic objects and the KCC).

   Note, however, that currently the implementation of moving platforms lacks a little bit. It works well with horizontal movement, but vertical is problematic. While the KCC works well with it (so you can make an elevator for the player) the dynamic objects shake and jump around when on a platform moving downwards.

4. Dynamic Object (dynamicObjectNoCCD and dynamicObjectCCD classes), inheriting from staticObject

   Represents a body controlled by the physics engine. The update method sets the position of it's node to

the position of it's body and geom. In other words, the position of the rendering object and the physical object is controlled by ODE itself, with little to no influence from the programmer (other than setting forces, joints and the like).

There are, additionally, two kinds of dynamic objects – with Continuous Collision Detection (CCD) and without. You can read more about my simple CCD implementation later in this document.

5. Ray Object (rayObject class), inheriting from physicalObject.

   A special object class meant solely for rays.

The Objects are meant to automate the process of updating positions and rotations in ODE simulations with Panda, and to make it easier to get from the Geom/Body to the gameplay mechanics assigned to it.

> ***Note:***
> *Since version 1.0 there no longer is an OdeGeomData class, which was previously said to soon be removed. Now the system operates solely on Objects, to which it gets directly from OdeGeoms.*

Objects you create don't need to inherit from any of my physical objects, as long as they contain the following:

- collisionCallback(self, object1, object2) method – called when this object collides with something. Object1 is this objects, object2 is the one this object collided with.

- selectionCallback(self, character, direction) method – used by the Player Controller for selecting objects in the world. Character is the character that attempts to use this objects, direction is the direction the controller's picker ray aims at.

- objectType variable – tells the World Manager more precisely how to react to this object. Possible (used and understood by the system out of the box) values are: “static”, “kinematic”, “dynamic”, “ccd”, “trigger” and “ray”.

  Note:
  Previously, there was the isTrigger variable in the OdeGeomData class, which no longer exists. Now trigger is a separate object type of it's own.

You can add other variables, callbacks and supported object type values at will, obviously.

BitMask management since version 1.2.0

Since 1.2.0 the odeWorldManager.py file includes a new, fully automated, method for managing and generating BitMasks.

CONTINUOUS COLLISION DETECTION

What is it and what it is useful for

Continuous Collision Detection (CCD) is basically meant to ensure that there is no tunneling in the simulation. Tunneling is a frequent problem with discreet collision detection, because that method only checks for collisions in actual frames, and not between them. Because of that, a small and fast moving object can be in front of a wall in one frame, and behind it in the next. In such case, the collision with the wall, that should have happened between frames, will go unnoticed by the engine, and the object will keep moving instead of stopping on the wall.

The problem is that the object doesn't need to travel at very high velocities – it doesn't need to be a bullet, it might be, for example, a thrown grenade; as long as it's small enough it will tunnel. The bigger the time step the more visible the problem is, but for objects small and/or fast enough even time steps of 1/200 sec won't help.

CCD that I've implemented is based on a very simple method, but at the same time probably the only one that

doesn't require messing around with ODE's guts.

I get the previous frame position of the body, the current frame position, and then I put a number of, so called, “helper geoms” between those positions, to fill the space. Once they're there, the normal collision detection does it's job. Using a callback, the dynamic object is informed about all collisions detected using the helpers. Then, of all the colliding helper geoms, it finds the one nearest to the previous position of the body and moves the body there.

Now, here's one of the most important changes between 1.0 and 1.2. Previously, the bodies' velocity was altered after collision handling to keep it below the level at which CCD kicks in. This prevented the objects from getting permanently stuck to the surfaces they collided with, but it also caused the objects to bounce off surfaces in a non-realistic way, because a lot of the initial momentum was lost. This is, however, no longer the case. I won't go into details here (the comments in the source and the code itself are meant for that), but the point is the objects now (mostly) bounce correctly, while still not getting stuck on stuff.

Compare how grenades and balls bounce off walls in 1.0.1 and in 1.2.x to see the difference.

The downside of this improvement in bouncing is that theoretically, in certain situations (like very, very small geoms on bumpy surfaces), you may experience tunneling. Still though, this can be easily eliminated by tweaking a couple of variables which control the new behavior. You can read more about that in the code.

You can enable visualization of the CCD process by setting the showCCD variable in the dynamicObjectCCD class to True. The visualization now uses the neat FenrirWolf's wire geoms.

Limitations

This CCD implementation is still quite simple. While version 1.2 removes it's most annoying flaw – incorrect bouncing – there are still a few less problematic ones left.

Currently only Boxes and Spheres are supported. Compound shapes are also not supported and they might never be. Trimesh is, obviously, not supported as well, and it will never be for sure (however, it's generally not a good practice to use trimesh too extensively, and a really bad practice to use it for dynamic objects whenever there's a way to avoid it).

With really high velocities it might prove too slow. It's not the most efficient approach to CCD and it's Python. It can choke when it needs to add a large number of helper geoms to the simulation between frames.

Also, note that the CCD is limited to dynamic objects, so making sure your fast moving kinematic objects don't tunnel is up to you. This shouldn't be that much of a problem, though. Kinematic objects are usually used for scripted events and in that case you usually know what exactly is going to happen with those objects.

KINEMATIC CHARACTER CONTROLLER

The kinematic character controller is the standard approach to handling characters in computer games. Similar ones are available in Physx, Bullet and Havok (this one also has a decent dynamic controller implementation available, though).

There are actually two possible approaches to character controllers – dynamic and kinematic. The dynamic method uses a physical body (usually capsule-shaped) which is constrained to stand up straight (usually with a joint connected to the world) and is then pushed around with forces. The kinematic method, on the other hand, is a concept much closer to the “physics-less times” in games. Instead of a body with a collision detection shape, we only have the collision detection shape (or a “kinematic body”, if the engine supports those, which ODE doesn't), which we move around using some custom code.

At first, the dynamic controller might seem better and simpler to make, but actually it's very difficult to prevent it from behaving like an inert body. The number of constraints needed to be put on it makes it pointless.

Features

The KCC should provide a stable foundation for characters in FPP, TPP and other game genres that use humanoid characters. Here's a list of the features supported:

- 2 dimensional movement
  - Walking, running
- 3 dimensional movement
  - Walking steps
  - Flying, which can be used for ladders, swimming or no-clip functionality.
- Jumping, with space awareness
  - Ceiling penetration prevention
  - Automatic crouching when jumping into small spaces.
    If the space is too small to even crouch in it, and yet somehow the KCC fits into it, it will automatically return to the previous stable position.
- Crouching, with space awareness
  - Standing up is locked (without the need for area triggers) when there's not enough space to stand.
  - Ceiling penetration prevention when pushing dynamic objects through small spaces. If a dynamic object gets stuck (usually because of the friction) the KCC will walk through it, and not pop out of the tunnel through it's ceiling.
- Kinematic and dynamic Moving Platforms

Changes between 1.0.1 and 1.2.0

It is, finally, possible to easily set the size of the KCC. Using one simple method you can set the walking and crouching height of the KCC, it's radius and the maximum steps size. Because of that I've also changed the sample map. The KCC in the sample is now 1.85 meters high, while previously it was of unknown (but larger) size. That resulted in wrong scale of basically everything in the world (even though proportions were mostly correct).

Another change is support for moving platforms. The KCC can now stand on kinematic objects and be carried by them, which allows for the creation of elevators, conveyor belts, escalators, trains and so on. Additionally, the KCC can also be carried by dynamic objects so it can, for example, stand on a dynamic box inside a train and still correctly move with the train's floor.

Limitations

There's no slope limiter. My game is set in urban environment, which renders a slope limiter irrelevant. Because of that, it's unlikely I will write it. Still, it shouldn't be too difficult.

PLAYER CONTROLLER

The playerController class inherits from the KCC class.

It contains additional features, such as:

- FPP camera with 3 versions of mouselook,
- Picking up and using objects in the environment using a raycast,
- Carrying objects,
- Sitting (on chairs, in cars etc.),
- Flashlight
- Smooth camera movement when walking stairs and in stand-to-crouch transitions.

Using objects in the world

Usually in FPP games there are certain pickable or usable objects that the player can interact with. This interaction is usually done with a Raycast and that's the case with the Player Controller.

When the user aims at an object and presses the “use” key (right mouse button in the sample) the player controller gets the nearest detected object in front of the camera. Once this object is found, the player controller calls the object's selectionCallback.

The picking mechanics was changed in 1.2.0 to no longer use the inefficient in-place raycasts. Instead, it now uses the same mechanics as the gun class – the aiming ray is always present and collects hits in every frame, finding the closest one as they come.

Picking up and carrying objects

Pickable objects (or “pickables”) are a kind of usable objects that can be picked up by the player. The default behavior for the player in the sample is to carry such objects in front of the camera (like in Half Life, Deus Ex etc.).

There's a method called placeGeomInFrontOfCamera in the playerController class, which is responsible for keeping the carried object where it's supposed to be. Inside of this method you can put a set of “if/elif” sequences to control how certain kinds of objects behave, depending on your needs and your inventory implementation. I don't put any ootb mechanism here to not constrain your inventory design.

The way the player carries objects can be controlled with a series of settings.

The carried object can have it's collisions enabled, in which case it will interact with (push away) other objects in the world, or disabled, in which case it will have no effect on the environment.

The playerController.jiggleHeld variable controls whether the carried object is affected by the environment. If it's set to True, the carried object will “shake” when touching other objects in the environment. If it's False, the environment will have no effect on the carried object. Obviously this setting is only relevant when the carried object is kept enabled.

By default, the carried object is set transparent, so it doesn't block the view for the player.

The placement of the carried object can also be controlled. It can be kept at a constant distance from the character's capsule (constraining the carried object's movement only to the player's local Z axis), or it can be kept directly in front of the camera when the player looks up. This can be set with the curveUp variable inside the playerController.placeGeomInFrontOfCamera method.

MORE ON PICKABLE OBJECTS

These can be objects like boxes, which can be carried around, thrown or stacked by the player; or objects that the player keeps hidden in their inventory, like guns.

The pickable objects are, by default, dynamic objects. They can be easily extended to support complex inventory systems.

FIREARMS

The framework provides a “gun” class, which you can use to make most kinds of firearms. The mechanics is based on rays, so it's not fit for rocket launchers and the like.

You can set your gun to be automatic or semi-automatic, and you can set it to use multiple rays for a shotgun-like functionality. Obviously, you can have an automatic shotgun as well.

For automatic guns you can control the rate of fire, and for shotguns – the dispersion of the shots.

The gun class itself acts like a pistol by default (semi-automatic, single ray). There are also two derived classes for presentation: assault rifle (automatic, single ray) and shotgun (semi-automatic, multiple rays).

It's possible to get multiple hit points from a single ray (this is, actually, the only behavior right now). In other words, the guns shoot through walls and other objects. You might find this fact useful (or not, depending on your design).

Implementation details and limitations

For now, it's not possible to control the accuracy of single-hitpoint (single ray) firearms. The ray will always point directly at the center of the screen. Also, there's no recoil.

The guns can only be used by the Player, although support for NPCs will obviously come.

The implementation I used for the gun class is probably a little unusual. Instead of doing a raycast exactly when the Player hits “fire”, I keep the rays sticking from the camera all the time (for as long as the Player holds the gun) and just collect the hits from every simulation pass and then process or clean them depending on the value of the isShooting variable (which sets to True when the Player presses fire and to False when the Player releases the fire button (for automatic) or when a frame passes (for semi)).

The limitation of this implementation is that it follows the camera in a less precise way. This shouldn't be a problem for most games, but if you want to make a game as fast as quake, it surely will be.

The reasoning behind such choice is that ODE doesn't support ray casting by itself, and my odeWorldManager.doRaycast() method is simply too slow for the purpose of shooting. While it can handle a single, semi-automatic pistol, it simply chokes on shotguns and automatic weapons, let alone multiple characters running around guns blazing. So I sacrificed some accuracy for a lot higher performance.

TRIGGERS

The idea of a trigger is twofold. There's *“trigger”*, as a possible value for the *Object.objectType variable, and there's the odeTrigger class. These two should not be confused with each other.

The “trigger” objectType tells the odeWorldManager that it should treat a certain object as a trigger, or a ghost, to be more precise. That is, when a collision is detected with this geom, the odeWorldManager calls it's callback, but doesn't setup a contact joint.

The odeWorldManager automatically ignores collisions between two objects of type “trigger”, at well as collisions between an object of type “trigger” and an object of type “static”, “ccd” or “ray”.

The odeTrigger class, on the other hand, represents an *Area Trigger*. The purpose of this class is to provide awareness of what objects enter, exit and remain inside of a certain area. Obviously, the odeTrigger's objectType value must also be set to “trigger”.

The odeTrigger acts like any other Object and it inherits from the physicalObject class.

You can set the message for a trigger, that will be sent whenever an object enters or exits the trigger. You can control what objects the trigger detects using BitMasks.

Whenever an object enters a trigger, it will send a message of the given name, suffixed with “_enter”. Whenever an object exits a trigger, it will send the same message suffixed with “_exit”.

Currently, the odeTrigger class inherits from staticObject, which means it cannot be animated. However, if you want to be able to move triggers around, you just need to change inheritance to kinematicObject, and provide a dummy node that you will animate.

CREATING BOX GEOMS IN BLENDER

The framework provides an easy mechanism for creating box geoms from panda nodes. This mechanism uses the model's bounding box to create a geom of the required size. I wrote this to be able to dump trimeshes, because

they proved to be very unstable in some scenarios.

To create a box geom out of a model use the physicalObject.setBoxGeomFromNodePath(node, remove) method. The “remove” argument tells the method whether to automatically remove the object after the box geom has been created.

You will also need a specifically prepared model in Blender. For the best results:

1. Make sure your model has no rotation in edit mode. It should be aligned with it's local axes.
2. Keep scale at 1.0 in object mode.
3. If the geom is meant to be placed under angles in the game, rotate it in object mode and do not apply the rotation to ObData.

BEST PRACTICES

Using TriMesh geoms

Limit the use of TriMesh Geoms. Those are very unstable. The KCC no longer flickers (too much) when colliding with them but other objects, especially spheres with high angular velocities, can “sink” into TriMeshes. Try using the setBoxGeomFromNodePath functionality for larger objects (walls and the like) and use TriMesh only for details that cannot be approximated with boxes.

Bit masks for static environment

For best performance use correct bit masks to make sure your static environment geoms do not collide with each other. For larger maps such collisions can result in a large performance overhead, despite that the World Manager drops them immediately.

Long and thin Box geoms

If you need to have long, thin geoms in your game, then use them with cube-shaped bodies. If you set your bodies with dynamicObject.setBoxBody(), this method will automatically make sure the body is cube-shaped. Long and thin box bodies can gain speed by themselves when they're rotating, which can cause the simulation to crash. More on that here: http://www.ode.org/old_list_archives/2001-November/017372.html