Открытый Динамический Движок
Руководство Пользователя v0.5

Russell Smith
Суббота 29 Мая, 2004

Содержание

1. Введение

Открытый Динамический Движок(Open Dynamics Engine, или ODE) это бесплатная библиотека промышленного качества, предназначенная для симуляции динамики составных жестких тел. Например она хорошо подходит для транспортных средств, существ с ногами и движущихся объектов в виртуальном пространстве. Она быстра, гибка и проста, имеет встроенное определение столкновений. ODE разрабатывает Rusell Smith(http://www.q12.org) с помощью нескольких помощников(http://opende.sourceforge.net/community.html).

Если выражение "симуляция поведения жесткого тела"(rigid body simulation) для вас ничего не значит, то посмотрите Что такое физический SDK(http://opende.sourceforge.net/slides/slides.html).

Это руководство пользователя ODE версии 0.5. Несмотря на маленький номер версии, ODE вполне развита и стабильна.

1.1. Возможности

ODE хорошо подходит для симуляции *составных* структур жестких тел. Составные структуры это жесткие тела различных форм, соединенные друг с другом различными видами сочленений(joint). Например, транспортные средства (у которых колеса соединены с подвеской), существа с ногами (ноги присоединены к телу) или просто набор объектов.

ODE разработана для интерактивной симуляции или симуляции в реальном времени. Хорошо подходит для симуляции поведения движущихся объектов в изменяемом окружении виртуальной реальности. Поскольку ODE быстра, проста в использовании и стабильна, пользователь имеет полную свободу в изменении структуры системы даже во время симуляции.

ODE имеет очень высокую стабильность интегрирования, поэтому ошибки симуляции не должны выходить из под контроля. С физической точки зрения это значит что система не должна "взрываться"(explode) без причины (поверьте мне это случается со многими симуляторами если не быть осторожным). ODE придает большее значение скорости и стабильности чем физической точности.

ODE имеет *жесткие* контакты. Это значит что во время контакта двух тел используется специальное непроникающие соединение. Во многих других симуляторах для представления контактов используются виртуальные пружины. Трудно правильно и без ошибок заставить работать такую систему.

ODE имеет встроенную систему определения столкновений. Тем не менее вы можете игнорировать ее и использовать свою, если хотите. Текущие примитивы, которые могут участвовать в столкновении, это сфера(sphere), прямоугольный параллелепипед(box), цилиндр с верхушкой (capped cylinder), плоскость (plane), луч (ray) и набор треугольников (triangular mesh) - большее количество объектов должно появиться позже. Система определения столкновений ODE обеспечивает быстрое нахождение потенциально пересекающихся объектов с помощью концепции "пространств"(spaces).

Возможности:

- Жесткие тела с произвольной массой.
- Типы сочленений(joint): шарик-в-разъеме(ball-and-socket), сгибание(hinge), скольжение(slider), сгибание-2(hinge-2), фиксация(fixed), угловой двигатель(angular motor), универсал(universal).
- Примитивы, участвующие в столкновении: сфера(sphere), прямоугольный параллелепипед(box), цилиндр с верхушкой(capped cylinder), плоскость(plane), луч(ray) и набор треугольников(triangular mesh).
- Пространства, в которых происходят столкновения: quad tree, hash space и simple.
- Методы симуляции: уравнения движения Лагранжа основанные на моделях Trinkle/Stewart и Anitescu/Potra.
- Используется интегрирование первого порядка(first order integrator). Оно быстрое но не достаточно точное для квантовой физики. Более высокая степень интегрирования появится позже.
- Выбор метода шага времени(time step): это или стандартный метод "большая матрица"(big matrix) или новый метод итераций QuickStep.
- Модель контактов и трения: Основана на Dantzig LCP решении Бараффа(Baraff), хотя в ODE реализовано быстрое приближение модели трения Coloumb.
- Имеет C интерфейс (хотя почти вся ODE написана на C++).
- C++ интерфейс лежит в основе C интерфейса.
- Написано много модулей, и много пишутся сейчас.
- Специфические оптимизации для различных платформ.
- Другие вещи которые я забыл упомянуть ...

1.2. Лицензия ODE

Эта библиотека является бесплатным программным обеспечением; вы можете ее передавать и/или модифицировать в соответствии со следующими правилами:

1. GNU Lesser General Public License(http://www.opensource.org/licenses/lgpl-license.html) опубликованное Free Software Foundation; версия лицензии 2.1, или на ваше усмотрение более поздней версии. Текст GNU Lesser General Public License включен в файл LICENSE.TXT.
2. BSD-style license(http://opende.sourceforge.net/ode-license.html) включена в файл LICENSE-BSD.TXT.

Эта библиотека разработана с надеждой на то что она будет полезна, но без каких бы то ни было гарантий. Для более детальной информации смотреть файлы LICENSE.TXT и LICENSE-BSD.TXT.

1.3. ODE сообщество

У вас есть вопросы или комментарии к ODE? Вы думаете что можете помочь? Пожалуйста напишите в ODE mailing list(http://q12.org/mailman/listinfo/ode).

2. Как инсталлировать и использовать ODE

2.1. Инсталляция ODE

Шаг 1: Распаковать ODE архив.

Шаг 2-4 (альтернативный): Если вы работаете под Windows и используете MSVC, вы можете использовать файлы проекта(project) и рабочего пространства(workspace) в VC6 директории дистрибутива.

Шаг 2: Получить GNU `make` средства. Большинство UNIX платформ поставляются вместе с ним, хотя иногда имеют название `gmake`. Версия для Windows доступна здесь http://q12.org/ode/bin/make.exe.

Шаг 3: Отредактировать настройки в файле `config/user-settings`. В этом файле дан список поддерживаемых платформ.

Шаг 4: Запустить GNU `make` для конфигурации и сборки ODE и графических тестовых программ. В процессе конфигурации создается файл `include/ode/config.h`.

Шаг 5: Инсталлировать ODE библиотеку в вашу систему с помощью копирования `lib/` и `include/` директорий в необходимые места, напр. в Unix:

- `include/ode/ --> /usr/local/include/ode/`
- `lib/libode.a --> /usr/local/lib/libode.a`

2.1.1. Сборка и запуск ODE тестов на MacOS X

ODE использует XWindows и OpenGL, для визуализации имитируемых сцен. Прежде чем собирать ODE, вам надо инсталлировать Apple X11 сервер и X11SDK (обычные средства разработчика).

Их можно взять у компании Apple по адресу http://www.apple.com/macosx/x11. ЗАМЕЧАНИЕ: маленькая ссылка в нижнем правом углу страницы в SDK.

Установив это программное обеспечение дальше можно следовать инструкциям по обычной инсталляции ODE.

Поскольку ODE использует X11, то вам надо запустить X11 сервер (который должен быть у вас уже установлен, в папке Applications).

Если вы запускаете тестовые приложения в XTerm, который открывает X11 сервер по умолчанию, то все будет в порядке. Тем не менее если вы запускаете их из MacOS X Terminal, в этом случае вам надо определить переменную окружения DISPLAY. Если DISPLAY не определена, то получите сообщение: “ cannot open X11 display ”.

Например для запуска теста boxstack вам надо ввести

```
cd ode/test
DISPLAY=:0.0 ./test_boxstack.exe
```

Вы можете определить переменные окружения в своем скрипте оболочки запуска(shell startup scripts) (например в ~/.bashrc если вы используете bash)

2.2. Использование ODE

Лучший способ понять как использовать ODE посмотреть программы в test/example. Имейте в виду следующее:

- Программам, использующим ODE, необходимо подключать только один заголовочный файл:

```
#include <ode/ode.h>
```

Предполагается что в `include/ode` находится директория с дистрибутивом ODE. Этот заголовочный файл указывает на другие директории ODE, поэтому вам надо указать путь вашему компилятору, напр. для linux

```
gcc -c -I /home/username/ode/include myprogram.cpp
```

- Когда ODE использует функцию `dWorldStep`, для хранения временных данных используется стек. Для больших систем может потребоваться стек в несколько мегабайт. Если в ваших экспериментах происходят ошибки связанные с нехваткой памяти или повреждением данных, особенно в Windows, попробуйте увеличить размер стека или использовать `dWorldQuickStep`.

3. Понятия

3.1. На заметку

[Здесь будет информация о динамике жесткого тела и симуляции, ее необходимо знать. А сейчас смотрите прекрасный обучающий материал с SIGGRAPH - http://www.cs.cmu.edu/~baraff/sigcourse/index.html].

3.2. Жесткие тела

Жесткие тела, с точки зрения симуляции, обладают различными свойствами. Некоторые из этих свойств меняются со временем:

- Вектор позиции (x,y,z), указывающий на точку расположения тела (body's point of reference). На текущий момент точка расположения должна совпадать с центром массы тела.
- Линейной скоростью точки расположения является вектор (vx,vy,vz).
- Ориентация тела представлена кватернионом (qs, qx, qy, qz) или матрицей вращения 3x3.
- Вектор угловой скорости (wx,wy,wz), описывает как ориентация меняется во времени.

Другие свойства тела обычно остаются постоянными:

- Масса тела.
- Позиция центра масс по отношению к точке расположения. В текущей реализации центр масс и точка расположения должны совпадать.
- Матрица инерции. Это матрица 3x3 которая описывает как распределяется масса тела относительно центра массы.

В принципе каждое тело имеет x-y-z локальную систему координат, которая перемещается и вращается вместе с телом как показано на рисунке 1.

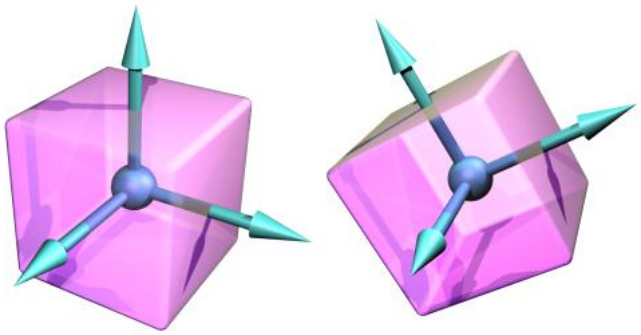

Рисунок 1: Локальная система координат тела.

Начало этой системы координат совпадает с точкой расположения тела. Некоторые значения в ODE (вектора, матрицы и т.д.) задаются в локальной системе координат тела, а другие в глобальной системе координат.

Учтите, что *форма* жесткого тела не является динамическим свойством. Только для *определения столкновений* имеет значение форма тела.

3.2.1. Острова и выключенные тела

Тела соединяются друг с другом с помощью сочленений(joints). “Остров”(island) тел представляет собой группу, которая не может быть разделена – другими словами каждое тело как то соединено с другим телом в острове.

Каждый остров в мире, во время шагов симуляции, обрабатывается отдельно. Вот что необходимо знать: если в симуляции принимают участие N схожих островов, то время расчета одного шага симуляции будет $O(N)$.

Каждое тело может быть включено(enabled) или выключено(disabled). Выключенные тела, во время шагов симуляции, “убираются” и не обрабатываются. Выключение тел это эффективный способ сократить время вычислений в тех случаях, когда точно известно что тела не подвижны или оказывают несущественное влияние на симуляцию.

Если в острове присутствует хоть одно включенное тело, то в этом случае на следующем шаге симуляции включается весь остров. Таким образом чтобы выключить весь остров, необходимо чтобы *все* тела острова были выключены. Если выключенный остров вступит во взаимодействие с включенным телом, то в этом случае весь остров будет включен, поскольку контактное сочленение(contact joint) включит какое то тело острова.

3.3. Интегрирование

Процесс симуляции системы жестких тел во времени называется интегрированием(integration). Каждый шаг интегрирования увеличивает текущее время на заданный шаг и устанавливает новые значения состояний для всех жестких тел. Существует две главные проблемы при интегрировании:

- Насколько оно точно? А именно, насколько точно поведение имитируемой системы совпадает с тем что происходит в реальной жизни?
- Насколько оно стабильно? А именно, как ошибки в вычислениях сказываются на физически некорректном поведении? (т.е. вынуждая системы “взрываться” без причины)

Текущая реализация интегрирования в ODE очень стабильна, но не очень точна даже несмотря на маленький шаг времени. Для большинства случаев это не является проблемой – поведение ODE почти во всех случаях смотрится прекрасно с точки зрения физики. Тем не менее не стоит использовать ODE для квантовой физики, до тех пор пока в следующих версиях проблемы с точностью не будут решены.

3.4. Аккумулятор силы

Между каждым шагом интегрирования пользователь может вызывать функции приложения сил к жесткому телу. Эти силы накапливаются в “аккумуляторе силы”(force accumulators) объекта жесткое тело. Когда происходит следующий шаг интегрирования, сумма всех приложенных сил вызывает перемещение тела. Аккумулятор силы устанавливается в ноль после каждого шага интегрирования.

3.5. Сочленения и соединения

В реальной жизни сочленение похоже на сгибание(hinge), используемое для соединения двух тел. Сочленение ODE имеет с ним много общего: это такое взаимоотношение между двумя телами, при котором тела могут занимать определенную позицию и ориентацию друг относительно друга. Такое взаимоотношение называется *соединением*(*constraint*) – слова сочленение и соединение часто взаимозаменяемы.

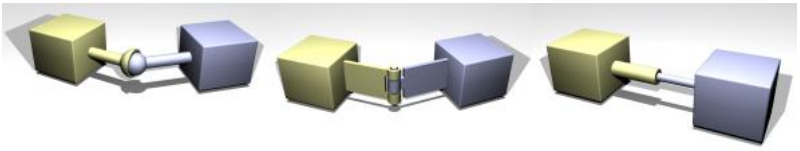

Рисунок 2: Три различных типа соединений.

Первое соединение называется сочленение шарик-в-разъеме(ball and socket joint), где “шарик” одного тела совпадает с расположением “разъема” другого тела. Второе это сочленение сгибание(hinge joint), здесь сгибание происходит в одном месте вдоль оси сгиба. Третье это сочленение скольжение(slider joint), здесь соединяются “поршень” и “разъем” вдоль одной линии, к тому же тела имеют одинаковую ориентацию.

На каждом шаге интегрирования всем сочленениям позволено применять *силы соединения*(*constraint forces*) к тем телам, к которым они присоединены. Эти силы вычисляются так, чтобы тела двигались, сохраняя связь с сочленением.

Каждое сочленение имеет многочисленные параметры, контролирующие геометрию. Например позицию точки шарика в разъема в сочленении шарик-в-разъеме. Функции, устанавливающие параметры сочленения, принимают *глобальные* координаты. В следствии этого каждое жесткое тело, к которому планируется присоединять сочленение, должно быть корректно размещено прежде чем присоединять сочленение.

3.6. Группы сочленений

Группа сочленений(joint group) это специальный контейнер, который содержит сочленения мира. Сочленения могут быть добавлены в группу, а когда сочленения группы больше не нужны, то вся группа может быть уничтожена с помощью одного вызова функции. Тем не менее отдельное сочленение группы не может быть уничтожено до тех пор пока группа не будет пуста.

Это особенно полезно для контактных сочленений, которые образовываются и удаляются из мира группами на каждом шаге времени.

3.7. Ошибки в сочленениях и параметр уменьшения ошибки (ERP)

Когда сочленение присоединяется к двум телам, необходимо чтобы тела занимали определенное положение и ориентацию друг относительно друга. Тем не менее возможно такое расположение тел, что сочленения не будет соединено. Такая “ошибка в сочленении”(joint error) может случиться в двух случаях:

1. Если пользователь установил позицию/ориентацию одного тела некорректно по отношению к позиции/ориентации другого тела.
2. Во время симуляции могут появляться ошибки, что приведет к смещению тел от необходимых позиций.

Рисунок 3 показывает пример ошибки в сочленении шарик-в-разъеме (здесь шарик и разъем не совпадают).

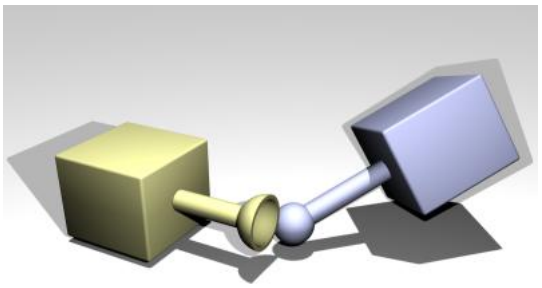

Рисунок 3: Пример ошибки в сочленении шарик-в-разъеме.

Механизм уменьшения ошибок в сочленении следующий: во время каждого шага симуляции каждому сочленению прилаживаются специальные силы, чтобы вернуть тела на правильные позиции. Эта сила контролируется *параметром уменьшения ошибки(error reduction parameter)* (ERP) и принимает значение от 0 до 1.

ERP определяет пропорцию, в которой ошибка в сочленении будет исправляться в следующем шаге симуляции. Если ERP=0, то корректирующая сила прилагаться не будет и тела будут перемещаться в соответствии с ходом симуляции. Если ERP=1, то будет предприниматься попытка исправить все ошибки в сочленениях на следующем шаге симуляции. Тем не менее устанавливать ERP=1 не рекомендуется поскольку ошибки в сочленении нельзя

полностью устранить из-за различных внутренних округлений. Рекомендуется устанавливать значения ERP=0.1 до 0.8 (0.2 значение по умолчанию).

Глобальное значение ERP воздействует на большинство сочленений симуляции. Тем не менее некоторые сочленения имеют локальные значения ERP, которые контролируют некоторые аспекты поведения сочленения.

3.8. Мягкое соединение и смешивающая сила соединения (CFM)

Большинство соединений по своей природе “твердые”(hard). Это значит что соединение находится в определенных условиях которые никогда не могут быть нарушены. Например, шарик всегда должен быть в разъеме, а сгибание должно происходит вдоль одной линии. На практике соединения могут быть нарушены непреднамеренным возникновением ошибок в системе, но с помощью параметра уменьшения ошибки можно откорректировать эти ошибки.

Не все соединения жестки. Некоторые “мягкие”(soft) соединения разработаны для того, чтобы быть нарушенными. Например, контактное соединение, которое предотвращает сталкивающиеся объекты от взаимного проникновения, по умолчанию жестки, это выглядит так как будто сталкивающиеся поверхности сделаны из стали. Но для симуляции материалов помягче можно сделать мягкое соединение, тем самым позволив естественному проникновению иметь место при взаимодействии двух объектов.

Существует два параметра которые контролируют различие между жестким и мягким соединением. Первый это параметр уменьшения ошибки (ERP), который уже был представлен. Второй это смешивающая сила соединения(constraint force mixing) (CFM), которая будет описана ниже.

3.8.1. Смешивающая сила соединения (CFM)

Далее следует описание значения CFM с технической точки зрения. Если вы просто хотите знать как использовать его на практике, то можете переходить к следующему разделу.

Обычно уравнение соединения(constraint equation) для каждого сочленения имеет форму

$$J * v = c$$

где v вектор скорости тел, J “Jacobian” матрица в которой одна строка представляет одну степень свободы сочленения и c правосторонний вектор. На следующем шаге времени вектор *lambda* вычисляется так чтобы силы приложенные к телам сохранили соединение сочленения

$$force = J^T * lambda$$

ODE использует следующий трюк. Уравнение соединения ODE имеет следующую форму

$$J * v = c + CFM * lambda$$

где *CFM* квадратно диагональная матрица(square diagonal matrix). *CFM* смешивает результирующую силу соединения в том соединении в котором она возникла. Ненулевое (положительное) значение *CFM* позволяет оригинальному уравнению соединения быть нарушенным пропорционально CFM раз, восстанавливая силу *lambda*, которая необходима для соединения. Решение для *lambda* дает

$$(J M^{-1} J^T + CFM/h)\ lambda = c/h$$

Таким образом *CFM* просто добавляет диагональ к оригинальной системной матрице. Дополнительное преимущество от использования положительного значения *CFM* заключается в уменьшении странностей системы и увеличение точности разложения на множители.

3.8.2. Как использовать ERP и CFM

Для многих сочленений ERP и CFM можно устанавливать независимо. Они могут быть установлены для контактных сочленений, ограничений сочленений(joint limits) и еще в других местах, контролируя мягкость(spongyness) и упругость(springyness) сочленения (или ограничения сочленения).

Если CFM установлен в ноль, соединение будет жестким. Если в CFM установлено положительное число, то появляется возможность нарушать соединение “смещением” (например в контактных соединениях сдерживать два контактирующих объекта вместе). Другими словами соединение становится мягким и мягкость будет нарастать с увеличением CFM. Происходит здесь следующие, соединению позволяется быть нарушенным пропорционально CFM раз, восстанавливая силу, которая нужна для удержания соединения. Учтите что установка в CFM отрицательного значения приведет к непредсказуемым отрицательным последствиям, например к нестабильности. Не делайте этого.

Устанавливая значения в ERP и CFM, вы можете добиваться различных эффектов. Например вы можете сымитировать упругий контакт, когда два тела будут колебаться как будто соединены пружиной. Или вы можете сымитировать более мягкий контакт, без колебаний. Вообще настраивая ERP и CFM можно достичь любого желаемого эффекта пружинистости или торможения соединения. Если у вас есть коэффициент жесткости пружины(spring constant) k_p и коэффициент торможения(damping constant) k_d, то константы ODE вычисляются так:

$$ERP = h\ k_p\ /\ (h\ k_p + k_d)$$
$$CFM = 1\ /\ (h\ k_p + k_d)$$

где h размер шага(stepsize). Эти значения дадут тот же эффект что и пружинно-тормозящая система(spring-and-damper system) сымитированная с помощью неявного интегрирования первого порядка.

Увеличение CFM, особенно глобальной CFM, может сократить многочисленные ошибки симуляции. Если система ведет себя странно, это может значительно увеличить стабильность. Вообще если система ведет себя неправильно первое что необходимо сделать это увеличить глобальную CFM.

3.9. Обработка столкновений

[Здесь надо много чего написать об обработке столкновений.]

Столкновения между телами, или между телом и статическим окружением, обрабатываются следующим образом:

1. Перед каждым шагом симуляции пользователь вызывает функции определения столкновений, для того чтобы определить что до чего дотронулось. Эти функции возвращают список точек контакта(contact points). Каждая точка контакта определена позицией в пространстве, вектором нормали к поверхности и глубиной проникновения.

2. Для каждой точки контакта создается специальное контактное сочленение(contact joint). Контактное сочленение дает дополнительную информацию о контакте, например о трении контактирующих поверхностей, пружинистости или мягкости и различных других свойствах.
3. Контактные сочленения помещаются в "группу" сочленений, группа позволяет быстро добавлять и удалять сочленения из системы. Скорость

симуляции падает с ростом числа контактов, поэтому существуют различные способы для сокращения количества точек контакта.

4. Происходит шаг симуляции.

5. Все контактные сочленения удаляются из системы.

Учтите что не обязательно использовать встроенную систему определения столкновений - другие библиотеки определения столкновений так же могут быть использованы, при условии что они предоставляют правильную информацию о контакте.

3.10. Типичный код симуляции

Типичный ход симуляции выглядит так:

1. Создать динамический мир.
2. Создать тела в динамическом мире.
3. Установить состояния(позиции и т.д.) для всех тел.
4. Создать сочленения в динамическом мире.
5. Соединить тела с сочленениями.
6. Установить параметры всем сочленениям.
7. Создать мир столкновений и геометрические объекты столкновений, если необходимо.
8. Создать группу сочленений для хранения контактных сочленений.
9. Цикл:
    1. Применить необходимые силы к телам.
    2. Установить необходимые параметры сочленений.
    3. Вызвать определение столкновений.
    4. Создать контактное сочленение для каждой точки столкновения и поместить сочленение в группу контактных сочленений.
    5. Произвести шаг симуляции.
    6. Удалить все сочленения из группы контактных сочленений.
10. Уничтожить динамический мир и мир столкновений.

3.11. Физическая модель

Здесь обсуждаются различные методы и упрощения используемые в ODE.

3.11.1. Упрощение трения

[Мне действительно необходимо здесь больше рисунков.]

Модель трения Coulomb проста, но эффективна для случая с точками контакта. Это просто отношение между нормалью и касательными силами присутствующими в точке контакта (смотри раздел о контактном сочленении для описания этих сил). Правило следующие:

$$|f_T| <= mu * |f_N|$$

Где f_N и f_T вектор силы нормали и касательный вектор силы соответственно, а *mu* коэффициент трения (обычно около 1.0). Это уравнение определяет “конус трения”(friction cone) – представьте конус у которого f_N является осью, точка контакта вершиной. Если суммарный вектор силы трения попадает в этот конус, то контакт находится в “режиме застревания”(sticking mode) и сила трения достаточна для предотвращения проскальзывания поверхностей друг относительно друга. Если вектор силы лежит на поверхности конуса, то контакт находится в “режиме скольжения”(sliding mode) и силы трения обычно не достаточно для того чтобы предотвратить проскальзывание контактирующих поверхностей. Таким образом параметр *mu* определяет максимальное соотношение касательной силы к силе нормали.

Из соображений эффективности модель трения ODE представлена конусом трения. На данный момент можно выбрать одно из двух упрощений:

1. Значение *mu* меняется таким образом что определяет максимальную(касательную) силу трения которая может быть представлена в контакте, направление трения совпадает с этой касательной. Это вообще-то физически не корректно, потому что нет зависимости от силы нормали, но может быть полезно и требует меньше вычислений. Учите что в этом случае *mu* является пределом силы(force limit) и для симуляции должно быть подобрано подходящее значение.
2. Конус трения упрощается до пирамиды трения(friction pyramid) в соответствии с первым и вторым направлениями трения [Мне действительно нужна здесь картинка]. Дальнейшее упрощение состоит в следующем: во первых ODE вычисляет силу нормали, полагая что трение во всех контактах отсутствует. Затем вычисляется максимальный предел f_m для (касательной) силы трения из

$$f_m = mu * |f_N|$$

и затем решается вся система с этими фиксированными ограничениями (схожим способом с пунктом 1 выше). Отличие от настоящей пирамиды трения состоит в том что значение *mu* не совсем фиксировано. Это упрощение проще использовать поскольку *mu* представляет собой безразмерную величину, такую как коэффициент трения Coloumb, и поэтому может быть установлен как константа в пределах 1.0 без учета специфики симуляции.

4. Типы данных и соглашения

4.1. Основные типы данных

Библиотека ODE может быть собрана в двух вариантах: с одинарной или двойной точностью внутренних вычислений. Одинарная точность быстрее и требует меньшего количества памяти, но в симуляции могут появляться многочисленные ошибки, что будет заметно визуально. Вы получаете меньшую точность и стабильность, используя одинарную точность.

[надо описать какие факторы влияют на точность и стабильность].

Форматом с плавающей запятой является `dReal`. Другими часто используемыми типами являются `dVector3`, `dVector4`, `dMatrix3`, `dMatrix4`, `dQuaternion`.

4.2. Объекты и их ID

Можно создавать различные виды объектов, а именно:

- dWorld - динамический мир.
- dSpace - пространство столкновений.
- dBody - жесткое тело.
- dGeom - геометрия (для столкновений).
- dJoint - сочленение
- dJointGroup - группа сочленений.

Функции, оперирующие этими объектами, принимают и возвращают их ID. ID типы объектов следующие `dWorldID, dBodyID`, и т.д.

4.3. Соглашения об аргументах

Все 3-х компонентные вектора (x,y,z) устанавливаются "set" функциями по индивидуальному компоненту.

Все 3-х компонентные вектора (x,y,z) принимаются с помощью get( ) функций, которые ссылаются на массив `dReal` через указатель.

Вектора большего размера всегда устанавливаются и принимаются с помощью указателей на массив `dReal`.

Все координаты задаются в глобальной системе координат, если не указанно обратного.

4.4. C против C++

Библиотека ODE написана на C++, но ее внешний интерфейс сделан простыми C функциями, а не классами. Почему так?

- Использовать C интерфейс проще - возможности C++ не очень помогают ODE.
- Предотвращаются искажения C++ и проблемы с многочисленными компиляторами.
- Пользователь не должен быть знаком с C++ для того чтобы начать использовать ODE.

4.5. Отладка

Библиотека ODE может быть скомпилирована в режиме "отладки" или "релиза". Режим отладки медленнее, но аргументы функций проверяются много раз что бы быть уверенным в их достоверности. Режим релиза быстрее, но таких проверок не производится.

5. Мир

Объект мир - это контейнер для жестких тел и сочленений. Объекты из различных миров не могут взаимодействовать, т.е. жесткие тела одного мира не могут столкнуться с телами из другого мира.

Все тела в мире существуют в одном месте в одно время, поэтому одна из причин использовать различные миры это симуляция систем с различной точностью.

Большинству приложений необходим только один мир.

```
dWorldID dWorldCreate();
```

Создает новый пустой мир и возвращает его ID номер.

```
void dWorldDestroy (dWorldID);
```

Уничтожает мир и все что в нем находится. Сюда входят все тела, все сочленения которые не являются частью группы сочленений. Сочленения которые входят в группу сочленений будут деактивированы и потом могут быть уничтожены вызовом например `dJointGroupEmpty( )`.

```
void dWorldSetGravity (dWorldID, dReal x, dReal y, dReal z);
void dWorldGetGravity (dWorldID, dVector3 gravity);
```

Устанавливает и получает вектор гравитации мира. Единица измерения м/с/с, поэтому вектор гравитации Земли будет (0,0,-9.81), пологая что ось +z направлена вверх. По умолчанию гравитации нет, т.е. (0,0,0).

```
void  dWorldSetERP (dWorldID, dReal erp);
dReal dWorldGetERP (dWorldID);
```

Устанавливает и получает глобальное значение ERP, которое контролирует как будет производиться корректировка ошибок на каждом шаге времени(time step). Обычно значение лежит в пределах 0.1--0.8. По умолчанию 0.2.

```
void  dWorldSetCFM (dWorldID, dReal cfm);
dReal dWorldGetCFM (dWorldID);
```

Устанавливает и получает глобальное значение CFM(constraint force mixing). Обычно значения лежат в пределах 10^{-9} -- 1. По умолчанию 10^{-5} если используется одинарная точность или 10^{-10} если используется двойная точность.

```
void  dWorldSetAutoDisableFlag (dWorldID, int do_auto_disable);
int   dWorldGetAutoDisableFlag (dWorldID);
void  dWorldSetAutoDisableLinearThreshold (dWorldID, dReal linear_threshold);
dReal dWorldGetAutoDisableLinearThreshold (dWorldID);
void  dWorldSetAutoDisableAngularThreshold (dWorldID, dReal angular_threshold);
dReal dWorldGetAutoDisableAngularThreshold (dWorldID);
void  dWorldSetAutoDisableSteps (dWorldID, int steps);
int   dWorldGetAutoDisableSteps (dWorldID);
void  dWorldSetAutoDisableTime (dWorldID, dReal time);
dReal dWorldGetAutoDisableTime (dWorldID);
```

Устанавливают и получают параметры авто-выключения(auto-disable) для только что созданных тел. Смотреть раздел 6.5 для описания возможностей авто-выключения. Параметры по умолчанию:

- AutoDisableFlag=disabled
- AutoDisableLinearThreshold=0.01
- AutoDisableAngularThreshold=0.01
- AutoDisableSteps=10
- AutoDisableTime=0

```
void dWorldImpulseToForce (dWorldID, dReal stepsize,
                           dReal ix, dReal iy, dReal iz, dVector3 force);
```

Если вы хотите приложить линейный или угловой импульс к жесткому телу вместо силы или вращающего момента, тогда вы можете использовать эту функцию для преобразования желаемого импульса в силу/вращающий момент, а потом вызвать функцию `dBodyAdd...`

Эта функция помещает вектор силы в `force` преобразованный из импульса (`ix`,`iy`,`iz`). Текущий алгоритм просто масштабирует импульс на1/`stepsize`, где `stepsize` это размер *следующего* шага.

Эта функция принимает `dWorldID` потому что в будущем расчет силы будет зависеть от параметров интегрирования, установленных в данном мире.

```
void dCloseODE();
```

Эта функция освобождает дополнительную память которую использует ODE и которая не может быть освобождена другими функциями, например `dWorldDestroy()`. Вы можете использовать эту функцию при завершении вашего приложения для предотвращения утечки памяти.

5.1. Функции шага

```
void dWorldStep (dWorldID, dReal stepsize);
```

Шаг мира. Используется метод "большая матрица"(big matrix) который требует времени расчета порядка m^3 и памяти порядка m^2, где m общее количество строк матрицы.

Для больших систем требуется много памяти и скорость работы будет низка, но на текущий момент это самый точный метод.

```
void dWorldQuickStep (dWorldID, dReal stepsize);
```

Шаг мира. Используется метод итераций, который требует времени порядка $m*N$ и памяти порядка m, где m общее количество строк и N количество итераций.

Для больших систем это работает намного быстрее чем `dWorldStep()`, но менее точно.

QuickStep отлично подходит для кучи объектов особенно вместе с auto-disable возможностями. Тем не менее, имеет не очень хорошую точность для неустойчивых систем(near-singular). Система может становиться неустойчивой, когда используются контакты с сильным трением, двигатели и определенные составные структуры. Например, робот с несколькими ногами, сидящий на земле, может быть неустойчивым.

Вот несколько советов предотвращения проблем неточности QuickStep'a:

- Увеличить CFM.
- Уменьшить количество контактов в вашей системе(т.е. минимизировать количество контактов для ступней робота или создания).
- Не использовать чрезмерное трение в контактах.
- Если можно, то использовать скользящие контакты.
- Избегать кинетических петель(kinematic loops)(тем не менее, кинетических петель не избежать в созданиях с ногами).
- Не использовать чрезмерную силу двигателя.
- Использовать двигатель который сообщает силу а не вращение.

Увеличение итераций в QuickStep может немного помочь, но не сильно, если система неустойчива.

```
void dWorldSetQuickStepNumIterations (dWorldID, int num);
int  dWorldGetQuickStepNumIterations (dWorldID);
```

Устанавливает и получает количество итераций которые использует метод QuickStep на каждом шаге расчета. Значение по умолчанию 20 итераций.

5.2. Параметры контакта

```
void  dWorldSetContactMaxCorrectingVel (dWorldID, dReal vel);
dReal dWorldGetContactMaxCorrectingVel (dWorldID);
```

Устанавливает и получает максимальное значение корректирующей скорости, которая может происходить в контактах. Значение по умолчанию бесконечность(т.е. неограниченная). Уменьшение этого значения может помочь предотвратить "проваливание"(popping) глубоко врезавшихся объектов.

```
void dWorldSetContactSurfaceLayer (dWorldID, dReal depth);
dReal dWorldGetContactSurfaceLayer (dWorldID);
```

Устанавливает и получает глубину слоя поверхности вокруг геометрических объектов. Контакты могут погружаться на глубину слоя поверхности прежде чем остановиться. Значение по умолчанию ноль. Увеличение этого значения на небольшую величину(напр. 0.001) может помочь решить проблему с дрожанием(jittering), когда контакт то появляется то исчезает.

6. Функции жестких тел

6.1. Создание и уничтожение тел

```
dBodyID dBodyCreate (dWorldID);
```

Создает тело в заданном мире с массой по умолчанию и в позиции (0,0,0). Возвращает его ID.

```
void dBodyDestroy (dBodyID);
```

Уничтожает тело. Все сочленения(joints), которые были присоединены к телу, переводятся в неопределенное состояние (т.е. отсоединяются и не принимают участия в дальнейшей симуляции, но НЕ удаляются).

6.2. Позиция и ориентация

```
void dBodySetPosition   (dBodyID, dReal x, dReal y, dReal z);
void dBodySetRotation   (dBodyID, const dMatrix3 R);
```

```
void dBodySetQuaternion (dBodyID, const dQuaternion q);
void dBodySetLinearVel  (dBodyID, dReal x, dReal y, dReal z);
void dBodySetAngularVel (dBodyID, dReal x, dReal y, dReal z);
const dReal * dBodyGetPosition    (dBodyID);
const dReal * dBodyGetRotation    (dBodyID);
const dReal * dBodyGetQuaternion (dBodyID);
const dReal * dBodyGetLinearVel   (dBodyID);
const dReal * dBodyGetAngularVel (dBodyID);
```

Эти функции устанавливают и получают позицию, вращение, линейную и угловую скорости тел. После установки группы тел результат симуляции не определен пока новая конфигурация не сопоставлена с уже имеющимися сочленениями/соединениями. Возвращаемые значения являются указателями на внутренние структуры данных, поэтому данные имеют смысл пока в системе не произошло изменений.

Hmmm. `dBodyGetRotation` возвращает матрицу вращения 4x3.

6.3. Масса и сила

```
void dBodySetMass (dBodyID, const dMass *mass);
void dBodyGetMass (dBodyID, dMass *mass);
```

Устанавливают и получают массу тела (смотреть функции массы тела).

```
void dBodyAddForce            (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddTorque           (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddRelForce         (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddRelTorque        (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddForceAtPos       (dBodyID, dReal fx, dReal fy, dReal fz,
                                        dReal px, dReal py, dReal pz);
void dBodyAddForceAtRelPos    (dBodyID, dReal fx, dReal fy, dReal fz,
                                        dReal px, dReal py, dReal pz);
void dBodyAddRelForceAtPos    (dBodyID, dReal fx, dReal fy, dReal fz,
                                        dReal px, dReal py, dReal pz);
void dBodyAddRelForceAtRelPos (dBodyID, dReal fx, dReal fy, dReal fz,
                                        dReal px, dReal py, dReal pz);
```

Прикладывают силы к телам (в абсолютных или относительных координатах). Каждое тело накапливает приложенные к нему силы, после каждого шага времени(time step) накопленные силы обнуляются.

Функции ...`RelForce` и ...`RelTorque` принимают в качестве параметров вектора в локальной системе координат этого тела.

Функции ...`ForceAtPos` и ...`ForceAtRelPos` принимают вектор с позицией точки приложения силы (в глобальной или локальной системе координат соответственно). Все другие функции прилагают силу к центру масс.

```
const dReal * dBodyGetForce  (dBodyID);
const dReal * dBodyGetTorque (dBodyID);
```

Возвращают текущий результат накопления в векторах силы/вращающего момента. Возвращаемые значения указывают на внутренние структуры данных, поэтому данные имеют смысл пока в системе жестких тел не произошло изменений.

```
void dBodySetForce  (dBodyID b, dReal x, dReal y, dReal z);
void dBodySetTorque (dBodyID b, dReal x, dReal y, dReal z);
```

Устанавливают вектора накопления силы/вращающего момента. Это полезно делать для обнуления силы и вращающего момента тел перед их деактивацией, в этом случае функции накопления силы будут вызываться, хотя тело деактивируется.

6.4. Утилиты

```
void dBodyGetRelPointPos (dBodyID, dReal px, dReal py, dReal pz,
                          dVector3 result);
void dBodyGetRelPointVel (dBodyID, dReal px, dReal py, dReal pz,
                          dVector3 result);
void dBodyGetPointVel    (dBodyID, dReal px, dReal py, dReal pz,
                          dVector3 result);
```

Утилитарные функции, которые принимают точку на теле (`px`,`py`,`pz`) и возвращают позицию точки или скорость в глобальных координатах (в `result`). Функции `dBodyGetRelPointXXX` принимают точку в системе координат тела, а `dBodyGetPointVel` в глобальной системе координат.

```
void dBodyGetPosRelPoint (dBodyID, dReal px, dReal py, dReal pz,
                                   dVector3 result);
```

Эта функция обратная `dBodyGetRelPointPos( )` . Принимает точку в глобальной системе координат (`x`,`y`,`z`) и возвращает в системе координат тела (`result`).

```
void dBodyVectorToWorld   (dBodyID, dReal px, dReal py, dReal pz,
                           dVector3 result);
void dBodyVectorFromWorld (dBodyID, dReal px, dReal py, dReal pz,
                           dVector3 result);
```

Вектор (`x`,`y`,`z`) в системе координат тела (или мира) преобразуется в мировую (или тела) систему координат (`result`).

6.5. Автоматическое включение и выключение

Каждое тело может быть включено(enabled) или выключено(disabled). Включенные тела участвуют в симуляции, в то время как выключенные тела деактивированы и их состояния не обновляются во время шагов симуляции(simulation step). Новые тела всегда создаются во включенном состоянии.

Отключенные тела, которые соединены сочленениями(joint) с включенными телами, автоматически включаются на следующем шаге симуляции.

Отключенные тела не тратят время CPU, поэтому неподвижные тела надо отключать. Это может быть сделано автоматически при помощи возможностей авто-выключения(auto-disable).

Если флаг авто-выключения установлен, то тела будут автоматически выключаться когда

1. Они будут бездействовать заданное количество шагов симуляции.
2. Они будут бездействовать заданное время симуляции.

Тело считается бездействующим если величина линейной и угловых скоростей меньше заданного порога.

Таким образом, каждое тело имеет пять параметров авто-выключения: флаг включения, количество шагов бездействия, время бездействия и пороги линейной/угловой скорости. Только что созданные тела берут эти параметры из мира.

Следующие функции устанавливают и получают параметры включения/выключения тела.

```
void dBodyEnable (dBodyID);
void dBodyDisable (dBodyID);
```

Ручное включение и выключение тела. Учтите что выключенное тело, соединенное сочленением с включенным телом, будет автоматически включено на следующем шаге симуляции.

```
int dBodyIsEnabled (dBodyID);
```

Возвращает 1 если тело включено и 0 если отключено.

```
void  dBodySetAutoDisableFlag (dBodyID, int do_auto_disable);
int   dBodyGetAutoDisableFlag (dBodyID);
```

Устанавливает и получает флаг авто-выключения тела. Если `do_auto_disable` имеет не нулевое значение, то тело будет автоматически выключаться при бездействии.

```
void  dBodySetAutoDisableLinearThreshold (dBodyID, dReal linear_threshold);
dReal dBodyGetAutoDisableLinearThreshold (dBodyID);
```

Устанавливает и получает порог линейной скорости для автоматического отключения. Величина линейной скорости тела должна быть ниже заданного порога, что бы тело считалось бездействующим. Установка порога в `dInfinity` исключает линейную скорость из рассмотрения.

```
void  dBodySetAutoDisableAngularThreshold (dBodyID, dReal angular_threshold);
dReal dBodyGetAutoDisableAngularThreshold (dBodyID);
```

Устанавливает и получает порог угловой скорости для автоматического отключения. Величина угловой скорости тела должна быть ниже заданного порога, что бы тело считалось бездействующим. Установка порога в `dInfinity` исключает угловую скорость из рассмотрения.

```
void  dBodySetAutoDisableSteps (dBodyID, int steps);
int   dBodyGetAutoDisableSteps (dBodyID);
```

Устанавливает и получает количество шагов симуляции, в течение которых тело должно бездействовать, что бы быть автоматически отключенным. Установка этого значения в ноль исключает шаги из рассмотрения.

```
void  dBodySetAutoDisableTime (dBodyID, dReal time);
dReal dBodyGetAutoDisableTime (dBodyID);
```

Устанавливает и получает время симуляции в течении которого тело должно бездействовать что бы быть автоматически отключенным. Установка этого значения в ноль исключает время симуляции из рассмотрения.

```
void  dBodySetAutoDisableDefaults (dBodyID);
```

Параметры авто-выключения тела устанавливаются как параметры по умолчанию для мира.

6.6. Разнообразные функции тела

```
void  dBodySetData (dBodyID, void *data);
void *dBodyGetData (dBodyID);
```

Получает и устанавливает указатель на данные пользователя для заданного тела.

```
void dBodySetFiniteRotationMode (dBodyID, int mode);
```

Эта функция контролирует то каким образом будет меняться ориентация тела на каждом шаге симуляции. Аргумент `mode` может быть:

- 0: Для обновления ориентации используется "бесконечно малая величина"(infinitesimal). Быстрый расчет, но порой могут возникать неточности при расчете быстро вращающихся тел, особенно когда такие тела соединены сочленениями с другими телами. По умолчанию новым телам присваивается это значение.
- 1: Для обновления ориентации используется "конечная"(finite) величина. Более сложный расчет, но и более точное вращение на высоких скоростях. Учтите, что вращение на высокой скорости может вызывать много типов ошибок в симуляции и этот режим исправляет только одну из них.

```
int dBodyGetFiniteRotationMode (dBodyID);
```

Возвращает текущий режим обновления ориентации тела (0 или 1).

```
void dBodySetFiniteRotationAxis (dBodyID, dReal x, dReal y, dReal z);
```

Эта функция устанавливает для тела ось вращения. Эта ось имеет смысл только когда установлен "конечный" режим обновления ориентации тела (смотреть dBodySetFiniteRotationMode( )).

Если ось установлена в ноль (0,0,0), то выполняется полное конечное(finite) вращение тела.

Если ось не нулевая, то происходит частично конечное(finite) вращение вокруг направления указанной оси, и бесконечно малое(infinitesimal) вращение вокруг перпендикулярной оси.

Это может быть полезным для уменьшения влияния некоторых видов ошибок вызванных быстрым вращением тел. Например если колеса машины вращаются с большой скоростью, то можно вызвать эту функцию и в качестве аргумента передать ось сгибания(hinge) сочленения колеса.

```
void dBodyGetFiniteRotationAxis (dBodyID, dVector3 result);
```

Возвращает текущую ось вращения тела.

```
int dBodyGetNumJoints (dBodyID b);
```

Возвращает количество сочленений присоединенных к телу.

```
dJointID dBodyGetJoint (dBodyID, int index);
```

Возвращает сочленение присоединенное к телу по заданному index. Индексы должны быть в диапазоне от 0 до *n*-1, где *n* значение возвращенное dBodyGetNumJoints( ).

```
void dBodySetGravityMode (dBodyID b, int mode);
int  dBodyGetGravityMode (dBodyID b);
```

Устанавливает/получает влияние гравитации мира на тело. Если mode не равно нулю, то гравитация влияет на тело, иначе не влияет. Гравитация всегда влияет на новые тела.

7. Типы сочленений и функции сочленений

7.1. Создание и уничтожение сочленений

```
dJointID dJointCreateBall (dWorldID, dJointGroupID);
dJointID dJointCreateHinge (dWorldID, dJointGroupID);
dJointID dJointCreateSlider (dWorldID, dJointGroupID);
dJointID dJointCreateContact (dWorldID, dJointGroupID,
                              const dContact *);
dJointID dJointCreateUniversal (dWorldID, dJointGroupID);
dJointID dJointCreateHinge2 (dWorldID, dJointGroupID);
dJointID dJointCreateFixed (dWorldID, dJointGroupID);
dJointID dJointCreateAMotor (dWorldID, dJointGroupID);
```

Создают новые сочленения заданного типа. В начале сочленение находится в неопределенном состоянии (т.е. не влияет на симуляцию) потому что не присоединено ни к какому телу. В группе сочленений с ID равным 0 сочленение просто устанавливается. Если номер группы отличен от 0 то сочленение помещается в эту группу. Контактные сочленения должны быть инициализированы структурой dContact.

```
void dJointDestroy (dJointID);
```

Уничтожает сочленение, отсоединяя его от тел и убирая из мира. Тем не менее если сочленение является частью группы, то эта функция не будет иметь эффекта - сначала надо очистить или уничтожить группу.

```
dJointGroupID dJointGroupCreate (int max_size);
```

Создает новую группу. Аргумент max_size не используется и должен быть равным 0. Оставлен для совместимости с более ранними версиями.

```
void dJointGroupDestroy (dJointGroupID);
```

Уничтожает группу сочленений. Все сочленения в данной группе будут уничтожены.

```
void dJointGroupEmpty (dJointGroupID);
```

Очищает группу сочленений. Все сочленения в данной группе будут уничтожены, но сама группа останется.

7.2. Разнообразные функции сочленений

```
void dJointAttach (dJointID, dBodyID body1, dBodyID body2);
```

Присоединяет сочленение к телам. Если сочленение уже куда то присоединено, то сначала оно будет отсоединено от старых тел. Для того чтобы присоединить сочленение к одному телу надо установить body1 или body2 в ноль - ноль значит статическое окружение. Установка обоих этих параметров в ноль переведет сочленение в неопределенное состояние и оно не будет принимать участия в симуляции.

Чтобы работать, некоторые сочленения, такие как сгибание-2(hinge-2), должны быть присоединены к двум телам.

```
void dJointSetData (dJointID, void *data);
void *dJointGetData (dJointID);
```

Устанавливает и получает указатель на данные определенные пользователем.

```
int dJointGetType (dJointID);
```

Получает тип сочленения. Может быть возвращена одна из следующих констант:

dJointTypeBall	ball-and-socket сочленение.
dJointTypeHinge	hinge joint сочленение.
dJointTypeSlider	slider joint сочленение.
dJointTypeContact	contact joint сочленение.
dJointTypeUniversal	universal joint сочленение.
dJointTypeHinge2	hinge-2 сочленение.
dJointTypeFixed	fixed joint сочленение.
dJointTypeAMotor	angular motor сочленение.

```
dBodyID dJointGetBody (dJointID, int index);
```

Возвращает тела к которым присоединено данное сочленение. Если index равен 0, то будет возвращено "первое" тело, соответствующие аргументу body1 в функции dJointAttach( ). Если index равен 1, то будет возвращено "второе" тело, соответствующие аргументу body2 функции dJointAttach( ).

Если одно из возвращенных значений равно нулю, то тело присоединено к статическому окружению. Если оба значения равны нулю, то сочленение

находится в неопределенном состоянии и не участвует в симуляции.

```
void dJointSetFeedback (dJointID, dJointFeedback *);
dJointFeedback *dJointGetFeedback (dJointID);
```

Во время шагов симуляции вычисляются силы, которые прилагают сочленения. Эти силы сочленение прилагает к телу и пользователь обычно не может знать сколько силы прилагает сочленение.

Если эта информация необходима пользователю, то он может создать структуру `dJointFeedback` и передать указатель в функцию `dJointSetFeedback()`. Эта структура определена следующим образом:

```
typedef struct dJointFeedback {
  dVector3 f1;       // сила, которую сочленение прилаживает к телу 1
  dVector3 t1;       // вращающий момент, который сочленение прилаживает к телу 1
  dVector3 f2;       // сила, которую сочленение прилаживает к телу 2
  dVector3 t2;       // вращающий момент, который сочленение прилаживает к телу 2
} dJointFeedback;
```

Во время шагов симуляции любая структура обратной связи(feedback structure), которая закреплена за сочленением, будет заполнена информацией о силе и вращающем моменте. Функция `dJointGetFeedback()` возвращает текущий указатель на структуру обратной связи или 0 если структура не определена. Функции `dJointSetFeedback()` можно передать 0 для прекращения получения информации обратной связи.

Далее идут некоторые замечания по реализации API. Может показаться странным что пользователь сам должен заботиться о размещении этой структуры. Почему для каждого сочленения просто статически не расположить эти данные? Причина в том что не всем пользователям нужна информация обратной связи, а даже тем кому нужна, то не для всех сочленений. Было бы неразумной тратой памяти расположение такой информации статически, особенно с добавлением новой информации в эту структуру в будущем.

Почему ODE не располагает эту структуру в внутри библиотеки по запросу пользователя? Причина в том, что для контактных(contact) сочленений (которые создаются и уничтожаются на каждом шаге симуляции) может потребоваться много времени для размещения информации в памяти если произойдет запрос на информацию обратной связи. Позволив пользователю самому заниматься расположением этой информации в памяти, обеспечивается лучшая стратегия управления памятью, т.е. просто расположение происходит не в фиксированном массиве.

Можно было бы иметь в API функцию обратного вызова(callback) сочленение-сила(joint-force). Это бы конечно работало, но имеется несколько проблем. Во первых функции обратного вызова склонны усложнять API и иногда требуют от пользователя неочевидных действий чтобы получить данные в нужное место. Во вторых это вызовет изменения в самом ODE (с нехорошими последствиями) и для того чтобы этого избежать надо будет проводить дополнительные проверки - что только все еще больше усложнит.

```
int dAreConnected (dBodyID, dBodyID);
```

Утилитарная функция: возвращает 1 если два тела соединены сочленением, в противном случае 0.

```
int dAreConnectedExcluding (dBodyID, dBodyID, int joint_type);
```

Утилитарная функция: возвращает 1 если два тела соединены сочленением не `joint_type`, в противном случае 0. `joint_type` может быть одной из `dJointTypeXXX` констант. Эта функция полезна для решения когда стоит создавать контактное сочленение(contact joint): если тела уже соединены не контактными сочленениями то может и не надо добавлять контакт, тем не менее нет ничего страшного в добавлении контакта между телами, которые уже имеют контакты.

7.3. Функции установки параметров сочленения

7.3.1. Шарик-в-разъеме(ball and socket)

Сочленение шарик-в-разъеме показано на рисунке 4.

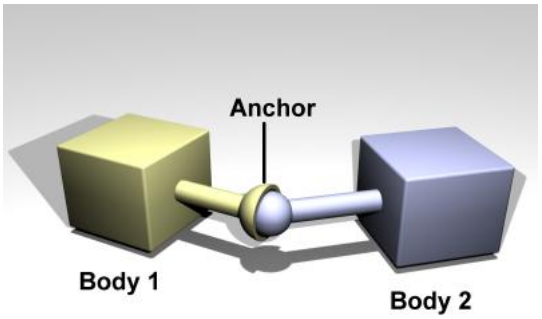

Рисунок 4: Сочленение шарик-в-разъеме.

```
void dJointSetBallAnchor (dJointID, dReal x, dReal y, dReal z);
```

Устанавливает точку соединения(anchor) сочленения. Сочленение будет пытаться удержать два тела относительно этой точки. Входное значение дается в мировых координатах.

```
void dJointGetBallAnchor (dJointID, dVector3 result);
```

Возвращает точку соединения сочленения в мировых координатах. Значение возвращается для тела 1. Если с сочленением все в порядке, то возвращенные значения для тел 1 и 2 будут совпадать.

```
void dJointGetBallAnchor2 (dJointID, dVector3 result);
```

Возвращает точку соединения сочленения в мировых координатах. Значение возвращается для тела 2. Вы можете думать о сочленении шарик-в-

разъеме как о попытке возврата функциями dJointGetBallAnchor() и dJointGetBallAnchor2() одинакового результата. Если с сочленением все в порядке функция вернет такое же значение как и dJointGetBallAnchor( ) без ошибок в округлении. Функция dJointGetBallAnchor2( ) может быть использована вместе с функцией dJointGetBallAnchor( ) для определения того насколько части сочленения отдалились друг от друга.

7.3.2. Сгибание(hinge)

Сочленение сгибание показано на рисунке 5.

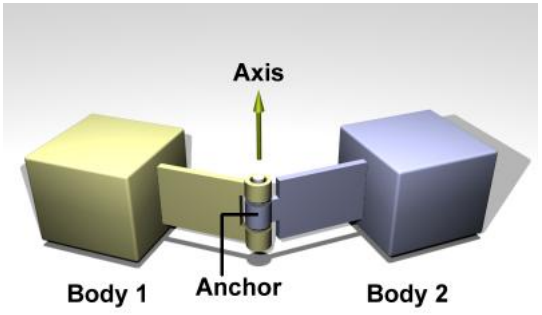

Рисунок 5: Сочленение сгибание.

```
void dJointSetHingeAnchor (dJointID, dReal x, dReal y, dReal z);
void dJointSetHingeAxis (dJointID, dReal x, dReal y, dReal z);
```

Устанавливает параметры: точку сгиба(hinge anchor) и ось сгиба(hinge axis).

```
void dJointGetHingeAnchor (dJointID, dVector3 result);
```

Получает точку сгиба сочленения в мировых координатах. Значение возвращается для тела 1. Если с сочленением все в порядке, то возвращенные значения для тел 1 и 2 будут совпадать.

```
void dJointGetHingeAnchor2 (dJointID, dVector3 result);
```

Получает точку сгиба сочленения в мировых координатах. Значение возвращается для тела 2. Если с сочленением все в порядке то возвращенное значен

```
void dJointGetHingeAxis (dJointID, dVector3 result);
```

Получает ось сгиба.

```
dReal dJointGetHingeAngle (dJointID);
dReal dJointGetHingeAngleRate (dJointID);
```

Получает угол сгиба, зависящий от времени. Угол измеряется между двумя телами или между телом и статическим окружением. Значение угла лежит между -pi..pi.

Когда точка сгиба или ось определена, угол между текущим положением прикрепленных тел считается нулевым.

7.3.3. Скольжение(slider)

Сочленение скольжение показано на рисунке 6.

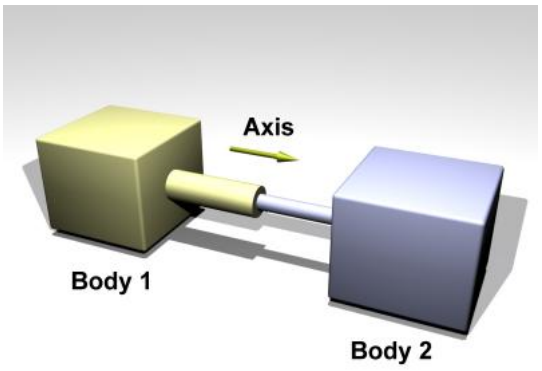

Рисунок 6: Сочленение скольжение.

```
void dJointSetSliderAxis (dJointID, dReal x, dReal y, dReal z);
```

Устанавливает ось скольжения(slider axis).

```
void dJointGetSliderAxis (dJointID, dVector3 result);
```

Получает ось скольжения.

```
dReal dJointGetSliderPosition (dJointID);
dReal dJointGetSliderPositionRate (dJointID);
```

Получает позицию скольжения, зависящую от времени.
Когда ось определена, позиция скольжения, присоединенных тел, считается равной нулю.

7.3.4. Универсал(universal)

Универсальное сочленение показано на рисунке 7.

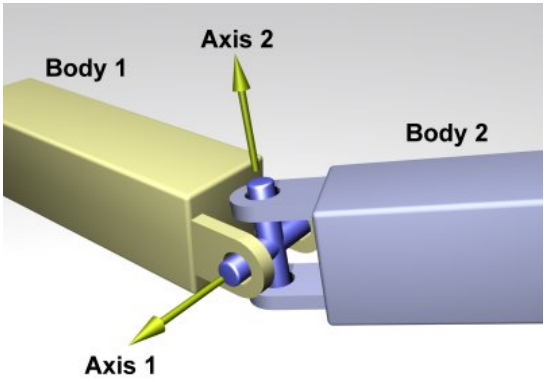

Рисунок 7: Универсальное сочленение.

Универсальное сочленение похоже на сочленение шарик-в-разъеме, у которого ограничена степень свободы вращения. Определив ось 1 для тела 1 и ось 2 для тела 2 перпендикулярно друг другу, их перпендикулярность будет сохраняться. Другими словами можно сказать, что тела будут сохранять перпендикулярную ориентацию относительно своих осей.

На картинке два тела соединены крест накрест. Ось 1 относится к телу 1 а ось 2 к телу 2. Угол между осями 90 градусов, поэтому если вы начнете вращать тело 1, то это вызовет вращение тела 2.

Универсальное сочленение эквивалентно сочленению сгибание-2(hinge-2), когда оси сгибания-2 перпендикулярны друг другу и жестко прикреплены к подвеске.

Универсальные сочленения проявляют себя в автомобилях, а именно при соединении двигателя с валом(shaft) вращающегося вдоль оси машины. Вы захотели изменить направление вала. Проблема в том, что если вы повернете вал, то другие части машины не будут повернуты согласно новому направлению. Поэтому, если вы вставите универсальное сочленение, то вы можете использовать сдерживающую силу чтобы заставить повернуться второй вал под тем же углом что и первый.

Другое использование этого сочленения в руках простых виртуальных созданий. Представьте что кто-то держит руки перед собой. Вы захотите чтобы руки могли двигаться вверх, вниз и вперед, назад, а не вращались вдоль своих осей.

Вот функции универсального сочленения:

```
void dJointSetUniversalAnchor (dJointID, dReal x, dReal y, dReal z);
void dJointSetUniversalAxis1 (dJointID, dReal x, dReal y, dReal z);
void dJointSetUniversalAxis2 (dJointID, dReal x, dReal y, dReal z);
```

Устанавливает универсальное соединение(universal anchor) и параметры осей. Оси 1 и 2 должны быть перпендикулярны друг другу.

```
void dJointGetUniversalAnchor (dJointID, dVector3 result);
```

Получает точку соединения сочленения в мировых координатах. Возвращается точка для тела 1. Если с сочленением все в порядке, то значение будет таким же как и для тела 2.

```
void dJointGetUniversalAnchor2 (dJointID, dVector3 result);
```

Получает точку соединения сочленения в мировых координатах. Возвращается точка для тела 2. Вы можете думать что, как и в сочленении шарик-в-разъеме, универсальное сочленение старается сохранить возвращаемые значения одинаковыми. Если с сочленением все в порядке, то функция вернет такое же значение как и dJointGetUniversalAnchor(). Функция dJointGetUniversalAnchor2( ) может быть использована для определения того насколько далеко разошлось сочленение.

```
void dJointGetUniversalAxis1 (dJointID, dVector3 result);
void dJointGetUniversalAxis2 (dJointID, dVector3 result);
```

Возвращает параметры универсальных осей.

7.3.5. Сгибание-2(hinge-2)

Сочленение сгибание-2 показано на рисунке 8.

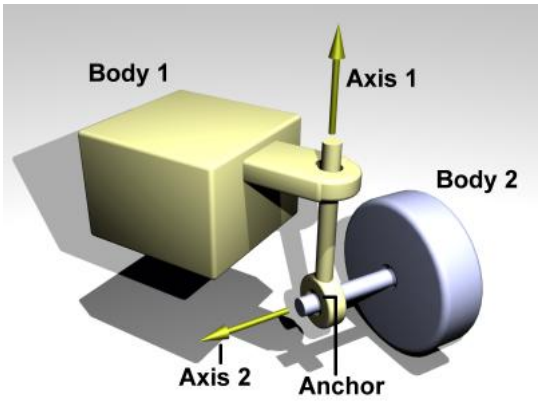

Рисунок 8: Сочленение сгибание-2.

Сочленение сгибание-2 похоже на два сочленения соединенных последовательно, с разными осями сгибания. В примере показанном на рисунке это поворачивающие колесо машины, где одна ось позволяет колесу поворачиваться а вторая вращаться.

Сочленение сгибание-2 имеет точку соединения(anchor point) и две оси сгибания. Ось 1 определяется относительно тела 1(может быть осью поворота, если тело 1 рама(chassis)). Ось 2 определяется относительно тела 2(может быть осью вращения колеса, если тело 2 колесо).

Ось 1 может иметь ограничения сочленения(joint limits) и двигатель (motor), ось 2 только двигатель.

Ось 1 может функционировать как ось подвески, т.е. происходить сжатие вдоль этой оси.

Сочленение сгибание-2 с осью 1 перпендикулярной оси 2 эквивалентно универсальному сочленению с добавлением подвески.

```
void dJointSetHinge2Anchor (dJointID, dReal x, dReal y, dReal z);
void dJointSetHinge2Axis1 (dJointID, dReal x, dReal y, dReal z);
void dJointSetHinge2Axis2 (dJointID, dReal x, dReal y, dReal z);
```

Устанавливает параметры осей и точки соединения. Оси 1 и 2 не должны лежать на одной прямой.

```
void dJointGetHinge2Anchor (dJointID, dVector3 result);
```

Получает точку соединения в мировых координатах. Значение возвращается для тела 1. Если с сочленением все в порядке, то значение будет такое же как и для тела 2.

```
void dJointGetHinge2Anchor2 (dJointID, dVector3 result);
```

Получает точку соединения в мировых координатах. Значение возвращается для тела 2. Если с сочленением все в порядке, возвращенное значение будет таким же как и dJointGetHinge2Anchor( ). Если нет, то значения будут немного отличаться. Это может быть использовано для определения того насколько далеко друг от друга отошли части сочленения.

```
void dJointGetHinge2Axis1 (dJointID, dVector3 result);
void dJointGetHinge2Axis2 (dJointID, dVector3 result);
```

Получает параметры осей.

```
dReal dJointGetHinge2Angle1 (dJointID);
dReal dJointGetHinge2Angle1Rate (dJointID);
dReal dJointGetHinge2Angle2Rate (dJointID);
```

Получает углы сгибание-2(вокруг осей 1 и 2) и это значение может со временем меняться.

Когда установлена точка соединения или оси, угол между телами относительно этой точки считается нулевым.

7.3.6. Фиксация(fixed)

Фиксированное сочленение(fixed joint) сохраняет фиксированную позицию и ориентацию тел, или тела и статического окружения. Использование такого сочленения не является хорошей идеей за исключением случаев отладки. Если вам надо использовать два тела склеенных вместе, то лучше представить их как одно тело.

```
void dJointSetFixed (dJointID);
```

Вызов этой функции фиксирует сочленение в текущем положении и ориентации друг относительно друга.

7.3.7. Контакт(contact)

Контактное сочленение(contact joint) показано на рисунке 9.

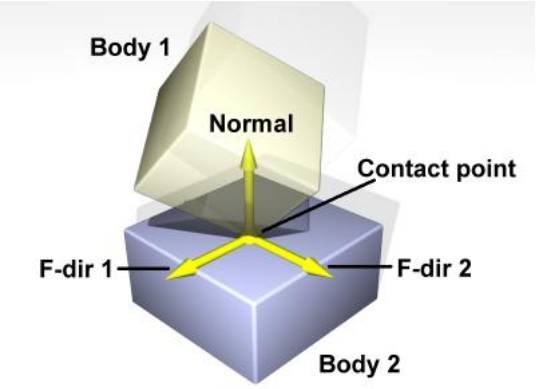

Рисунок 9: Контактное сочленение.

Контактное сочленение предотвращает тело 1 и 2 от взаимного проникновения в точке контакта. Оно только позволяет телам иметь "обратную"(outgoing) скорость в направлении нормали точки контакта. Время жизни контактного сочленения один шаг времени(one time step). Они создаются и уничтожаются в ответ на столкновения.

Контактное сочленение может имитировать трение в точке контакта, прилагая специальные силы в двух направлениях: перпендикулярных нормали.

Когда создается контактное сочленение, должна быть заполнена структура `dContact`. Она определена следующим образом:

```
struct dContact {
  dSurfaceParameters surface;
  dContactGeom geom;
  dVector3 fdir1;
};
```

`geom` это подструктура, которая устанавливается функцией определения столкновений. Она описана в разделе столкновений.

`fdir1` это вектор "первого направления трения" который определяет направление, вдоль которого должна быть приложена сила трения. Он должен быть в единицах масштаба(unit length) и перпендикулярен нормали контакта (т.е. быть касательной к поверхности контакта). Он должен быть определен только в том случае если флаг `dContactFDir1` в `surface.mode` установлен. Вектор "второго направления трения" вычисляется как перпендикуляр к нормали и `fdir1`.

`surface` это подструктура, которая настраивается пользователем. Ее члены определяют свойства взаимодействующих поверхностей. Структура имеет следующие члены:

- `int mode` - Флаги контакта. Всегда должен быть установлен. Это комбинация одного или более следующих флагов:

`dContactMu2`	Если не установлен, то используется `mu` для обоих направлений трения. Если установлен, то `mu` используется для первого направления трения, а `mu2` для направления трения 2.
`dContactFDir1`	Если установлен, то `fdir1` берется направлением трения 1, в противном случае `fdir1` рассчитывается автоматически как перпендикуляр к нормали контакта (в этом случае результирующие направление не предсказуемо).
`dContactBounce`	Если установлен, то поверхность контакта считается упругой, другими словами тела будут пружинить друг от друга. Точное значение упругости определяется `bounce` параметром.
`dContactSoftERP`	Если установлен, то параметр уменьшения ошибки нормали контакта контролируется через `soft_erp` параметр. Этот параметр нужен чтобы сделать поверхность мягче.
`dContactSoftCFM`	Если установлен, то смешивающая сила соединения нормали контакта контролируется через `soft_cfm` параметр. Этот параметр нужен чтобы сделать поверхность мягче.
`dContactMotion1`	Если установлен, то предполагается что поверхность контакта движется независимо от тел. Если этот флаг установлен, то `motion1` определяет скорость поверхности в направлении трения 1.
`dContactMotion2`	То же что и выше, но в направлении трения 2.
`dContactSlip1`	Скольжение – зависящие – от – силы (FDS) в направлении трения 1.
`dContactSlip2`	Скольжение – зависящие – от – силы (FDS) в направлении трения 2.
`dContactApprox1_1`	Использование пирамидного приближения трения(friction pyramid approximation) в направлении трения 1. Если этот флаг не установлен, то используется константное – ограничение – силы (constant-force-limit)(`mu` является ограничителем силы).
`dContactApprox1_2`	Использование пирамидного приближения трения(friction pyramid approximation) в направлении трения 2. Если этот флаг не установлен, то используется константное – ограничение – силы (constant-force-limit)(`mu` является ограничителем силы).
`dContactApprox1`	Эквивалентно установке `dContactApprox1_1` и `dContactApprox1_2`.

- `dReal mu`: коэффициент трения Coulomb. Значение должно лежать в диапазоне от 0 до `dInfinity`. 0 дает отсутствие трения в контакте, при `dInfinity` контакт никогда не будет скользить. Учтите, что отсутствие трения в контакте требует гораздо меньше времени на вычисления чем трение с конечным значением и бесконечное значение трения также проще в вычислениях чем трение с конечным значением. Это значение всегда должно быть определено.

- `dReal mu2`: Необязательный(optional) коэффициент трения Coulomb для направления трения 2 (0..`dInfinity`). Должен быть определен только в случае установки соответствующего флага в `mode`.

- `dReal bounce`: Параметр восстановления состояния(restitution parameter) (0..1). 0 значит что поверхность абсолютно не упруга, 1 максимальная упругость. Должен быть определен только в случае установки соответствующего флага в `mode`.

- `dReal bounce_vel`: Минимальная скорость необходимая для упругости (в м/с). Скорость ниже этого порога определяет параметр упругости равным 0. Должен быть определен только в случае установки соответствующего флага в `mode`.
- `dReal soft_erp`: Параметр “мягкости” нормали контакта. Должен быть определен только в случае установки соответствующего флага в `mode`.
- `dReal soft_cfm`: Параметр “мягкости” нормали контакта. Должен быть определен только в случае установки соответствующего флага в `mode`.
- `dReal motion1,motion2`: Скорость поверхности в направлении 1 и 2 (в м/с). Должен быть определен только в случае установки соответствующего флага в `mode`.
- `dReal slip1,slip2`: Коэффициенты скольжения–зависящего–от–силы (FDS) для направлений трения 1 и 2. Должен быть определен только в случае установки соответствующего флага в `mode`.

  FDS это эффект, который вынуждает контактирующие поверхности скользить относительно друг друга со скоростью пропорциональной силе приложенной по касательной к поверхности.

  Рассмотрим точку контакта в которой коэффициент трения *mu* бесконечность. Если сила *f* будет приложена к контактирующим поверхностям, пытаясь сдвинуть их друг относительно друга, они не будут двигаться. Тем не менее, если коэффициент FDS равен положительному *k*, тогда поверхности будут двигаться друг относительно друга с постоянной скоростью *k*f*.

  Учтите что это не нормальный эффект трения: сила не вызывает постоянного *ускорения* поверхностей друг относительно друга – вызывает кратковременное ускорение для достижения постоянной скорости.

  Это бывает полезно для моделирования некоторых ситуаций, в частности в шинах. Например, рассмотрим машину, стоящую на дороге. Толчок в направлении ее движения, вызовет движение машины (т.е. шины начнут вращаться). Толчок в перпендикулярном направлении ее движения не будет иметь никакого эффекта, поскольку шины не будут вращаться в этом направлении. Тем не менее – если машина движется со скоростью *v*, приложение силы *f* в перпендикулярном направлении вызовет скольжение шин по дороге со скоростью пропорциональной *f*v*. (Да, это действительно произойдет).

  Для моделирования этой ситуации в ODE необходимо установить параметры шина-дорога следующим образом: установить направление трения 1 так, что бы оно совпадало с направлением вращения шин и установить FDS коэффициент скольжения в направлении трения 2 равный *k*v*, где *v* скорость вращения шин, а *k* коэффициент шин подобранный экспериментальным путем.

  Учтите, что FDS не имеет отношения к эффекту трения Coulomb – оба этих режима могут быть использованы одновременно в одной точке контакта.

7.3.8. Угловой двигатель(angular motor)

Угловой двигатель (AMotor) позволяет контролировать относительную угловую скорость двух тел. Угловая скорость может быть контролируема по трем осям, позволяя вращать и останавливать вращение вокруг этих осей (Смотреть раздел ниже “Параметры движения и остановки ”). В основном это полезно в шариковом сочленении (которое имеет неограниченную степень свободы), когда надо ограничить степень свободы. Для использования AMotor в шариковом сочленении надо просто присоединить его к тем же телам что и шариковое сочленение.

AMotor может быть использован в различных режимах. В `dAMotorUser` режиме, пользователь определяет оси, контролируемые AMotor’ом. В `dAMotorEuler` режиме, AMotor вычисляет *euler углы* сообщающие относительное вращение, позволяя контролировать вращение и остановку. AMotor с euler углами показан на рисунке 10.

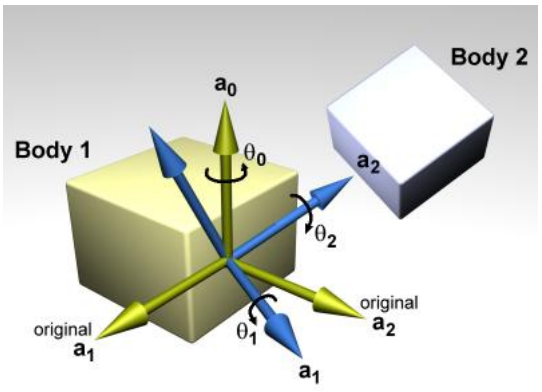

Рисунок 10: AMotor сочленение с euler углами.

На этой диаграмме a_0, a_1 и a_2 три оси вдоль которых контролируется угловое движение. Желтые оси (включая a_2) относятся к телу 2. Для получения осей тела 2 из осей тела 1 производится следующая последовательность вращений:

- *theta* $_0$ вращается вокруг a_0.
- *theta* $_1$ вращается вокруг a_1 (a_1 повернута относительно первоначальной позиции).
- *theta* $_2$ вращается вокруг a_2 (a_2 повернута дважды относительно первоначальной позиции).

Существует важное ограничение при использовании euler углов: угол *theta* $_1$ не должен выходить за пределы - *pi* /2 ... *pi* /2. Если это случится то AMotor станет не стабильным (эта особенность +/- *pi* /2). Таким образом, вы должны установить подходящие остановки(stops) на оси 1.

```
void dJointSetAMotorMode (dJointID, int mode);
int  dJointGetAMotorMode (dJointID);
```

Устанавливает (и получает) режим углового двигателя. Параметр `mode` может быть одной из следующих констант:

`dAMotorUser`	Оси AMotor и настройка углов сочленения(joint angle settings) полностью контролируются пользователем. Это режим по умолчанию.
`dAMotorEuler`	Euler углы автоматически рассчитываются. Ось a_1 так же автоматически рассчитывается. В этом режиме оси AMotor должны быть корректно установлены, как описано ниже. Когда инициализируется этот режим, текущая относительная ориентация тел сообщает всем

	euler углам 0.

```
void dJointSetAMotorNumAxes (dJointID, int num);
int  dJointGetAMotorNumAxes (dJointID);
```

Устанавливает (и получает) количество угловых осей(angular axes), контролируемых AMotor'ом. Аргумент может лежать в диапазоне от 0 (сочленение деактивировано) до 3-х. В режиме `dAMotorEuler` автоматически устанавливается 3.

```
void dJointSetAMotorAxis (dJointID, int anum, int rel,
                          dReal x, dReal y, dReal z);
void dJointGetAMotorAxis (dJointID, int anum, dVector3 result);
int  dJointGetAMotorAxisRel (dJointID, int anum);
```

Устанавливает (и получает) оси AMotor'а. Аргумент `anum` выбирает оси (0, 1 или 2). Каждая ось может иметь один из трех режимов "относительной ориентации", выбираемых `rel`:

- 0: ось прикрепляется к глобальной системе координат.
- 1: ось прикрепляется к первому телу.
- 2: ось прикрепляется ко второму телу.

Вектор оси (`x`,`y`,`z`) всегда определяется в глобальной системе координат независимо от `rel`. Две `GetAMotorAxis` функции, одна возвращает ось а вторая режим "относительной ориентации".

Для `dAMotorEuler` режима:

- Должны быть установлены только оси 0 и 2. Ось 1 будет определятся автоматически на каждом шаге времени.
- Оси 0 и 2 должны быть перпендикулярны друг другу.
- Ось 0 должна быть прикреплена к первому телу, а ось 2 ко второму телу.

```
void dJointSetAMotorAngle (dJointID, int anum, dReal angle);
```

Говорит AMotor'у какой текущий угол должен быть вдоль оси `anum`. Эта функция должна использоваться только в `dAMotorUser` режиме, потому что в этом режиме у AMotor'а нет другой возможности узнать углы сочленения. Информация об угле необходима, если для оси установлены остановы(stops) и не нужна, если установлены двигатели(motors).

```
dReal dJointGetAMotorAngle (dJointID, int anum);
```

Возвращает текущий угол для оси `anum`. В режиме `dAMotorUser` это значение просто соответствует значению установленному с помощью `dJointSetAMotorAngle`. В режиме `dAMotorEuler` возвращает соответствующий euler угол.

```
dReal dJointGetAMotorAngleRate (dJointID, int anum);
```

Возвращает текущий показатель угла(angle rate) для оси `anum`. В режиме `dAMotorUser` всегда ноль, поскольку не хватает информации. В режиме `dAMotorEuler` сообщает euler показатель угла.

7.4. Общие

Функции установки параметров геометрии сочленения должны вызываться после того как сочленение будет присоединено к телам, и эти тела корректно размещены, в противном случае сочленение может быть некорректно инициализировано. Если сочленение ни к чему не присоединено, то эти функции не дадут никакого эффекта.

Для функций возвращающих параметры, если система вышла из заданных параметров (т.е. присутствуют ошибки сочленения), возвращенные значения соединения/оси будут верны только относительно тела 1(или тела 2, если вы определили тело 1 как ноль в функции `dJointAttach`).

Точка соединения по умолчанию для всех сочленений (0,0,0). Ось по умолчанию для всех сочленений (1,0,0).

Когда ось установлена, она должна быть нормализована пропорционально телу. Скорректированная ось затем будет возвращаться функциям получения параметров(getting functions).

Когда измеряется угол или позиция, значение ноль соответствует начальной позиции тел друг относительно друга.

Учтите что нет функций для установки углов или позиций сочленений (или их показателей) напрямую, вместо этого вы должны сообщать позицию или скорость телам.

7.5. Параметры движения и остановки

Когда сочленение впервые создано, ничто не препятствует его движению в любом направлении. Например, сгибание может происходить на любой угол, а скольжение на любую длину.

Этот диапазон движений может быть ограничен установкой остановок(stops) в сочленении. Угол сочленения (или позиция) будет предохранен от достижения меньше заданного минимального значения или превышения высшее заданного верхнего значения. Учтите что угол сочленения (или позиция) считается нулевым в начальной позиции тел.

Так же как и остановки, многие типы сочленений могут иметь двигатели(motors). Двигатели прилаживают вращающий момент (или силу) к сочленениям для достижения желаемой скорости в точке вращения (или скольжения). Двигатели имеют ограничения силы, это значит, что они могут прилагать не более заданного значения силы/вращающего момента в сочленении.

Двигатели имеют два параметра: желаемая скорость и максимальная сила с помощью которой может быть достигнута эта скорость. Это очень упрощенная модель двигателя из реальной жизни. Тем не менее этого достаточно для моделирования двигателя, управляемого коробкой передач, подключаемого к сочленению. Такие устройства часто контролируют установку желаемой скорости и могут генерировать только максимальное количество мощности для достижения этой скорости (сообщая сочленению определенное количество силы).

Двигатели так же могут быть использованы для точного моделирования сухого (или Coulomb) трения в сочленениях. Просто установите желаемую скорость в ноль, а максимальное значение силы в некоторую константу – тогда все движения сочленения будут ограничены этой силой.

Альтернативное использование остановок и двигателей в сочленениях это простое приложение сил которые действуют на тела. Прилаживать силы двигателем просто, а остановки в сочленениях можно имитировать с помощью ограничения сил упругости. Тем не менее прилаживать силы напрямую

часто не очень хороший подход и может привести к некоторым проблемам стабильности если не быть осторожным.

Рассмотрим случай прилаживания силы к телу для достижения желаемой скорости. Для расчета этой силы вы используете информацию о текущей скорости, что то типа этого:

force = *k* * (*desired speed* - *current speed*)

Здесь имеется несколько проблем. Во-первых, параметр *k* должен быть настроен вручную. Если он будет слишком мал, то тело будет долго набирать требуемую скорость. Если он будет велик, то симуляция станет нестабильной. Во-вторых, даже если *k* выбран правильно, телу понадобится несколько шагов времени чтобы достигнуть желаемой скорости. В-третьих, если на тело подействует любая “внешняя” сила, желаемая скорость никогда не будет достигнута (требуется более сложное уравнение для расчета силы, с дополнительным набором параметров и со своими проблемами).

Двигатели сочленения(joint motors) решают все эти проблемы: они устанавливают скорость тела за один шаг времени, прилаживая столько силы сколько необходимо. Двигатели сочленения не нуждаются в дополнительных параметрах, поскольку они реализованы как соединения(constraints). С помощью них можно эффективно корректировать силу в последующих шагах времени. Это делает использование двигателей сочленения более сложным в расчетах чем просто приложение силы, но более точным и стабильным, и требует гораздо меньше времени на разработку. Это особенно актуально в больших системах жестких тел.

Схожие аргументы относятся и к остановкам сочленения(joint stops).

7.5.1. Функции установки параметров

Вот функции, которые устанавливают параметры двигателя(motor) и остановки(stop) (а так же другие параметры) в сочленении:

```
void dJointSetHingeParam (dJointID, int parameter, dReal value);
void dJointSetSliderParam (dJointID, int parameter, dReal value);
void dJointSetHinge2Param (dJointID, int parameter, dReal value);
void dJointSetUniversalParam (dJointID, int parameter, dReal value);
void dJointSetAMotorParam (dJointID, int parameter, dReal value);
dReal dJointGetHingeParam (dJointID, int parameter);
dReal dJointGetSliderParam (dJointID, int parameter);
dReal dJointGetHinge2Param (dJointID, int parameter);
dReal dJointGetUniversalParam (dJointID, int parameter);
dReal dJointGetAMotorParam (dJointID, int parameter);
```

Устанавливают/получают параметры ограничения/движения для каждого типа сочленения. Следующие параметры:

`dParamLoStop`	Нижний предел угла или позиции. Установка этого значения в `-dInfinity` (значение по умолчанию) выключает нижний предел. Для вращающихся сочленений, это значение должно быть больше чем - *pi* что бы был эффект.
`dParamHiStop`	Верхний предел угла или позиции. Установка этого значения в `dInfinity` (значение по умолчанию) выключает верхний предел. Для вращающихся сочленений, это значение должно быть меньше *pi* что бы был эффект. Если верхний предел меньше нижнего предела, то оба предела не дают никакого эффекта.
`dParamVel`	Желаемая скорость двигателя (здесь должна быть угловая или линейная скорость).
`dParamFMax`	Максимальная сила или вращающий момент, который двигатель будет использовать для достижения желаемой скорости. Это значение всегда должно быть больше или равно нулю. Установка этого значения в ноль (значение по умолчанию) выключает двигатель.
`dParamFudgeFactor`	В текущей реализации остановка/двигатель есть маленькая проблема: когда сочленение имеет остановку и двигатель, который пытается не допустить остановки, слишком большая сила может быть приложена за один шаг времени, вызывая “скачки” движения. Это надстроечный показатель(fudge factor) используется для масштабирования избыточной силы. Значение должно лежать между нулем и единицей (значение по умолчанию). Если в сочленении видны скачки движения, значение должно быть уменьшено. Делая это значение меньше можно уберечь двигатель от движения сочленения из положения остановки.
`dParamBounce`	Упругость остановок. Параметр восстановления лежит в диапазоне 0..1. 0 значит остановка совсем не упруга, 1 значит максимальная упругость.
`dParamCFM`	Значение смешивающей силы соединения (CFM) используется когда нет остановки.
`dParamStopERP`	Параметр уменьшения ошибки (ERP) используемый остановками.
`dParamStopCFM`	Значение смешивающей силы соединения (CFM), используемой остановками. Вместе со значением ERP может быть использовано для получения более мягких остановок. Учтите, предполагается что сочленения не достигают лимита силы, сочленения достигшие лимита силы будут работать некорректно.
`dParamSuspensionERP`	Параметр уменьшения ошибки в подвеске (ERP). На данный момент реализован только в сгибание-2(hinge-2) сочленении.
`dParamSuspensionCFM`	Значение смешивающей силы соединения (CFM) в подвеске. На данный момент реализована только в сгибание-2(hinge-2) сочленении.

Если параметры не реализованы в данном сочленении, то они не будут оказывать на него влияния.

Имена параметров могут опционально иметь цифры (2 или 3) для указания второго или третьего набора параметров, т.е. для второй оси в сочленении сгибание-2, или третьей оси в сочленении AMotor. Константна `dParamGroup` определяется как: `dParamX`*i* = `dParamX` + `dParamGroup` * (*i*-1)

7.6. Установка напрямую силы/вращающего момента сочленения

Двигатель(motor) позволяет устанавливать скорость сочленения напрямую. Тем не менее может понадобиться задать сочленению напрямую силу или вращающий момент. Следующие функции делают это. Учтите что они не выполняют функцию двигателей, они просто вызывают функции `dBodyAddForce/dBodyAddTorque` для тел к которым они присоединены.

```
dJointAddHingeTorque(dJointID joint, dReal torque)
```

Прилаживает вращающий момент вокруг оси сгиба. Здесь происходит приложение вращающего момента величиной `torque` в направлении оси сгиба тела 1 и такой же величины но в противоположном направлении для тела 2. Эта функция просто является оболочкой для `dBodyAddTorque`.

```
dJointAddUniversalTorques(dJointID joint, dReal torque1, dReal torque2)
```

Прилаживает `torque1` вокруг универсальной оси 1, и `torque2` вокруг универсальной оси 2. Эта функция просто является оболочкой для

dBodyAddTorque.

```
dJointAddSliderForce(dJointID joint, dReal force)
```

Прилаживает заданную силу в направлении скольжения. Здесь происходит приложение силы величиной `force` в направлении скольжения оси тела 1 и такой же величины но в противоположном направлении для тела 2. Эта функция просто является оболочкой для `dBodyAddForce`.

```
dJointAddHinge2Torques(dJointID joint, dReal torque1, dReal torque2)
```

Прилаживает `torque1` вокруг сгибание2(hinge2) оси 1, и `torque2` вокруг сгибание2(hinge2) оси 2. Эта функция просто является оболочкой для `dBodyAddTorque`.

```
dJointAddAMotorTorques(dJointID joint, dReal torque0, dReal torque1,
                       dReal torque2)
```

Прилаживает `torque0` вокруг AMotor оси 0, `torque1` вокругAMotor оси 1 и `torque2` вокруг AMotor оси 2. Если двигатель имеет меньше чем три оси, то старшие значения игнорируются. Эта функция просто является оболочкой для `dBodyAddTorque`.

8. StepFast

Замечание: алгоритм StepFast будет заменен QuickStep алгоритмом: смотреть функцию `dWorldQuickStep`. Тем не менее, большинство следующего описания относится к QuickStep, за исключением некоторых деталей.

Шаг системы(step the system) в функции ODE `dWorldStep` на текущий момент реализован с помощью метода “большая матрица”(big matrix). Для больших систем может потребоваться много времени для расчета и много памяти. Алгоритм StepFast1 обеспечивает альтернативный путь решения этой проблемы, принося в жертву точность. Его использование заключается в простой замене вызова `dWorldStep` на `dWorldStepFast1`.

Диаграмма на рисунке 11 показывает преимущества в скорости над стандартным `dWorldStep` алгоритмом.

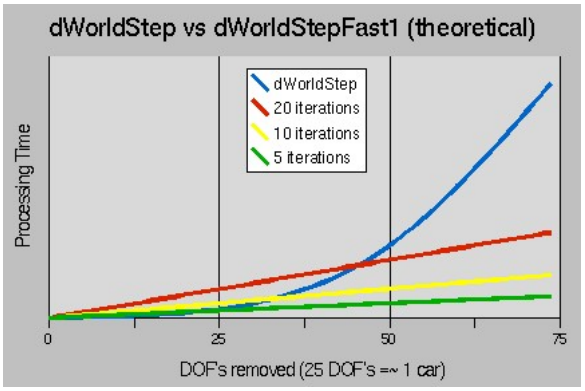

Рисунок 11: Преимущество в скорости StepFast.

График определяет количество Степеней Свободы(Degrees Of Freedom) (DOFs) в системе ко времени затраченном на расчет одного шага. Вы можете сказать, что время работы `dWorldStep` алгоритма пропорционально количеству DOFs в кубе. Тем не менее алгоритм StepFast1 строго линеен. Поэтому при увеличении размеров островов(islands) (например для кучи машин, свалки “тряпичных” тел, кирпичной стены) StepFast1 алгоритм масштабируется лучше чем `dWorldStep`. Все это значит что даже в худшем случае ваше приложение будет сохранять более стабильное количество кадров в секунду.

График расхода памяти ODE к количеству DOFs примерно выглядит так же(смотреть на рисунок 12).

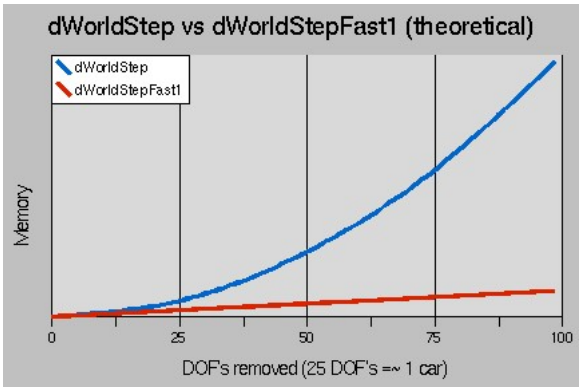

Рисунок 12: Преимущество использования памяти StepFast.

`dWorldStep` требует памяти пропорционально количеству DOFs. StepFast1 хотя и линеен, но не зависит от количества итераций в одном шаге. Поэтому страхи типа “Было большое количество объектов и ODE вылетела без сообщения об ошибке” (обычно из-за переполнения стека) не относятся к StepFast1. Просто кадр будет рассчитываться минуту или больше, но он будет рассчитан.

8.1. Когда использовать StepFast1

Как было показано высшее StepFast1 хороша когда дело касается скорости и использования памяти. Вся эта мощь не дается даром: выигрываем в одном, проигрываем в другом. Я уже упоминал что StepFast1 приносит в жертву точность в обмен на скорость и лучшее использование памяти. Вы вероятно захотите получить большую точность в обмен на скорость, увеличивая количество итераций в одном шаге. Хотя вы можете и не достигнуть точности `dWorldStep` (или даже превзойти ее), вы можете быть уверены что большее количество итераций дадут более точный результат (но более медленный). Поэтому StepFast1 можно использовать в большинстве ситуаций.

Основной ответ на этот вопрос таков: используйте StepFast1 когда у вас имеется немного параметров влияющий на стабильность, и вы хотите использовать преимущества скорости или расхода памяти. Если во время вашей симуляции большое количество объектов вступают в контакт, и скорость dWorldStep становится слишком медленной, попробуйте использовать StepFast1. Большинство систем работают прекрасно если заменить функцию dWorldStep на dWorldStepFast1. В некоторых случаях требуется небольшая настройка чтобы заработал StepFast1, в основном это касается масс тел. Когда сочленение присоединено к телам с большим соотношением масс (т.е. масса одного тела в несколько раз больше массы другого тела), StepFast1 может иметь проблемы с решением такой ситуации.

Другой совет использования StepFast1 это сразу ориентироваться на этот метод. Если вы знаете что будете строить большие миры с большим количеством физических объектов и планируете использовать StepFast1. Для решения проблемы, описанной высшее, вы надо привести все массы тел к 1.0. Или чтобы они варьировались в маленьком диапазоне, например от 0.5 до 1.5. Большинство предложений повышения стабильности и скорости также относятся и к StepFast1, за исключением того что надо использовать как можно меньше сочленений. Лучшим образом это проявляется в том что вы можете получить лучшую скорость и стабильность соединяя несколько тел фиксированными сочленениями(fixed joint) чем используя одно тело с большой массой, особенно если из-за этого массивного тела вам приходится переключаться на dWorldStep что бы сохранит стабильность.

Последний совет по использованию StepFast1 это использовать его только там где это необходимо. Поскольку StepFast1 использует такие же структуры тела и мира как и dWorldStep, то можно переключатся между ними. Хороший пример когда стоит переключения между ними это при подсчете контактных сочленений(contact joint), пока вы определяете столкновения. Поскольку определение столкновений определяется до выполнения шага симуляции, используя этот метод вы гарантируете что большой остров не затормозит систему как dWorldStep. Лучшим решением является использование гибридного решения, маленькие острова посылать в dWorldStep, а большие острова dWorldStepFast1. Возможно, это послужит источником изменений в библиотеке в будущем.

8.2. Когда НЕ стоит использовать StepFast1

Существует несколько специфических ситуаций, когда не стоит использовать StepFast1, я верю что они могут быть охарактеризованы одним предложением: Не используйте StepFast1 когда точность важнее скорости и количества используемой памяти. Вы вероятно захотите, в случае заметных неточностей, сделать большее количество итераций (20+).

8.3. Как это работает

За исключением нескольких интересных сторон здесь дано краткое описание того как работает StepFast1 алгоритм. Основная проблема состоит в том, что ODE пытается решить систему линейных уравнений с (m =количество соединений) неизвестными, где одно соединения является 1 степенью свободы сочленения. Для больших островов тел со многими сочленениями может потребоваться большой массив размером порядка $O(m^2)$, который требует время на решение порядка $O(m^3)$. StepFast1 избегает построения большой матрицы делая следующие допущение: уменьшая шаг времени(timestep) можно локализовать сочленение так чтобы друге сочленения не оказывали на него влияния, и любые конфликтующие сочленения не действовали друг на друга во время движения. Поэтому StepFast1 использует тот же метод решения (LCP) для решения той же проблемы с той лишь разницей что сочленение локализуется (m <= 6). Достигается это за счет дробления шага времени и повторения процесса с гораздо меньшим шагом (i =максимальное количество итераций). Поэтому время работы StepFast1 приблизительно равно $O(m\ i)$. Это действительно близко к $O(j\ i\ (m/j)^3) = O(m\ i\ (m/j)^2)$, где j =сочленений, но (m/j) никогда не >6, поэтому $(m/j)^2$ считается за константу.

8.4. Экспериментальные утилиты, входящие в состав StepFast1

Добавлено несколько экспериментальных функций как часть StepFast1. С помощью автоматического выключения и включения тел можно добиться дополнительной оптимизации. Вот основные идеи:

- Тело рассматривается как кандидат на выключение если его скорость (линейная или угловая) падает ниже определенного порога, с помощью функции AutoDisableThreshold. В интересах производительности текущая скорость тела считается равной скорости тела в квадрате. Поэтому может потребоваться установка меньшего значения чем ожидалось. 0.003 в test_crash работает нормально, и является значением по умолчанию.

- Тело, рассматриваемое как кандидат на выключение после определенного количества шагов (AutoDisableSteps), будет выключено. Хорошо подходит для прямоугольных параллелепипедов (boxs), которые после отскоков от земли пошатываются несколько раз и застывают неподвижно. Значение, необходимое для порога AutoDisableSteps, в основном должно быть меньше (например 1) чем в примере test_crash(10+), и является значением по умолчанию.

- АвтоВыключение(AutoDisabling) по умолчание выключено, используйте dBodySetAutoDisableSF1(body, true) для того чтобы включить его.

- Тело автоматически включается при вступлении в контакт с другим включенным телом.

- Включенные тела включают другие(AutoEnableDepth) тела на каждом шаге. Включение одного тела, на краю выключенных тел, вызывает включение остальных тел, область включения тел контролируется с помощью параметров АвтоВыключения. Установка ГлубиныАвтоВключения(AutoEnableDepth) в большое значение позволяет получить требуемую функциональность. Установка значения в 0 позволяет получить новую функциональность: выключенные тела никогда не будут включены, ведя себя просто как статическая геометрия. Значение 3 ведет себя хорошо в test_crash, но значение по умолчанию 1000 сохраняет стандартную функциональность.

Учтите что функции, имеющие отношение к авто-выключению, еще до конца не реализованы!

8.5. API

```
void dWorldStepFast1(dWorldID, dReal stepsize, int maxiterations);
```

Шаг мира в секундах, используемый в StepFast1 алгоритме, задается в stepsize. Количество итераций в maxiterations.

```
void dWorldSetAutoEnableDepthSF1(dWorldID, int autoEnableDepth);
int dWorldGetAutoEnableDepthSF1(dWorldID);
```

Устанавливает и получает параметр ГлубиныАвтоВключения(AutoEnableDepth), используемый в StepFast1 алгоритме.

```
void dBodySetAutoDisableThresholdSF1(dBodyID, dReal autoDisableThreshold);
dReal dBodyGetAutoDisableThresholdSF1(dBodyID);
```

Устанавливает и получает ПорогАвтоВыключния(AutoDisableThreshold) для данного тела, используемый в StepFast1 алгоритме.

```
void dBodySetAutoDisableStepsSF1(dBodyID, int AutoDisableSteps);
int dBodyGetAutoDisableStepsSF1(dBodyID);
```

Устанавливает и получает количество ШаговАвтоВыключения(AutoDisableSteps) для данного тела, используемых в StepFast1 алгоритме.

```
void dBodySetAutoDisableSF1(dBodyID, int doAutoDisable);
int dBodyGetAutoDisableSF1(dBodyID);
```

Устанавливает и получает флаг АвтоВыключения(AutoDisable) для данного тела, используемый в StepFast1 алгоритме. Если `doAutoDisable` не ноль, то авто-выключение включено. Если `doAutoDisable` равен нулю, то авто-выключение выключено.

9. Функции поддержки

9.1. Функции вращения

Ориентация жесткого тела представлена с помощью кватернионов. Кватернион состоит из четырех чисел [*cos*(*theta* /2),*sin*(*theta* /2)**u*] где *theta* угол вращения, а *u* единица масштаба(unit length) оси вращения.

Так же каждое жесткое тело имеет матрицу вращения, образованную от кватерниона. Матрица вращения и кватернион всегда совпадают.

Вот некоторая информация о кватернионах:

- q и –q определяют одинаковое вращение.
- Инверсный кватернион равен [*q*[0] -*q*[1] -*q*[2] -*q*[3]].

Следующие утилитарные функции работают с матрицами вращения и кватернионами.

```
void dRSetIdentity (dMatrix3 R);
```

Устанавливает `R` в единичную матрицу (т.е. вращения нет).

```
void dRFromAxisAndAngle (dMatrix3 R,
                         dReal ax, dReal ay, dReal az, dReal angle);
```

Вычисляет матрицу вращения `R` как вращение на угол `angle` в радианах вокруг оси (`ax`,`ay`,`az`).

```
void dRFromEulerAngles (dMatrix3 R,
                        dReal phi, dReal theta, dReal psi);
```

Вычисляет матрицу вращения `R` на основании трех углов вращения Euler.

```
void dRFrom2Axes (dMatrix3 R, dReal ax, dReal ay, dReal az,
                  dReal bx, dReal by, dReal bz);
```

Вычисляет матрицу вращения `R` на основании двух векторов ‘a’ (`ax`,`ay`,`az`) и ‘b’(`bx`,`by`,`bz`). ‘a’ и ‘b’ это желаемые оси x и y вращаемой системы координат. Если необходимо, ‘a’ и ‘b’ должны быть приведены к единицам масштаба, а ось ‘b’ перпендикулярна ‘a’. Ось ‘z’ будет векторным произведением ‘a’ и ‘b’.

```
void dQSetIdentity (dQuaternion q);
```

Устанавливает `q` в единичное вращение (т.е. вращения нет).

```
void dQFromAxisAndAngle (dQuaternion q, dReal ax, dReal ay, dReal az,
                         dReal angle);
```

Вычисляет `q` как вращение на угол `angle` в радианах вокруг оси (`ax`,`ay`,`az`).

```
void dQMultiply0 (dQuaternion qa,
                  const dQuaternion qb, const dQuaternion qc);
void dQMultiply1 (dQuaternion qa,
                  const dQuaternion qb, const dQuaternion qc);
void dQMultiply2 (dQuaternion qa,
                  const dQuaternion qb, const dQuaternion qc);
void dQMultiply3 (dQuaternion qa,
                  const dQuaternion qb, const dQuaternion qc);
```

Устанавливает `qa` = `qb`*`qc`. Это тоже самое что `qa` = вращение `qc` следует за вращением `qb`. Версии 0/1/2 аналогичны функциям умножения, т.е. 1 использует инверсную `qb`, а 2 использует инверсную `qc`. В 3 используется инверсия обоих.

```
void dQtoR (const dQuaternion q, dMatrix3 R);
```

Конвертирует кватернион `q` в матрицу вращения `R`.

```
void dRtoQ (const dMatrix3 R, dQuaternion q);
```

Конвертирует матрицу вращения `R` в кватернион `q`.

```
void dWtoDQ (const dVector3 w, const dQuaternion q, dVector4 dq);
```

Заданная ориентация в `q` и вектор угловой скорости `w`, возвращаются в `dq` как dq/dt.

9.2. Функции массы

Параметры массы жесткого тела описаны в `dMass` структуре:

```
typedef struct dMass {
  dReal mass;   // масса жесткого тела
  dVector4 c;   // позиция центра тяжести в системе координат тела (x,y,z)
  dMatrix3 I;   // 3x3 матрица инерции в системе координат тела, вокруг POR
} dMass;
```

Следующие функции оперируют этой структурой:

```
void dMassSetZero (dMass *);
```

Устанавливает все параметры в ноль.

```
void dMassSetParameters (dMass *, dReal themass,
                         dReal cgx, dReal cgy, dReal cgz,
                         dReal I11, dReal I22, dReal I33,
                         dReal I12, dReal I13, dReal I23);
```

Устанавливает массу в заданное значение. Масса тела `themass`. Центр тяжести в системе координат тела (`cx`,`cy`,`cz`). Значения `Ixx` элементы матрицы инерции.

```
    [ I11 I12 I13 ]
    [ I12 I22 I23 ]
    [ I13 I23 I33 ]
```

```
void dMassSetSphere (dMass *, dReal density, dReal radius);
void dMassSetSphereTotal (dMass *, dReal total_mass, dReal radius);
```

Устанавливает параметры массы сферы по заданному радиусу и плотности, с центром масс в точке (0,0,0) в системе координат тела. Первая функция устанавливает плотность, вторая массу сферы.

```
void dMassSetCappedCylinder (dMass *, dReal density, int direction,
                             dReal radius, dReal length);
void dMassSetCappedCylinderTotal (dMass *, dReal total_mass,
                                  int direction, dReal radius, dReal length);
```

Устанавливает параметры массы цилиндра с верхушкой по заданным параметрам и плотности, с центром масс в точке (0,0,0) в системе координат тела. Радиус цилиндра (и сферической верхушки) в `radius`. Длина цилиндра (не считая верхушки) в `length`. Ось цилиндра ориентирована вдоль оси тела x, y, или z в соответствии с `direction` (1=x, 2=y, 3=z). Первая функция устанавливает плотность объекта, вторая массу.

```
void dMassSetCylinder (dMass *, dReal density, int direction,
                       dReal radius, dReal length);
void dMassSetCylinderTotal (dMass *, dReal total_mass, int direction,
                            dReal radius, dReal length);
```

Устанавливает параметры массы цилиндра по заданным параметрам и плотности, с центром масс в точке (0,0,0) в системе координат тела. Радиус цилиндра задан через `radius`. Длина цилиндра `length`. Ось цилиндра ориентирована вдоль оси тела x, y, или z в соответствии с `direction` (1=x, 2=y, 3=z). Первая функция устанавливает плотность объекта, вторая массу.

```
void dMassSetBox (dMass *, dReal density,
                  dReal lx, dReal ly, dReal lz);
void dMassSetBoxTotal (dMass *, dReal total_mass,
                       dReal lx, dReal ly, dReal lz);
```

Устанавливает параметры массы прямоугольного параллелепипеда по заданной размерности и плотности, с центром масс в точке (0,0,0) в системе координат тела. Длина сторон прямоугольного параллелепипеда вдоль осей x, y и z равняется `lx`, `ly` и `lz`. Первая функция устанавливает плотность объекта, вторая массу.

```
void dMassAdjust (dMass *, dReal newmass);
```

Параметру массы некоторого объекта присваивается значение новой массы `newmass`. Эту функцию полезно использовать для установки определенной массы вместе с функцией установки плотности.

```
void dMassTranslate (dMass *, dReal x, dReal y, dReal z);
```

Параметр массы некоторого объекта смещается на величину (`x`,`y`,`z`) относительно системы координат тела.

```
void dMassRotate (dMass *, const dMatrix3 R);
```

Параметр массы некоторого объекта вращается на величину `R` относительно системы координат тела.

```
void dMassAdd (dMass *a, const dMass *b);
```

Сложение массы `b` к массе `a`.

9.3. Математические функции

[Функций достаточно, но они еще не достаточно стандартизированы для документирования].

9.4. Функции памяти и ошибок

[Будут документированы позже].

10. Определение столкновений

ODE состоит из двух основных компонентов: движка симуляции физики(dynamics simulation engine) и движка определения столкновений(collision detection engine). Движок столкновений дает информацию о *форме* каждого тела. На каждом шаге симуляции происходит определение какие тела коснулись друг друга и информация о точках контакта(contact points) посылается пользователю. Пользователь должен на основании этих данных создать контактные сочленения(contact joints) между телами.

Использование определений столкновений ODE опционально – можно использовать другую систему определения столкновений, которая будет сообщать правильную информацию о контактах.

10.1. Точки контакта

Если два тела коснулись друг друга или если тело коснулось статического окружения, контакт представляется одной или более “точками контакта”(contact points). Каждая точка контакта представлена соответствующей структурой `dContactGeom`:

```
struct dContactGeom {
  dVector3 pos;       // позиция контакта
  dVector3 normal;    // вектор нормали
  dReal depth;        // глубина проникновения
  dGeomID g1,g2;      // контактирующая геометрия
```

```
};
```

`pos` запись о позиции контакта, в глобальных координатах.

`depth` глубина, на которую тела проникли друг в друга. Если глубина равна нулю, то это значит легкое касание тел, т.е. что тела “только что” коснулись. Тем не менее это редкость – симуляция будет не достаточно точной и часто нужен еще один шаг чтобы глубина стала не нулевой.

`normal` это вектор в единицах масштаба(unit length vector), который обычно является перпендикуляром к поверхности контакта.

`g1` и `g2` это объекты геометрии, которые столкнулись.

Принято соглашение, что если тело 1 продвинулось вдоль `normal` вектора на расстояние `depth` (или тело 2 продвинулось на такое же расстояние, но в противоположном направлении), тогда глубина контакта уменьшается до нуля. Это значит что вектор нормали указывает “в” тело 1.

В реальной жизни контакт между телами сложная вещь. Представление контакта точками контакта всего лишь приближение. “Участок” или “поверхность” контакта могла бы быть более точна с физической точки зрения, но это бы незамедлительно сказалось на скорости симуляции.

Каждая дополнительная точка контакта будет замедлять симуляцию и иногда полезно игнорировать некоторые точки в интересах скорости. Например, при столкновении двух прямоугольных параллелепипедов(box) образуется множество точек контакта, но мы можем выбрать только три лучшие из них. Таким образом мы приближаем приближение.

10.2. Геометрия

Геометрические объекты (или геометрия для краткости) это фундаментальные объекты в системе столкновений. Геометрия может быть представлена одной жесткой формой (такой как сфера или прямоугольный параллелепипед) или может состоять из группы геометрий – этот особый вид геометрии который называется “пространство”(space).

Любая геометрия может столкнутся с другой геометрией и образовать ноль или более точек контакта. Пространства имеют дополнительные возможности по определению столкновений входящих в них геометрий, образовывая внутренние точки контакта.

Геометрия может быть перемещаемая или не-перемещаемая. Перемещаемая геометрия имеет вектор позиции и матрицу вращения 3x3, как жесткое тело, которое может менять положение в течение симуляции. Не-перемещаемая геометрия не имеет таких возможностей – например, она может быть представлена статическим окружением, которое не может быть перемещено. Пространства это не-перемещаемая геометрия, потому что каждая геометрия, входящая в пространство, имеет свою позицию и ориентацию, а иметь позицию и ориентацию для пространства не имеет смысла.

При использовании движка столкновений в симуляции поведения жестких тел, перемещаемая геометрия ассоциируется с объектами жесткие тела. Это позволяет движку столкновений получать позицию и ориентацию геометрии от тел. Учтите что геометрия это не тоже самое что жесткое тело в том плане что геометрия имеет геометрические свойства (размер, форму, позицию и ориентацию) но не физические свойства (такие как скорость или масса). Тело и геометрия вместе представляют все свойства имитируемого объекта.

Каждая геометрия относится к какому либо *классу*, такому как сфера, плоскость или прямоугольный параллелепипед. Существует определенный набор классов, описанных ниже, и вы также можете определять свои классы.

Точкой расположения(point of reference) перемещаемой геометрии является точка, которая контролируется вектором позиции. Точкой расположения для стандартных классов геометрии обычно является центр масс. Эта возможность позволяет стандартным классам легко быть присоединенным к физическим телам. Если требуются другие точки расположения, преобразованный объект может использовать изолированную геометрию.

Понятия и функции, которые относятся к геометрии, будут описаны ниже, далее пойдут различные классы геометрии и функции, которые ими управляют.

10.3. Пространства

Пространство(space) это не-перемещаемая геометрия, которая может содержать другую геометрию. Это схоже с понятием жестких тел “мир”, за исключением того, что речь идет о столкновениях а не о физике.

Объекты пространства существуют для того чтобы сделать симуляцию быстрее. Без пространств вам надо было бы получать контакты в симуляции, вызывая `dCollide` для получения точек контакта для каждой пары геометрии. Для N геометрий понадобилось бы $O(N^2)$ проверок, что представляет собой увеличение времени вычислений если ваше окружение содержит много геометрии.

Лучший выход это вставить геометрию в пространства и вызвать `dSpaceCollide`. Пространства произведут отсечение столкновении(collision culling), это значит быстрое определение того, какие пары геометрии *потенциально* пересекаются. Эти пары передадутся функцию обратного вызова(callback function), которая в свою очередь вызовет `dCollide`. Таким образом экономится много времени, которое могло бы быть потрачено на `dCollide` проверки, потому что будет передана только часть от всех возможных вариантов объект-объект пар.

Пространства могут включать в себя другие пространства. Это может быть полезно для деления сталкивающихся пространств на несколько уровней иерархии для дальнейшей оптимизации скорости определения столкновений. Более детальное описание будет дано ниже.

10.4. Основные функции геометрии

Следующие функции могут быть применены к любой геометрии.

```
void dGeomDestroy (dGeomID);
```

Уничтожает геометрию, удаляя ее сначала из любого пространства. Эта функция уничтожает геометрию любого типа, но при создании геометрии вы должны вызывать функцию создания для данного вида.

При уничтожении пространства, если режим очистки(cleanup mode) установлен в 1 (по умолчанию), вся геометрия в пространстве будет автоматически уничтожена.

```
void dGeomSetData (dGeomID, void *);
void *dGeomGetData (dGeomID);
```

Эти функции устанавливают и получают указатель на данные определенные пользователем для геометрии.

```
void dGeomSetBody (dGeomID, dBodyID);
dBodyID dGeomGetBody (dGeomID);
```

Эти функции устанавливают и получают тело, ассоциированное с перемещаемой геометрией. Установка ассоциации тела с геометрией объединяют вектор позиции и матрицу вращения тела и геометрии, таким образом установка позиции или ориентации влияют на оба объекта.

Установка ID тела в ноль дает геометрии собственную позицию и ориентацию, независимо от тела. Если геометрия ранее была ассоциирована с телом, то изменение положения/ориентации тела будет вызывать только изменение положения/ориентации этого тела.

Вызов этих функций для не-перемещаемой геометрии будет выдавать ошибку(runtime error) в отладочном режиме(debug build) ODE.

```
void dGeomSetPosition (dGeomID, dReal x, dReal y, dReal z);
void dGeomSetRotation (dGeomID, const dMatrix3 R);
void dGeomSetQuaternion (dGeomID, const dQuaternion);
```

Устанавливает вектор позиции, матрицу вращения или кватернион для перемещаемой геометрии. Эти функции аналогичны `dBodySetPosition`, `dBodySetRotation` и `dBodySetQuaternion`. Если геометрия ассоциирована с телом, то это вызовет изменение позиции/вращения/кватерниона тела.

Вызов этих функций для не-перемещаемой геометрии будет выдавать ошибку(runtime error) в отладочном режиме(debug build) ODE.

```
const dReal * dGeomGetPosition (dGeomID);
const dReal * dGeomGetRotation (dGeomID);
void dGeomGetQuaternion (dGeomID, dQuaternion result);
```

Первые две функции возвращают указатель на вектор позиции и матрицу вращения. Возвращаемые значения являются указателями на внутренние структуры данных, поэтому вектора актуальны пока в системе не произойдут изменения. Если геометрия ассоциирована с телом, то будут возвращены указатели на позицию/ориентацию тела, т.е. результат будет такой же как при вызове `dBodyGetPosition` или `dBodyGetRotation`.

`dGeomGetQuaternion` копирует кватернион геометрии в подготовленное место. Если геометрия ассоциирована с телом, то будет возвращен кватернион тела, т.е. результат будет такой же как при вызове `dBodyGetQuaternion`.

Вызов этих функций для не-перемещаемой геометрии будет выдавать ошибку(runtime error) в отладочном режиме(debug build) ODE.

```
void dGeomGetAABB (dGeomID, dReal aabb[6]);
```

Возвращает оси `aabb` соответствующие ограничивающему параллелепипеду(bounding box), который окружает данную геометрию. `aabb` массив имеет следующие элементы (*minx*, *maxx*, *miny*, *maxy*, *minz*, *maxz*). Если геометрия является пространством, то будет возвращен ограничивающий параллелепипед окружающий всю геометрию в этом пространстве.

Эта функция может вернуть ранее рассчитанный ограничивающий параллелепипед, это происходит когда геометрия не двигается.

```
int dGeomIsSpace (dGeomID);
```

Возвращает 1, если заданная геометрия является пространством, иначе 0.

```
dSpaceID dGeomGetSpace (dGeomID);
```

Возвращает пространство в которое входи заданная геометрия, или 0 если геометрия не входит ни в какое пространство.

```
int dGeomGetClass (dGeomID);
```

Возвращается номер класса для данной геометрии. Стандартные классы имеют следующие номера:

`dSphereClass`	Сфера
`dBoxClass`	Прямоугольный параллелепипед
`dCCylinderClass`	Цилиндр с верхушкой
`dCylinderClass`	Цилиндр
`dPlaneClass`	Бесконечная плоскость (не-перемещаема)
`dGeomTransformClass`	Преобразование геометрии
`dRayClass`	Луч
`dTriMeshClass`	Набор треугольников
`dSimpleSpaceClass`	Простое пространство
`dHashSpaceClass`	Пространство основанное на сборной таблице

Классы, определенные пользователем, будут возвращать номера своих классов.

```
void dGeomSetCategoryBits (dGeomID, unsigned long bits);
void dGeomSetCollideBits (dGeomID, unsigned long bits);
unsigned long dGeomGetCategoryBits (dGeomID);
unsigned long dGeomGetCollideBits (dGeomID);
```

Устанавливает и получает битовые поля “категории”(category) и “столкновения”(collide) для заданной геометрии. Эти битовые поля используются пространствами для управления того как геометрия будет взаимодействовать друг с другом. Битовые поля состоят не менее чем из 32 бит. По умолчанию все биты категории и столкновения, только созданного объекта, установлены.

```
void dGeomEnable (dGeomID);
void dGeomDisable (dGeomID);
int  dGeomIsEnabled (dGeomID);
```

Включает и выключает геометрию. Выключенная геометрия полностью игнорируется `dSpaceCollide` и `dSpaceCollide2`, несмотря на то что она является членом пространства.

`dGeomIsEnabled()` возвращает 1 если геометрия включена и 0 если выключена. Новая геометрия создается во включенном состоянии.

10.5. Определение столкновений

“Мир” определения столкновений создается посредством пространства с последующим добавлением геометрии в это пространство. На каждом шаге времени мы хотим получать список контактов для всех геометрий которые пересеклись. Для этого используются три функции:

- `dCollide` пересекает две геометрии и генерирует точки контакта.

- `dSpaceCollide` определяет какие пары в пространстве могут потенциально пересечься и вызывает функцию обратного вызова для каждой такой пары. Эта функция не генерирует точки контакта потому что пользователь может захотеть выделить некоторые пары – например некоторые проигнорировать или использовать другую стратегию создания точек. Это решение принимается в функции обратного вызова, в ней же можно решить использовать `dCollide` или нет для каждой пары.

- `dSpaceCollide2` определяет какие геометрии из одного пространства могут потенциально пересекаться с другими геометриями из другого пространства и вызывает функцию обратного вызова для каждой такой пары. Она также тестирует геометрию не из пространства на пересечение с пространством. Эта функция полезна для организации иерархии столкновений, т.е. когда пространства содержат другие пространства.

Система столкновений была разработана так чтобы дать пользователю максимальную гибкость в решении какие объекты тестировать на пересечение. Поэтому функции три, а не например одна для генерации всех точек контактов.

Пространства могут содержать другие пространства. Эти подпространства обычно представляют из себя набор геометрии (или других пространств) расположенных рядом друг с другом. Деление мира столкновений на иерархию полезно для увеличения производительности. Вот пример когда из этого можно извлечь пользу:

Допустим, у нас есть две машины, перемещающиеся по поверхности земли. Каждая машина состоит из множества геометрий. Если всю геометрию машин разместить в одном пространстве, то время необходимое на определение столкновения между машинами будет пропорционально общему количеству геометрии (или даже количеству геометрии в квадрате в зависимости от типа пространства).

Увеличить скорость определения столкновений можно, представив каждую машину своим пространством. Геометрию машин разместить в пространстве-машин(car-spaces), а пространства машин разместить в пространстве высшего уровня. На каждом шаге времени `dSpaceCollide` будет вызываться для пространства высшего уровня. Здесь будет происходить проверка на пересечение пространств-машин между собой (а именно между ограничивающими параллелепипедами) и если пересечение произошло будет вызываться функция обратного вызова. Функция обратного вызова затем может проверить геометрию в пространствах-машин с помощью функции `dSpaceCollide2`. Если машины находятся не рядом, то функция обратного вызова не будет вызываться и таким образом не будет в пустую тратиться время на ненужный тест.

Если используется иерархия пространств, тогда функция обратного вызова может вызываться рекурсивно, т.е. если `dSpaceCollide` вызывает функцию обратного вызова, которая вызывает `dSpaceCollide` с той же функцией обратного вызова. В этом случае пользователь должен быть уверен, что функция обратного вызова правильно входит второй раз.

Вот пример функции обратного вызова, которая просматривает все пространства и подпространства, генерируя все возможные точки контактов пересекающейся геометрии:

```
void nearCallback (void *data, dGeomID o1, dGeomID o2)
  {
    if (dGeomIsSpace (o1) || dGeomIsSpace (o2)) {
      // столкновение пространства с чем-нибудь
      dSpaceCollide2 (o1,o2,data,&nearCallback);
      // столкновения геометрии внутри пространства(пространств)
      if (dGeomIsSpace (o1)) dSpaceCollide (o1,data,&nearCallback);
      if (dGeomIsSpace (o2)) dSpaceCollide (o2,data,&nearCallback);
    }
    else {
      // столкновение двух геометрий не из пространств, и генерация точек
      // контакта между o1 и o2
      int num_contact = dCollide (o1,o2,max_contacts,contact_array,skip);
      // добавление этих точек контакта в симуляцию
      ...
    }
  }

  ...

  // столкновение всех объектов друг с другом
  dSpaceCollide (top_level_space,0,&nearCallback);
```

Функция обратного вызова пространства не позволяет модифицировать пространство пока этим пространством занимаются `dSpaceCollide` и `dSpaceCollide2`. Например, вы не можете добавлять или удалять геометрию из пространства, и вы не можете перемещать геометрию внутри пространства. Сделав так вы вызовете ошибку(runtime error) в отладочном режиме(debug build) ODE.

10.5.1. Битовые поля категории и столкновения

Каждая геометрия имеет битовое поле “категории”(category) и “столкновения”(collide), которые могут помочь в алгоритмах пространств(space algorithms) для определения того как должны взаимодействовать геометрии. Использование этих возможности опционально – по умолчанию считается что геометрия может столкнуться с любой другой геометрией.

Каждый бит поля значит различную категорию объекта. Текущее значение категорий определяется пользователем. Битовые поля категории относят геометрию к той или иной категории. Битовое поле столкновения указывает, какие категории геометрии могут сталкиваться при определении столкновения.

Пары геометрии могут посылаться функциями `dSpaceCollide` и `dSpaceCollide2` на рассмотрение в функцию обратного вызова только если установленный бит столкновения одной геометрии совпадает с установленным битом категории другой геометрии. Точная проверка осуществляется как показано ниже:

```
// проверка, может ли геометрия o1 и геометрия o2 столкнуться
  cat1 = dGeomGetCategoryBits (o1);
  cat2 = dGeomGetCategoryBits (o2);
  col1 = dGeomGetCollideBits (o1);
  col2 = dGeomGetCollideBits (o2);
  if ((cat1 & col2) || (cat2 & col1)) {
```

```
    // вызывается функция обратного вызова с o1 и o2
  }
  else {
    // ничего не делать, o1 и o2 не могут столкнуться
  }
```

Учтите что только dSpaceCollide и dSpaceCollide2 используют эти битовые поля, в dCollide они игнорируются.

Обычно геометрия принадлежит к какой то одной категории, поэтому только один бит категории должен быть установлен. Гарантированная ширина битового поля составляет 32 бита, поэтому пользователь может определять произвольное взаимодействие до 32 объектов. Если существует более 32 объектов, то некоторые из них должны иметь одинаковую категорию.

Иногда в поле категории может быть установлено несколько битов, т.е. если геометрия является пространством, то вы можете захотеть установить категорию пространства всем геометриям принадлежащим этому пространству.

Замечание разработчика: Почему бы не иметь одно битовое поле категории и делать проверку (`cat1 & cat2`)? Так проще, но одно поле требует больше битов для описания всех аспектов взаимодействия. Например, если 32 геометрии представляют 5-мерный гиперкуб, то требуется 80 бит для описания простейшей схемы. Простейшая схема также делает труднее определение того какие категории будут в той или иной ситуации.

10.5.2. Функции определения столкновения

```
int dCollide (dGeomID o1, dGeomID o2, int flags,
              dContactGeom *contact, int skip);
```

Если заданные геометрии o1 и o2 потенциально пересекаются, то генерируется информация о контакте. Только с помощью этой функции можно получить информацию о контакте.

flags определяет как информация о контакте должна быть сгенерированна если произошло касание. Младшие 16 бит представляют из себя целое число которое определяет максимальное количество течек контакта которые могут быть сгенерированны. Учите что определив количество как ноль, функция посчитает его как один – другими словами вы не можете запросить ноль контактов. Все другие биты flags должны быть сброшены. В будущем они могут быть задействованы для каких-нибудь целей.

contact указывает на массив dContactGeom структур. Массив как минимум должен вмещать максимально запрошенное количество точек контакта. Структура dContactGeom может быть вставлена в большую структуру – параметр skip определяет смещение одной dContactGeom от другой. Если skip равна размеру dContactGeom, тогда contact указывает на обычный(C-стиль) массив. Является ошибкой указывать skip меньше чем размер dContactGeom.

Если геометрии пересекаются, эта функция возвращает количество точек контакта (и обновляет contact массив), в противном случае возвращается ноль (и contact массив не затрагивается).

Если o1 или o2 являются пространствами, то будут проверены на столкновения все объекты из пространства o1 со всеми объектами из пространства o2, и возвращены точки их контакта. Этот метод столкновений пространств с геометрией (или пространства с пространством) не дает пользователю контроля над отдельным столкновением. Для получения этого контроля необходимо использовать dSpaceCollide и dSpaceCollide2.

Если o1 и o2 являются одной и той же геометрией, то будет возвращен 0. Технически говоря объект пересекается сам с собой, но бесполезно находить точки контакта в этом случае.

Этой функции все равно является ли o1 и o2 одним и тем же пространством или нет (или подпространством другого пространства).

```
void dSpaceCollide (dSpaceID space,
                    void *data, dNearCallback *callback);
```

Эта функция определяет какая пара геометрии в пространстве может потенциально пересечься и вызывает функцию обратного вызова для каждой такой пары. Функция обратного вызова имеет тип dNearCallback и определена как:

```
typedef void dNearCallback (void *data, dGeomID o1, dGeomID o2);
```

Аргумент data передается из dSpaceCollide прямо в функцию обратного вызова. Это значит то что определит пользователь. Аргументы o1 и o2 являются геометриями, которые могут находиться рядом друг с другом.

Функция обратного вызова может вызывать dCollide для o1 и o2 для генерирования точек контакта между каждой парой. Далее эти точки контакта могут быть добавлены в симуляцию как контактные сочленения. В функции обратного вызова пользователь конечно может и не вызывать dCollide, если решит что пара не должна пересекаться.

Сталкивающиеся пространства не обрабатываются особым образом, т.е. не происходит рекурсивного вызова. Функция обратного вызова может передать эти пространства в качестве одного или двух аргументов геометрии.

dSpaceCollide() гарантировано передаст все пары пересекающихся геометрий в функцию обратного вызова, но могут быть и ошибки и быть переданы не пересекающиеся пары. Количество ошибок зависит от внутреннего алгоритма, используемого в данном пространстве. Таким образом вы не должны ожидать что dCollide вернет контакты для всех пар переданных в функцию обратного вызова.

```
void dSpaceCollide2 (dGeomID o1, dGeomID o2,
                     void *data, dNearCallback *callback);
```

Эта функция схожа с dSpaceCollide за исключением того что передаются две геометрии (или пространства) в качестве аргументов. Функция обратного вызова вызывается для всех потенциально пересекающихся пар с одной геометрией из o1 и одной геометрией из o2.

Точное поведение зависит от типов o1 и o2:

- Если один аргумент является геометрией, а другой пространством, то будет вызвана функция обратного вызова для всех потенциальных пересечений между данной геометрией и объектами из этого пространства.
- Если оба o1 и o2 являются пространствами, тогда функция обратного вызова вызывается для всех потенциально пересекающихся пар с одной геометрией их пространства o1 и одной геометрией из пространства o2. Используемые алгоритмы зависят от типов пространств, в которых происходят столкновения. Если используется алгоритм без оптимизаций, то эта функция работает по одному из следующих принципов:
  1. Вся геометрия из o1 тестируется один-к-одному с o2.
  2. Вся геометрия из o2 тестируется один-к-одному с o1.

  Принципы могут зависеть от количества правил, но в основном геометрия пространства с меньших количеством объектов тестируется один-к-одному с другим пространством.
- Если оба аргумента одно и тоже пространство, это эквивалентно вызову dSpaceCollide с этим пространством.

- Если оба аргумента не являются пространствами, то функция обратного вызова вызывается один раз с этими аргументами.

Если один аргумент является пространством, а второй геометрией X из этого пространства, то такой случай не обрабатывается специально. В этом случае функция обратного вызова всегда будет вызываться с аргументами (X,X), потому что объект всегда пересекается сам с собой. Пользователь может организовать собственную проверку для игнорирования таких случаев, или просто передавать пару (X,X) в dCollide (она все равно гарантировано вернется 0).

10.6. Функции пространств

Существует несколько видов пространств. Каждый вид использует различную внутреннюю структуру данных для хранения геометрии и различные алгоритмы для выполнения отбора столкновений(collision culling).

- Простое пространство(simple space). Здесь не выполняется какого либо отбора столкновений – осуществляется простая проверка каждой возможной пары геометрии на пересечение, и сообщается о пересечении если их AABB'ы пересеклись. Время, требуемое на осуществление проверки пересечения для n объектов, равняется $O(n^2)$. Не может быть использовано для большого количества объектов, но этот алгоритм прекрасно подходит для небольшого количества объектов. Он также полезен в целях отладки потенциальных проблем в системе столкновений.

- Многомерное пространство, основанное на сборной таблице(multi-resolution hash table space). Здесь используется внутренняя структура данных, в которой записано как каждая геометрия перекрывает ячейки окружающей трехмерной сетки. Каждая ячейка представляет собой куб с длинной одной стороны равной 2^i, где i является целым, лежащим в диапазоне от минимального до максимального значения. Время, требуемое для осуществления проверки пересечения для n объектов, равняется $O(n)$ (до тех пор пока объекты не соберутся близко друг около друга), каждый объект может быстро образовать пару с другим объектом находящимся рядом.

- Пространство дерево квадрантов(quadtree space). Здесь используется заранее распределенное иерархическое AABB дерево для быстрого отбора столкновений. Особенно быстро работает для большого количества объектов в мире имеющего форму ландшафта. Количество используемой памяти равняется 4^глубина * 32 байта. На данный момент dSpaceGetGeom не реализована для пространства дерево квадрантов.

Вот функции, используемые для пространств:

```
dSpaceID dSimpleSpaceCreate (dSpaceID space);
dSpaceID dHashSpaceCreate (dSpaceID space);
```

Создается простое или многомерное, основанное на сборной таблице, пространство. Если `space` не равен нулю, то новое пространство вставляется в это пространство.

```
dSpaceID dQuadTreeSpaceCreate (dSpaceID space, dVector3 Center,
                               dVector3 Extents, int Depth);
```

Создается пространство дерево квадрантов. `center` и `extents` определяют размер корневого блока. `depth` задает глубину дерева – количество созданных блоков будет равняться 4^глубина.

```
void dSpaceDestroy (dSpaceID);
```

Уничтожает пространство. Эта функция в точности такая как dGeomDestroy за исключением того что она принимает `dSpaceID` в качестве аргумента. Когда уничтожается пространство, если режим очистки(cleanup mode) установлен в 1 (по умолчанию), вся геометрия в этом пространстве также уничтожается.

```
void dHashSpaceSetLevels (dSpaceID space, int minlevel, int maxlevel);
void dHashSpaceGetLevels (dSpaceID space, int *minlevel, int *maxlevel);
```

Устанавливает и получает некоторые параметры многомерного пространства, основанного на сборной таблице. Наименьшие и наибольшие размеры ячейки, используемые в сборной таблице, будут 2^`minlevel` и 2^`maxlevel` соответственно. `minlevel` должен быть меньше или равен `maxlevel`.

`dHashSpaceGetLevels` возвращает минимальное и максимально значение с помощью указателей. Если возвращается указатель ноль, то он игнорируется и ничего не возвращается.

```
void dSpaceSetCleanup (dSpaceID space, int mode);
int dSpaceGetCleanup (dSpaceID space);
```

Устанавливается и возвращается режим очистки в пространстве. Если режим очистки установлен в 1, то вся геометрия в пространстве будут уничтожено при уничтожении этого пространства. Если режим очистки установлен в 0, то этого не случится. Режим очистки по умолчанию, для нового пространства, равен 1.

```
void dSpaceAdd (dSpaceID, dGeomID);
```

Добавление геометрии в пространство. Ничего не происходит, если геометрия уже есть в пространстве. Эта функция вызывается автоматически, если аргумент `space` задан в функции создания геометрии.

```
void dSpaceRemove (dSpaceID, dGeomID);
```

Удаляет геометрию из пространства. Ничего не происходит, если геометрии нет в пространстве. Эта функция вызывается автоматически функцией dGeomDestroy, если геометрия есть в пространстве.

```
int dSpaceQuery (dSpaceID, dGeomID);
```

Возвращается 1, если заданная геометрия находится в заданном пространстве, иначе возвращается 0.

```
int dSpaceGetNumGeoms (dSpaceID);
```

Возвращается количество геометрии содержащейся в заданном пространстве.

```
dGeomID dSpaceGetGeom (dSpaceID, int i);
```

Возвращается `i`'я геометрия содержащаяся в заданном пространстве. `i` должно быть в диапазоне от 0 до `dSpaceGetNumGeoms()`-1.

Если происходят какие либо изменения в пространстве (включая добавление и удаление геометрий), то нет никаких гарантий как изменятся индексы для геометрий в этом пространстве.

Эта функция гарантировано работает быстрее когда доступ к геометрии осуществляется по порядку, т.е. 0,1, 2, т.д. Произвольный доступ гораздо медленнее и зависит от внутренней реализации.

10.7. Классы геометрии

10.7.1. Класс сфера

```
dGeomID dCreateSphere (dSpaceID space, dReal radius);
```

Создает геометрию сфера(sphere) заданного `radius` и возвращает его ID. Если `space` не равен нулю, то сфера вставляется в это пространство. Точкой расположения(point of reference) является центр сферы.

```
void dGeomSphereSetRadius (dGeomID sphere, dReal radius);
```

Задает радиус указанной сферы.

```
dReal dGeomSphereGetRadius (dGeomID sphere);
```

Возвращает радиус указанной сферы.

```
dReal dGeomSpherePointDepth (dGeomID sphere, dReal x, dReal y, dReal z);
```

Возвращает глубину точки (`x`,`y`,`z`) для указанной сферы. Точки внутри геометрии всегда будут иметь положительную глубину, точки снаружи будут иметь отрицательную глубину и точки на поверхности всегда будут иметь глубину ноль.

10.7.2. Класс прямоугольный параллелепипед

```
dGeomID dCreateBox (dSpaceID space, dReal lx, dReal ly, dReal lz);
```

Создает геометрию прямоугольный параллелепипед(box) с длинами сторон (`lx`,`ly`,`lz`) и возвращает его ID. Если `space` не равен нулю, то происходит вставка в это пространство. Точкой расположения является центр прямоугольного параллелепипеда.

```
void dGeomBoxSetLengths (dGeomID box, dReal lx, dReal ly, dReal lz);
```

Задает длины сторон указанного прямоугольного параллелепипеда.

```
void dGeomBoxGetLengths (dGeomID box, dVector3 result);
```

Возвращает в `result` длины сторон указанного `box`.

```
dReal dGeomBoxPointDepth (dGeomID box, dReal x, dReal y, dReal z);
```

Возвращает глубину точки (`x`,`y`,`z`) для указанного прямоугольного параллелепипеда. Точки внутри геометрии всегда будут иметь положительную глубину, точки снаружи будут иметь отрицательную глубину и точки на поверхности всегда будут иметь глубину ноль.

10.7.3. Класс плоскость

```
dGeomID dCreatePlane (dSpaceID space,
                      dReal a, dReal b, dReal c, dReal d);
```

Создает геометрию плоскость(plane) с заданными параметрами и возвращает ее ID. Если `space` не равен нулю, то происходит вставка в это пространство. Уравнение плоскости имеет вид:

$$a*x+b*y+c*z = d$$

Вектор нормали к плоскости равен (a,b,c) и всегда должен иметь длину 1. Плоскости не перемещаемая геометрия. Это значит что в отличие от перемещаемой геометрии, у плоскости нет позиции и вращения. Это значит что параметры (a,b,c) всегда заданы в глобальных координатах. Другими словами предполагается что плоскость является частью статической геометрии и не связана с каким либо подвижным объектом.

```
void dGeomPlaneSetParams (dGeomID plane, dReal a, dReal b, dReal c, dReal d);
```

Задает параметры указанной `plane`.

```
void dGeomPlaneGetParams (dGeomID plane, dVector4 result);
```

Возвращает в `result` параметры указанной `plane`.

```
dReal dGeomPlanePointDepth (dGeomID plane, dReal x, dReal y, dReal z);
```

Возвращает глубину точки (`x`,`y`,`z`) для указанной плоскости. Точки внутри геометрии всегда будут иметь положительную глубину, точки снаружи будут иметь отрицательную глубину и точки на поверхности всегда будут иметь глубину ноль.

10.7.4. Класс цилиндр с верхушкой

```
dGeomID dCreateCCylinder (dSpaceID space, dReal radius, dReal length);
```

Создает геометрию цилиндр с верхушкой(capped cylinder) с заданными параметрами и возвращает его ID. Если `space` не равен нулю, то происходит вставка в это пространство.

Цилиндр с верхушкой похож на обычный цилиндр за исключением того что на концах цилиндра расположено по полусфере. Эта возможность делает внутренний код определения столкновения более точным и быстрым. В длину цилиндра `length` верхушки не входят. Предполагается что ось цилиндра располагается вдоль локальной оси геометрии Z. Радиус верхушек равен радиусу самого цилиндра и задается с помощью параметра `radius`.

```
void dGeomCCylinderSetParams (dGeomID ccylinder,
                              dReal radius, dReal length);
```

Задает параметры указанного цилиндра с верхушкой.

```
void dGeomCCylinderGetParams (dGeomID ccylinder,
                              dReal *radius, dReal *length);
```

Возвращает параметры `radius` и `length` указанного цилиндра с верхушкой.

```
dReal dGeomCCylinderPointDepth (dGeomID ccylinder,
                                dReal x, dReal y, dReal z);
```

Возвращает глубину точки (`x`,`y`,`z`) для указанного цилиндра с верхушкой. Точки внутри геометрии всегда будут иметь положительную глубину, точки снаружи будут иметь отрицательную глубину и точки на поверхности всегда будут иметь глубину ноль.

10.7.5. Класс луч

Луч(ray) отличается от всех классов геометрий тем что он не представлен твердым объектом(solid object). Он представляет из себя бесконечную тонкую линию, начинающеюся в точке расположения геометрии и имеющую направление вдоль локальной оси геометрии Z.

Вызывая `dCollide` между лучом и другой геометрией можно получить более одной точки контакта. Лучи имеют собственное соглашение об информации о контакте для `dContactGeom` структуры (поэтому не очень удобно создавать контактное сочленение на основании этой информации).

- `pos` – Это точка в которой луч пересек поверхность геометрии, независимо от того луч вошел или вышел из геометрии.
- `normal` – Это нормаль к поверхности геометрии в точке контакта. Если в `dCollide` первым параметром передается геометрия луч, тогда нормаль будет ориентирована правильно (в противном случае она будет иметь противоположный знак).
- `depth` – Это расстояние от начала луча до точки контакта.

Лучи полезны для таких вещей как проверка видимости, определение пути снарядов или лучей света и для размещения объектов.

```
dGeomID dCreateRay (dSpaceID space, dReal length);
```

Создает геометрию луч заданной длины и возвращает его ID. Если `space` не равен нулю, то происходит вставка в это пространство.

```
void dGeomRaySetLength (dGeomID ray, dReal length);
```

Задает длину указанного `ray`.

```
dReal dGeomRayGetLength (dGeomID ray);
```

Получает длину указанного `ray`.

```
void dGeomRaySet (dGeomID ray, dReal px, dReal py, dReal pz,
                  dReal dx, dReal dy, dReal dz);
```

Задает начальную позицию (`px`,`py`,`pz`) и направление (`dx`,`dy`,`dz`) указанного `ray`. Матрица вращения будет применена к лучу полагая что локальная ось Z является первоначальным направлением. Учтите что в этой функции не указывается длина луча.

```
void dGeomRayGet (dGeomID ray, dVector3 start, dVector3 dir);
```

Получает начальную позицию (`start`) и направление (`dir`) луча. Возвращенное значение вектора направления будет в единицах масштаба(unit length).

10.7.6. Класс набор треугольников

Набор треугольников(triangle mesh) (TriMesh) представляет собой произвольный набор треугольников. Система столкновений набора треугольников имеет следующие возможности:

- Может быть представлен любой “суп” из треугольников – т.е. треугольники не обязательно должны представлять собой ленту(strip), веер(fan) или сетку(grid).
- Работает хорошо для относительно больших треугольников.
- Используется временная связность(temporal coherence) для ускорения проверки столкновений. Когда происходит проверка столкновения геометрии с одним trimesh’ем, данные располагаются внутри trimesh’а. Эти данные могут быть очищены с помощью функции `dGeomTriMeshClearTCCache`. В будущем будет возможность отключать эту функцию.

Столкновение Trimesh/Trimesh выполняется достаточно хорошо, но есть несколько незначительных предостережений:

- Размер шага, который вы будете использовать, должен быть уменьшен для большей точности столкновений. Столкновения не-выпуклых(non-convex) форм больше зависят от этих форм чем столкновения примитивов. В будущем локальная геометрия контакта(local contact geometry) будет изменена более основательно (и более сложным образом) для не-выпуклых объектов(polytopes) с целью их упрощения до выпуклых объектов таких как сферы и кубы.

- В целях эффективности при работе со столкновениями, функции dCollideTTL нужны позиции сталкивающихся trimesh’ей на предыдущем шаге времени. Это необходимо для вычисления предполагаемой скорости каждого сталкивающегося треугольника, которая потом будет использована для нахождения направления проникновения, нормалей контакта и т.д. Это требует от пользователя обновлять эти переменные на каждом шаге времени. Такое обновление производится вне ODE т.к. оно не входит в ODE. Код который это делает выглядит примерно так:

```
const double *DoubleArrayPtr =
  Bodies[BodyIndex].TransformationMatrix->GetArray();
dGeomTriMeshDataSet( TriMeshData,
  TRIMESH_LAST_TRANSFORMATION,
  (void *) DoubleArrayPtr );
```

Матрицей трансформации является стандартная гомогенная(homogeneous) матрица трансформации 4x4, а “DoubleArray” является стандартным плоским массивом из 16 значений этой матрицы.

ЗАМЕЧАНИЕ: Класс набор треугольников не закончен, поэтому в будущих API можно ожидать изменений.

```
dTriMeshDataID dGeomTriMeshDataCreate();
void dGeomTriMeshDataDestroy (dTriMeshDataID g);
```

Создает и уничтожает dTriMeshData объект вместе с расположенными данными пользователя.

```
void dGeomTriMeshDataBuild (dTriMeshDataID g, const void* Vertices,
                            int VertexStride, int VertexCount,
                            const void* Indices, int IndexCount,
                            int TriStride, const void* Normals);
```

Используется для заполнения `dTriMeshData` объекта данными. Здесь не происходит копирование данных, поэтому указатели должны указывать на данные. Вот как работает расположение данных:

```
struct StridedVertex {
  dVector3 Vertex;  // 4-ех компонентных векторов можно не использовать, для уменьшения использования памяти
  // Данные пользователя
};
int VertexStride = sizeof (StridedVertex);

struct StridedTri {
  int Indices[3];
  // Данные пользователя
};
int TriStride = sizeof (StridedTri);
```

Аргумент `Normals` опционален: нормали к каждой плоскости(faces) trimesh объекта. Например,

```
  dTriMeshDataID TriMeshData;
  TriMeshData = dGeomTriMeshGetTriMeshDataID (
        Bodies[BodyIndex].GeomID);

  // как long так и dReal == числа с плавающей запятой(floats)
  dGeomTriMeshDataBuildSingle (TriMeshData,
        // Вершины
        Bodies[BodyIndex].VertexPositions,
        3*sizeof(dReal), (int) numVertices,
        // Плоскости(Faces)
        Bodies[BodyIndex].TriangleIndices,
        (int) NumTriangles, 3*sizeof(unsigned int),
        // Нормали
        Bodies[BodyIndex].FaceNormals);
```

Заранее рассчитанные нормали сокращают время расчета контакта, но в этом нет необходимости. Если вы не хотите вычислять нормали к плоскостям (или если у вас огромные trimesh'ы и вы знаете что могут касаться только несколько плоскостей, и хотите сэкономить время), просто передайте "NULL" в качестве `Normals` аргумента, и dCollideTTL позаботится об вычислении нормалей сама.

```
void dGeomTriMeshDataBuildSimple (dTriMeshDataID g, const dVector3*Vertices,
                                  int VertexCount, const int* Indices,
                                  int IndexCount);
```

Функция простого формирования(simple build) присутствует просто для удобства.

```
typedef int dTriCallback (dGeomID TriMesh, dGeomID RefObject, int TriangleIndex);
void dGeomTriMeshSetCallback (dGeomID g, dTriCallback *Callback);
dTriCallback* dGeomTriMeshGetCallback (dGeomID g);
```

Опциональный вызов функции обратного вызова для произвольного треугольника. Позволяет пользователю сказать что ему нужен контроль над столкновением конкретного треугольника. Если возвращается ноль, значит контакта не было.

```
typedef void dTriArrayCallback (dGeomID TriMesh, dGeomID RefObject,
                                const int* TriIndices, int TriCount);
void dGeomTriMeshSetArrayCallback (dGeomID g, dTriArrayCallback* ArrayCallback);
dTriArrayCallback *dGeomTriMeshGetArrayCallback (dGeomID g);
```

Опциональный вызов функции обратного вызова для произвольной геометрии. Позволяет пользователю получить список всех пересекшихся треугольников за один прием.

```
typedef int dTriRayCallback (dGeomID TriMesh, dGeomID Ray, int TriangleIndex,
                             dReal u, dReal v);
void dGeomTriMeshSetRayCallback (dGeomID g, dTriRayCallback* Callback);
dTriRayCallback *dGeomTriMeshGetRayCallback (dGeomID g);
```

Опциональный вызов функции обратного вызова луча(ray). Позволяет пользователю определить пересечение луча с конкретным треугольником на основании координат пересечения. Пользователь например может организовать битовый массив для указания с какими треугольникам пересекся луч.

```
dGeomID dCreateTriMesh (dSpaceID space, dTriMeshDataID Data,
                        dTriCallback *Callback,
                        dTriArrayCallback * ArrayCallback,
                        dTriRayCallback* RayCallback);
```

Конструктор. `Data` является членом данных о вершинах только что созданного набора треугольников.

```
void dGeomTriMeshSetData (dGeomID g, dTriMeshDataID Data);
```

Перемещает текущие данные.

```
void dGeomTriMeshClearTCCache (dGeomID g);
```

Очищает внутренние кэши(cashes) временных связей(temporal coherence).

```
void dGeomTriMeshGetTriangle (dGeomID g, int Index, dVector3 *v0,
                              dVector3 *v1, dVector3 *v2);
```

Извлекает треугольник из объектного пространства. Аргументы `v0`, `v1` и `v2` опциональны.

```
void dGeomTriMeshGetPoint (dGeomID g, int Index, dReal u, dReal v,
                           dVector3 Out);
```

Извлекает позицию из объектного пространства на основании входных данных.

```
void dGeomTriMeshEnableTC(dGeomID g, int geomClass, int enable);
int dGeomTriMeshIsTCEnabled(dGeomID g, int geomClass);
```

Эти функции могут быть использованы для включения/выключения использования временных связей во время проверки столкновений треугольников. Временная связь может быть включена/выключена для пары: набор треугольников/класс геометрии, на данный момент работает только для сфер и прямоугольных параллелепипедов. Значение по умолчанию для сфер и прямоугольных параллелепипедов 'false'.

Параметр 'enable' должен быть 1 в качестве истины(true) и 0 в качестве ложь(false).

Временная связь опциональна, потому что возникают проблемы эффективности в случаях когда tri-mesh может столкнуться со многими другими геометриями на протяжении своего времени жизни. Если вы разрешили временную связь для tri-mesh'а, то проблему можно упростить иногда вызывая dGeomTriMeshClearTCCache.

10.7.7. Класс трансформации геометрии

Трансформация геометрии 'T' есть геометрия которая инкапсулирована(encapsulates) в другую геометрию 'E', позволяя E быть размещенной и повернутой произвольным образом относительно точки расположения(point of reference).

Большинство перемещаемых геометрий (таких как сферы и прямоугольные параллелепипеды) имеют свои точки расположения, являющиеся их центрами масс, и позволяют им легко быть присоединенным к динамическим объектам. Трансформация объектов дает вам больше гибкости – например вы можете сместить центр сферы или перевернуть цилиндр так чтобы его ось стала отличной от оси по умолчанию.

T имитирует объект E следующим образом: T находится в том же пространстве и присоединен к тому же телу что и E. Сам по себе E может не находиться в каком либо пространстве или быть присоединенным к какому то телу. Позиция и вращение E устанавливаются как константные значения которые говорят как E расположен *относительно* T. Если позиция и вращение оставлено в значениях по умолчанию, то T будет вести себя в точности как E если вы захотите использовать его напрямую.

```
dGeomID dCreateGeomTransform (dSpaceID space);
```

Создает новый объект трансформации геометрии(geometry transform) и возвращает его ID. Если `space` не равен нулю, то происходит вставка в это пространство. При создании инкапсулированная геометрия устанавливается в 0.

```
void dGeomTransformSetGeom (dGeomID g, dGeomID obj);
```

Устанавливает геометрию которая будет являться трансформированной геометрией `g`. Объект `obj` не должен принадлежать ни какому пространству и не быть ассоциированным с каким либо телом.

Если для `g` включен режим очистки(clean-up mode) и он уже является инкапсулированным объектом, то старый объект будет уничтожен прежде чем быть замененным на новый.

```
dGeomID dGeomTransformGetGeom (dGeomID g);
```

Получает геометрию, которая является трансформацией геометрии `g`.

```
void dGeomTransformSetCleanup (dGeomID g, int mode);
int dGeomTransformGetCleanup (dGeomID g);
```

Устанавливает и получает режим очистки трансформации геометрии `g`. Если режим очистки установлен в 1, тогда инкапсулированные объекты будут уничтожаться при уничтожении трансформированной геометрии. При установке режима очистки в 0 этого происходить не будет. Значение по умолчанию для режима очистки 0.

```
void dGeomTransformSetInfo (dGeomID g, int mode);
int  dGeomTransformGetInfo (dGeomID g);
```

Устанавливает и получает режим "информации" трансформированной геометрии `g`. Режим может быть 0 или 1. Значение по умолчанию 0.

При режиме 0, когда трансформированный объект сталкивается с другим объектом (используя `dCollide (tx_geom,other_geom,...)`), поле `g1` в `dContactGeom` структуре устанавливается таким же как поле *инкапсулирующего* трансформированный объект. Это значение `g1` позволяет вызывающему запрашивать тип трансформированной геометрии, но не позволяет определить позицию в глобальных координатах или ассоциированное с ним тело, поскольку оба эти свойства используются по разному для инкапсулированных геометрий.

При режиме 1, поле `g1` в `dContactGeom` структуре устанавливает сам трансформированный объект. Это делает объект таким же как и все остальные геометрии, поэтому `dGeomSetBody` вернет присоединенное тело, а `dGeomGetPosition` вернет глобальную позицию. Для получения текущего типа инкапсулированной геометрии необходимо воспользоваться dGeomTransformGetGeom.

10.8. Классы определенные пользователем

Классы геометрии ODE реализованы внутри как C++ классы. Если вы хотите определить свои классы, то это можно сделать двумя способами:

1. Использовать C функции, описанные в этом разделе. Преимущество этого метода состоит в том что обеспечивается четкое разделение между вашим кодом и кодом ODE.
2. Добавит классы прямо в исходный код ODE. Преимущество состоит в том что вы можете использовать для реализации C++, что немного упрощает реализацию. Этот метод предпочтительнее в случае если ваш класс определения столкновения хорошо реализован и вы хотите внести свой вклад в базу свободных программ.

Далее следует C API для классов определяемых пользователем.

Каждый класс пользователя должен иметь уникальный целый номер. Новый класс геометрии (назовем его 'X') должен предоставить ODE:

1. Функции, обрабатывающие определение столкновений и генерацию контактов между X и другими классами. Эти функции должны иметь тип `dColliderFn`, и быть определены следующим образом

```
typedef int dColliderFn (dGeomID o1, dGeomID o2, int flags,
                         dContactGeom *contact, int skip);
```

Интерфейс такой же как и у dCollide. Каждая функция должна обрабатывать свой случай столкновения, где `o1` имеет тип X, а `o2` любого другого типа.
2. Функция "селектор" имеет тип `dGetColliderFnFn` и должна быть определена как

```
typedef dColliderFn * dGetColliderFnFn (int num);
```

Эта функция принимает номер класса (num), и возвращает функцию столкновения, которая должна обрабатывать столкновение класса X с классом num. Ноль должен возвращаться в случае если не известно как X должна сталкиваться с num. Учтите, что если сталкиваются два класса X и Y, должна быть только *одна* функция для обработки столкновения.

Эта функция вызывается редко – возвращаемые значения кэшируются и используются повторно.

3. Функция, которая должна вычислять оси соответствующие ограничивающему параллелепипеду (AABB) для данного класса. Эта функция должна иметь тип dGetAABBFn, определенный следующим образом

```
typedef void dGetAABBFn (dGeomID g, dReal aabb[6]);
```

Эта функция принимает g, которая имеет тип X, и возвращает выровненные оси ограничивающего параллелепипеда для g. Массив aabb состоит из элементов (*minx*, *maxx*, *miny*, *maxy*, *minz*, *maxz*). Если вы не хотите вычислять значения AABB ограничивающего параллелепипеда, то вы просто можете определить указатель на dInfiniteAABB, которая возвращает +/- бесконечность в каждом направлении.

4. Количество байт “данных класса” которые необходимы этому классу. Например сфера хранит радиус в области данных своего класса, а прямоугольный параллелепипед хранит длины своих сторон.

Следующие возможности не обязательны для классов геометрии:

1. Функция, которая бы уничтожала данные класса. Большинство классов не нуждаются в подобной функции, но некоторые классы могут захотеть освободить захваченную память или ресурсы. Эта функция должна иметь тип dGeomDtorFn, определенный как

```
typedef void dGeomDtorFn (dGeomID o);
```

Аргумент o должен иметь тип X.

2. Функция, которая бы тестировала пересечение AABB класса X с AABB других классов. Она используется для раннего теста в функции столкновений. Эта функция должна иметь тип dAABBTestFn, и определена как

```
typedef int dAABBTestFn (dGeomID o1, dGeomID o2, dReal aabb2[6]);
```

Аргумент o1 имеет тип X. Если эта функция определена, то она вызывается функцией dSpaceCollide, где o1 пересекается с геометрией o2, AABB которой задан в aabb2.1 возвращает, если aabb2 пересекается с o1, в противном случае 0.

Это полезно для большой поверхности земли. Земля обычно имеет очень большой AABB и не имеет особого смысла проводить проверку на пересечение ее с другими объектами. Эта функция может проводить проверку AABB других объектов с землей, избегая проблем с вызовом специфических функция определения столкновения. Особенная экономия времени получается при проверке объектов GeomGroup.

Вот функции для управления определенным классом:

```
int dCreateGeomClass (const dGeomClass *classptr);
```

Регистрируется новый класс геометрии определенный в classptr. Возвращается номер нового класса. По соглашению, принятому в ODE, номер класса считается глобальной переменной с именем dXxxClass, где Xxx имя класса (напр. dSphereClass).

Вот определение dGeomClass структуры:

```
struct dGeomClass {
  int bytes;                  // количество байт необходимых данным
  dGetColliderFnFn *collider; // функция обработки столкновения
  dGetAABBFn *aabb;           // функция ограничивающего параллелепипеда
  dAABBTestFn *aabb_test;     // aabb проверка, должна быть 0 если отсутствует
  dGeomDtorFn *dtor;          // деструктор, должен быть 0 если отсутствует
};
```

```
void * dGeomGetClassData (dGeomID);
```

Возвращает указатель на данные пользователя заданной геометрии (это будет блок из необходимого количества байт).

```
dGeomID dCreateGeom (int classnum);
```

Создает геометрию с заданным номером класса. Блок необходимых данных изначально установлен в 0. Объект может быть добавлен в пространство с помощью dSpaceAdd.

При реализации нового класса обычно необходимо сделать следующие:

1. Если класс еще не был создан, создать его. Необходимо быть осторожным и создавать класс только один раз.
2. Вызвать dCreateGeom для определения этого класса.
3. Определить необходимые данные этого класса.

10.9. Составные объекты

Рассмотрим следующие объекты:

- Стол, представленный прямоугольным параллелепипедом сверху и прямоугольными параллелепипедам в качестве ножек.
- Ветви деревьев, смоделированные при помощи нескольких цилиндров сочлененных вместе.
- Молекулы, в которых каждый атом представлен сферой.

Если предполагается что эти объекты *жестким*(*rigid*), то необходимо использовать отдельные жесткие тела для их представления. Но это может сказаться на скорости определения столкновения, потому что нет отдельного класса геометрии для представления сложных форм таких как стол или молекула. Решением является использование *составных* объектов, представляющих комбинацию из нескольких геометрий.

Нет дополнительный функций для управления составными объектами: просто создавайте каждый компонент геометрии и присоединяйте его к тому же самому телу. Двигайте и вращайте отдельные геометрии вместе в одном объекте, трансформация геометрии может быть использована для инкапсуляции.

Вот все что необходимо сделать!

Тем не менее есть одно *предостережение*: Вы никогда не должны создавать составные объекты у которых точки контакта находились бы близко друг к другу. Например рассмотрим стол, сделанный из прямоугольного параллелепипеда в качестве верхней части и прямоугольных параллелепипедов в качестве ножек. Если ножки соединены с верхом и стол лежит на боку, то точки контакта для прямоугольных параллелепипедов могут совпасть в том месте где ножки соединены с верхом. ODE не содержит специальной обработки совпадающих точек контакта, поэтому такая ситуация может привести к многочисленным ошибкам и странному поведению.

В этом примере геометрия стола должна быть прилажена так чтобы ножки небыли на краю, располагая их так чтобы не было одинаково сгенерированных точек контакта. В основном избегать различных перекрывающихся(overlap) контактных поверхностей, или поверхностей у которых края лежат на одной линии.

10.10. Утилитарные функции

```
void dClosestLineSegmentPoints (const dVector3 a1, const dVector3 a2,
                                const dVector3 b1, const dVector3 b2,
                                dVector3 cp1, dVector3 cp2);
```

Для заданных двух отрезков(line segment) A и B с точками на концах `a1`-`a2` и `b1`-`b2`, возвращаются ближайшие друг к другу точки лежащие на A и B (в `cp1` и `cp2`). В случае параллельности линий существует множество решений, решение заключается в возврате конечных точек принадлежащих линиям. Так же корректно обрабатывается случай нулевой длины сегмента линии, напр. если `a1`=`a2` и/или `b1`=`b2`.

```
int dBoxTouchesBox (const dVector3 _p1, const dMatrix3 R1,
                    const dVector3 side1, const dVector3 _p2,
                    const dMatrix3 R2, const dVector3 side2);
```

Для заданных прямоугольных параллелепипедов (`p1`,`R1`,`side1`) и (`p2`,`R2`,`side2`) возвращается 1 если они пересеклись и 0 если нет. `p` является центром прямоугольного параллелепипеда, `R` матрица вращения и `side` вектор с длинами сторон x/y/z.

```
void dInfiniteAABB (dGeomID geom, dReal aabb[6]);
```

Эта функция может быть использована для получения AABB классом геометрии, если вы не хотите рассчитывать правильный AABB. Она возвращает +/- бесконечность в каждом направлении.

10.11. Замечания по реализации

10.11.1. Большие пространства

Часто мир столкновений содержит много объектов, которые являются частью статического окружения и не представлены каким либо жестким телом. Определение столкновений ODE содержит оптимизации для определения таких неподвижных геометрий и предварительного расчета необходимой информации для экономии времени вычислений. Например заранее рассчитываются ограничивающие прямоугольный параллелепипеды и внутренние структуры данных о столкновениях.

10.11.2. Использование другой библиотеки определения столкновений

Использование библиотеки определения столкновений ODE является опциональной возможностью – можно использовать стороннюю библиотеку определения столкновений, которая будет корректно заполнять `dContactGeom` структуру для инициализации контактных сочленений.

Динамическое ядро(dynamics core) ODE в основном не зависит от того какая библиотека определения столкновений используется, за исключением четырех пунктов:

1. Тип `dGeomID` должен быт определен, так как каждое тело может хранить указатель на объекты геометрии ассоциированный с ним.

2. Должна быть определена функция `dGeomMoved()`, имеющая следующий прототип:

```
void dGeomMoved (dGeomID);
```

Эта функция вызывается из динамического ядра ODE всякий раз когда тело движется: она указывает что геометрический объект, ассоциированный с телом, сейчас в новой позиции.

3. Должна быть определена функция `dGeomGetBodyNext()`, имеющая следующий прототип:

```
dGeomID dGeomGetBodyNext (dGeomID);
```

Эта функция вызывается динамическим ядром для просмотра списка геометрий ассоциированных с телом. Заданная геометрия присоединяется к телу, функция возвращает следующую геометрию присоединенную к этому телу, 0 если больше геометрий нет.

4. Должна быть определена функция `dGeomSetBody()`, имеющая следующий прототип:

```
void dGeomSetBody (dGeomID, dBodyID);
```

Эта функция вызывается в теле кода деструктора (со вторым аргументом установленным в 0) удаляя все ссылки на геометрии для тела.

Если вы хотите в другой библиотеке определения столкновений получать данные о движении тела из ODE, вы должны определить эти типы и функции соответствующим образом.

11.Как сделать хорошую симуляцию

[просто несколько замечаний]

11.1. Точность и стабильность интегрирования

- Интегрирование не дает точного решения
- Это стабильно
- Типы интегрирования (exp & imp, order)
- Компромисс между точностью, стабильностью и работоспособностью

11.2. Поведение может зависеть от размера шага

- Меньше шаг = больше точность, высшее стабильность
- 10*0.1 не то же самое что 5*0.2
- Влияет на скорость расчета симуляции

11.3. Как сделать быстрее

Какие факторы влияют на скорость исполнения? Каждое сочленение добавляет определенное количество степеней свободы (DOFs) в систему. Например шарик в разъеме(ball and socket) добавляет три, а скольжение(hinge) добавляет пять. Для каждой отдельной группы тел соединенных сочленением справедливо следующие:

- m_1 количество сочленений в группе
- m_2 общее количество DOF'ов порожденными этими сочленениями и
- n общее количество тел в группе,

тогда время расчета одного шага для группы пропорционально:

$$k_1 O(m_1) + k_2 O(m_2{}^3) + k_2 O(n)$$

На текущий момент в ODE реализовано разложение на множители "системной" матрицы в которой одна строка/столбец представляет один DOF (вот откуда взялся $O(m_2{}^3)$). Для цепочки из 10 тел использующих сочленения шарик в разъеме, примерно 30-40% времени тратится на заполнение этой матрицы и 30-40% занимает разложение на множители.

Таким образом можно заключить что для ускорения симуляции:

- Использовать меньше сочленений – часто небольшие тела и ассоциированные с ними сочленения могут быть представлены "ложной"(fakes) физикой без ущерба физическому реализму.
- Замещение множества сочленений на более простые. Это будет сделать проще с выходом специализированных типов сочленений.
- Использовать меньше контактов
- Предпочтительнее использовать где только возможно вязкие контакты и контакты с отсутствием трения (которые добавляют один DOF) чем контакты с Coulomb трением (которые добавляют пять DOF).

В следующих реализациях ODE будут реализованы методы для лучшей масштабируемости в зависимости от количества сочленений.

11.4. Улучшение стабильности

- негнущиеся пружины / явные силы это плохо.
- жесткие соединения это хорошо.
- интегрирование зависит от шага времени.
- Использовать усиление сочленений(powered joint), ограничение сочленений(joint limits), встроенные пружины(built-in springs) там где это возможно избегая явных сил.
- соотносить массы – т.е. распределять их. Сочленения, которые соединяют тела с большой и маленькой массой, требуют сокращения шага времени для уменьшения ошибок в вычислениях.
- Движение тела не должно быть быстрее чем движение которое может обработать данный шаг времени.
- инерция должна быть вдоль осей.

Увеличение глобальной CFM сделает систему более числено устойчивой(numerically robust) и менее подверженной проблемам стабильности. Так же система будет выглядеть более "вязко"(spongy), поэтому надо искать компромисс.

Излишние соединения (два или более соединений которые "пытаются и делают одну и ту же работу") будут мешать друг другу и вызывать проблемы со стабильностью. Количество зависит от проблем особенно в системной матрице(system matrix). Например если два контактных сочленения соединяют два тела в одной точке. Другой пример если виртуальное сочленение сгибание(virtual hinge joint) создается между двумя телами соединенными уже с помощью двух шариковых сочленений(ball joints), расположенных по разные стороны осей сгибания (это плохо тем что два шариковых сочленения пытаются внести в систему шесть степеней свободы, но настоящее сочленение сгибание может внести только пять).

Лишние соединения борются друг с другом и генерируют странные силы которые могут мешать нормальным силам. Например воздействовать на тело таким образом что оно будет летать вокруг живя собственной жизнью, абсолютно игнорируя гравитации.

11.5. Использование смешивающей силы соединения(CFM)

- приводит к единой конфигурации
- исправляет: дрожание(jitter) или странные силы, усиливающие ошибки, LCP решение может замедляться
- делает сочленения более податливыми(compliant joints) (что так же может быть и не желательно)

11.6. Избежание странного поведения

- Странности в поведении могут проявляться когда больше чем необходимо сочленений ограничивают движения тел.
- Множество (несовместимых) сочленений между телами, особенно сочленение + контакт (не позволяет объектам столкнуться, потому что они уже соединены вместе).
- Увеличивайте CFM.
- Непреднамеренное поведение - цепочка из прямоугольных параллелепипедов, лежащая на полу, сбивается вместе.
- Используйте минимальное количество сочленений для правильного поведения. Используйте правильные сочленения для достижения желаемого поведения.
- Добавление глобальной CFM обычно помогает.

11.7. Другие замечания

- Дрожь(jitter) в контактах происходит когда они удаляются достаточно далеко друг от друга.
- Массу и длину задавайте в районе 1.
- Решение LCP требует различного количества итераций (только для не детерминированной части). Это может потребовать много времени, увеличение глобальной CFM предотвращает множественные контакты и ограничивает высокие значения сил.
- Ограничивайте сгибания(hinge) пределами +/- pi

12. FAQ

Это раздел содержит часто задаваемые вопросы и ответы на них. Для более полной информации можете проверить сайт поддержки сообщества http://q12.org/cgi-bin/wiki.pl?ODE_Wiki_Area.

12.1. Как я могу присоединить тело к статическому окружению

Используя функцию `dJointAttach` аргументами `(body,0)` or `(0,body)`.

12.2. Нужна ли для использования ODE графическая библиотека X ?

Нет. ODE является вычислительным движком(computational engine) и он абсолютно не зависит от какой бы то ни было графической библиотеки. Тем не менее примеры, идущие вместе с ODE, используют OpenGL и гораздо интереснее использовать ODE вместе с какой ни будь графической библиотекой чтобы пользователь мог увидеть результат симуляции. Но это уже ваши проблемы.

12.3. Почему мои жесткие тела то отскакивают то проникают друг в друга при столкновении? Мой параметр восстановления первоначального состояния установлен в ноль!

Иногда, когда жесткие тела сталкиваются без параметра восстановления первоначального состояния(restitution), они сначала слегка проникают друг в друга, а затем отскакивают лишь слегка коснувшись друг друга. Проблема усугубляется при увеличении шага времени. Что же здесь происходит?

Контактное сочленение создается только после того как произошло определение столкновения тел. Если используется фиксированный шаг времени, то может случиться так что тела уже проникли друг в друга. Механизм уменьшения ошибки отталкивает тела друг от друга, но это может потребовать несколько шагов времени (в зависимости от значения ERP параметра).

Такое проникновение и отталкивание может выглядеть как отскок и абсолютно не зависеть от того установлен параметр восстановления первоначального состояния или нет.

В некоторых других симуляторах отдельным жестким телам можно задавать различные шаги времени для того чтобы удостовериться что тела не проникнут сильно друг в друга. Тем не менее в ODE используется фиксированный шаг времени, поэтому автоматический выбор шага времени при котором бы тела не проникали друг в друга является проблемой для любого симулятора жестких тел (входная ARB структура должна содержать шаг для первого проникновения, это может привести к очень маленьким шагам).

Вот три совета для решения этой проблемы:

- Использовать меньший шаг времени.
- Увеличить ERP чтобы сделать проблему менее заметной.
- Как ни будь использовать переменный шаг времени.

12.4. Как можно создать неподвижное тело?

Другими словами как создать тело, которое бы не двигалось, но взаимодействовало бы с другими телами? Ответ – создать только геометрию без привязке к объекту жесткое тело. Ассоциировать геометрию с жестким телом имеющим ID ноль. Когда при контакте двух геометрий, одного с ненулевым ID а другого с нулевым ID, произойдет вызов функции обратного вызова(callback function) определения столкновения вы можете просто передать эти ID в `dJointAttach` функцию как обычно. Будет создан контакт между жестким телом и статическим окружением.

Не пытайтесь добиться того же эффекта устанавливая очень большую массу/инерцию для “неподвижного” тела, а затем восстанавливая его позицию/ориентацию на каждом шаге времени. Это может привести к непредсказуемым ошибкам в симуляции.

12.5. Почему желательно устанавливать ERP значение меньше единицы?

Из определения ERP значения, можно заключить что 1 является лучшим значением, потому что все ошибки сочленений будут полностью исправляться на каждом шаге времени. Тем не менее ODE использует различные виды приближений при интегрировании, поэтому ERP=1 обычно не будет исправлять 100% ошибок в сочленениях. ERP=1 будет работать в некоторых случаях, но так же может вызвать нестабильность в некоторых системах. В этих случаях надо уменьшать ERP для достижения лучшего поведения системы.

12.6. Лучше задавать скорость тела напрямую вместо прилаживания силы или вращающего момента?

Вы должны задавать скорость тела напрямую только если вы устанавливаете систему в первоначальное состояние. Если вы устанавливаете скорость тела на каждом шаге времени (например на основании данных захвата движений) тем самым вы неправильно пользуетесь вашей физической моделью, т.е. принуждая систему делать так как хотите вы, а не позволяя вести себя естественно.

Предпочтительнее задавать скорости тела во время симуляции с помощью двигателей сочленения(joint motors). Желаемые значения скоростей могут быть установлены за один шаг времени, предусматривая что ограничения на силу/вращающий момент установлены достаточно большими.

12.7. Почему, когда к телу присоединены другие тела с помощью сочленений, оно набирает скорость медленнее при заданной напрямую скорости?

Это случает когда скорость устанавливается только для одного тела без задания скоростей для остальных сочлененных тел. Делая так вы вызываете ошибку в системе, которая сохраняется на протяжении некоторого количества шагов времени поскольку нарушаются расстояния в сочленениях. Механизм уменьшения ошибки исправит эту ошибку и тела потянутся друг за другом, но это займет несколько шагов времени и вызовет заметное “замедление движения”(drag) исходного тела.

Установка скорости будет действовать только на одно тело. Если тело сочленено с другими телами, то надо устанавливать скорости и этих тел для предотвращения неправильного поведения.

12.8. Должен ли я приводить единицы измерения к 1.0 ?

Предположим что вам надо имитировать некоторое поведение в масштабе нескольких миллиметров и нескольких грам. Такие маленькие длины и массы обычно хорошо работают в ODE не вызывая ни каких проблем. Тем не менее порой вы можете столкнуться с проблемами стабильности, которые возникают из-за недостатка в точности при разложении на множители. В этом случае вы можете попытаться привести длины и массы вашей системы к 0.1..10. Размер шага времени должен быть отмасштабирован соответствующим образом. Подобные действия необходимо проводить при использовании больших длин и масс.

В основном значения масс и длин, лежащих в диапазоне 0.1..10, лучше разлаживаются на множители и меньше теряется в точности вычислений. Подобные действия особенно полезны при использовании одинарной точности.

12.9. Я сделал машину, но колеса не хотят оставаться на своих местах!

Если вы делаете симулятор автомобиля, то обычно вы создаете тело рамы(chassis body) и присоединяете к нему четыре тела колес(wheel bodies). Тем не менее иногда можно обнаружить что колеса вращаются в противоположном направлении, каким то образом сочленение дает неправильный результат(ineffective). Проблема наблюдается когда машины движется быстро (соответственно колеса тоже вращаются быстро) и машина пытается повернуть в сторону. Колеса начинают вращаться не в том месте в котором они присоединены хотя ось изменила направление. Если колеса вращаются медленно или делается медленный поворот, проблема менее заметна.

Проблема заключается в многочисленных ошибках вызванных высокой скоростью вращения колес. Две функции призваны помочь решить эту проблему: `dBodySetFiniteRotationMode` и `dBodySetFiniteRotationAxis`. Тела колес должны иметь свои настройки режима ограничения вращения и ограничивающие вращение оси должны задаваться на каждом шаге времени, совпадая со своими осями сгиба. Этого должно быть достаточно для решения большинства проблем.

12.10. Как мне сделать “одностороннее” взаимодействие при столкновении

Предположим вам надо обработать взаимодействие двух тел (A и B). Движение тела A должно влиять на тело B, а тело B не должно оказывать никакого влияния на A. Такая необходимость может возникнуть если тело B является камерой, перемещаемой в Виртуальном Окружении. Камера должна взаимодействовать с объектами сцены так чтобы не проникать в них, но движение камеры не должно оказывать влияния на ход симуляции. Как этого достичь?

Вот хорошее решение: при определении столкновения не создавать контактное сочленение между A и B, как это делается обычно. Вместо этого присоединить контактное сочленение между B и 0 (статическим окружением). Таким образом тело A будет казаться телу B статическим и не перемещаемым. Таким образом тела A и B могут немного проникать в друг друга, но для большинства приложений это не является проблемой.

12.11. Windows версия ODE не работает с большими системами

В ODE при использовании `dWorldStep` требуется размер стека приблизительно равный $O(n)+O(m^2)$, где n количество тел m сумма всех степеней свободы сочленений(joint constraint dimensions). Если m велико, то требуется много памяти!

В Unix-подобных ОС обычно размер стека растет по мере необходимости, верхний предел может быть сотни мегабайт. Компиляторы Windows обычно выделяют стек гораздо меньшего размера. Если в результате ваших экспериментов большая система потерпела крах(crash), постарайтесь увеличить размер стека. Например в командной строке MS VC++ размер стека увеличивается с помощью следующего флага `/Stack:num`.

Другой путь использовать `dWorldQuickStep`.

12.12. Мои простые вращающиеся тела нестабильны!

Если у вас есть прямоугольный параллелепипед(box) с различными длинами сторон и вы вращаете его в пространстве, то вы будете наблюдать что он вращается с постоянной скоростью. Но иногда в ODE прямоугольный параллелепипед может сам по себе увеличивать скорость вращения вращаясь все быстрее и быстрее пока не произойдет “взрыв” (исчезновение в бесконечности). Вот объяснение:

ODE использует полу неявное интегрирование(semi-implicit integrator) первого порядка. “Полу неявное” значит что некоторые силы вычисляются так, как будто используется неявное интегрирование, а другие силы вычисляются так, как будто используется явное интегрирование. Силы соединения (прилагаемые к телам чтобы удержать их вместе) являются неявными, а “внешние” силы (прилагаемые пользователем и вызывающие эффект вращения) являются явными. Таким образом погрешности в неявном интегрировании проявляются в уменьшении энергии – другими словами интегрирование тормозит систему для вас. Погрешности в явном интегрировании производят противоположный эффект – увеличивают энергию системы. Вот почему система, имитируемая с помощью явного интегрирования первого порядка, может взрываться.

Итак, для отдельно вращающегося тела в пространстве, используется явное интегрирование. Если моменты инерции тела остаются одинаковыми (напр. для сферы) для всех осей вращения, то ошибки интегрирования будут незначительными. Если моменты инерции тела не одинаковы для осей вращения, кинетическая энергия(momentum) передается между различными направлениями вращения. Такое поведение корректно с точки зрения физики, но вызывает большие ошибки в интегрировании. Поскольку интегрирование явное, то из-за ошибок энергия увеличивается, что в свою очередь вызывает все возрастающую скорость вращения, ошибок становится все больше и больше – далее следует взрыв. Проблеме наиболее подвержены длинные тонкие объекты у которых 3 момента инерции сильно отличаются.

Для предотвращения таких проблем следует следовать одному или нескольким следующим правилам:

- Используйте симметричные свободно вращающиеся тела (т.е. у которых все моменты инерции равны – матрица инерции все время равна единичной матрице). Заметьте что вы можете отображать и производить столкновения с длинным тонким прямоугольным параллелепипедом и использовать матрицу инерции сферы.
- Убедитесь что свободно вращающиеся тело вращается не слишком быстро (т.е. не прилагайте больших вращающих моментов или обеспечивайте дополнительные тормозящие силы).
- Вводите дополнительные тормозящие элементы в окружение, и не используйте столкновения с отскоком, это может привести отражению энергии.
- Пользуйтесь меньшим шагом времени. Это плохо по двум причинам: это медленно и в ODE на текущий момент реализовано только интегрирование первого порядка, так что прибавка в точности будет минимальной.
- Используйте более высокий порядок интегрирования. Это еще не реализовано в ODE.

В будущем я могу добавить в ODE возможность динамически изменять вращение выбранных тел таким образом, чтобы ошибки интегрирования ODE не оказывали влияния.

12.13. Мои катящееся тела (напр. колеса) иногда застревают в геометрии

Рассмотрим систему в которой тела катятся по поверхности составленной из множества геометрических объектов. Например это может быть машина едущая по поверхности земли (катящимися телами(rolling bodies) будут колеса). Если вы обнаружили что катящиеся тела загадочным образом останавливаются при перемещении от одного геометрического объекта к другому, или когда получается несколько точек контакта, в этом случае

необходимо использовать другую модель трения в контакте. В этом разделе объясняется эта проблема и ее решение.

12.13.1. Проблема

Пример такой системы показан на рисунке 13, здесь показан мяч который только скатился с наклонной поверхности и коснулся земли.

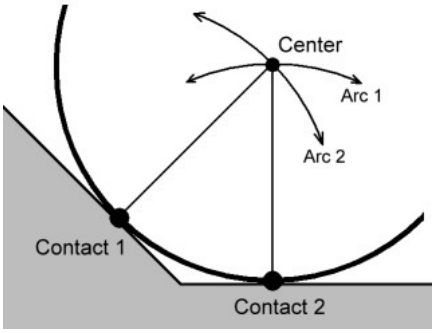

Рисунок 13: Проблема в контакте при качении.

Обычно мяч должен продолжать катиться по земле в правую сторону. Тем не менее если используется стандартная ODE модель трения в контакте - мяч остановится как только коснется земли. Почему?

ODE содержит два способа приближения трения: первый способ по умолчанию (называемый приближение константного – ограничения - силы(constant-force-limit approximation), или “прямоугольно параллелепипедное трение”(box friction)) и улучшенный способ (называемый “пирамидное приближение трения 1”(friction pyramid approximation 1)) который может быть получен с помощью установки `dContactApprox1` флага в поле контактного сочленения.

Рассмотрим картинку выше. Здесь присутствует две точки контакта, одна между мячом и наклонной поверхностью, другая между мячом и землей. Если в обоих контактах используется режим прямоугольно параллелепипедного трения и параметр `mu` установлен в `dInfinity`, то в этом случае мяч не может скользить по наклонной поверхностью или по земле.

Если скольжение в точке контакта мяча возможно, то центр мяча *должен* двигаться по дуге(arc) вокруг точки контакта. Таким образом центру мяча необходимо двигаться одновременно по “Дуге 1”(Arc 1) и “Дуге 2”(Arc 2). Единственным способом одновременного движения по двум путям является остановка мяча.

Это не ошибка в ODE – так что же здесь происходит? Объекты в реальной жизни не застревают в подобной ситуации. Проблема в том что в простом “прямоугольно параллелепипедном” приближении трения касательные силы, присутствующие в контакте, останавливают скольжение *независимо* от нормальных сил, препятствующих проникновению. Это не физика из реальной жизни, поэтому не удивляйтесь результату, который не соответствует реальной жизни.

Замете что проблема не проявляется если `mu` установлена в ноль, но это бесполезно, поскольку нам надо хоть какое то трение реального мира.

12.13.2. Решение

Решение состоит в использовании `dContactApprox1` флага в поле контактного сочленения и установки в `mu` подходящего значения между 0 и бесконечностью. Этот режим гарантирует присутствие касательной анти-скользящих(anti-slipping) силы если сила, направленная по нормали в точке контакта(contact normal force), не равна нулю. В примере приведенном высшее поворот приведет к тому что в контакте-1(contact-1) сила направленная по нормали будет равна нулю, т.е. в контакте-1 не будут прилагаться никакие силы, и проблема будет решена (мяч покатится дальше по земле).

Режим `dContactApprox1` годится не для всех ситуаций, вот почему он опционален. Важно помнить что хотя это и лучшее приближение трения, но оно не является истинным трением Coulomb. Таким образом возможно что вы еще встретите примеры с физически некорректным поведением.

13. Известный вопрос

- При задании массы жесткого тела, центр масс всегда находится в точке (0,0,0) относительно тела. Это ограничение происходит с самого начала ODE, поэтому теперь мы можем считать это “фишкой” :)

14. Внутри ODE

[сейчас просто заметки]

- Все 6x1 пространственных(spatial) скоростей и ускорений разделены на 3x1 компонентные вектора, которые хранятся непрерывно друг за другом в 4x1 виде.

- Коэффициент скорости Лагранжа(Lagrange multiplier velocity) основан на модели Trinkle и Stewart.
- Трение соответствует Baraff.

- Стабильность важнее точности.

- Разговор о различных методах возможен, а именно о том как в реальном времени соединения создают проблемы.

- Разложение на множители.

- Решение LCP.

- Уравнения движения.

- Модель трения и приближения.

Почему я не реализовал точную модель пирамидного трения(friction pyramid) или конусного трения(friction cone) (напр. версию Baraff ’a) ? Потому что

пришлось бы иметь дело с не симметричными (и возможно не определенными) матрицами для любого статического или динамического трения. Скорость считается более важной – текущей реализации трения необходимо только семетричное разложение на множители, что в два раза быстрее.

14.1. Соглашение о хранении матриц

Операции над матрицами, такие как разложение на множители, дороги, поэтому мы должны хранить данные таким образом чтобы с ними можно было удобно работать. Я хочу сделать позже 4-ех компонентную SIMD оптимизацию следующим образом: хранить матрицу по строкам, и каждая строка будет представлять структуру из четырех элементов. В неиспользуемые элементы в конце каждой строки/столбца должны быть записаны нули. Это называется “стандартный формат”. Надо надеяться что в будущем, с выходом все большего числа процессоров поддерживающих 4-ех компонентные SIMD’ы (особенно для быстрой 3D графики), такое решение принесет пользу.

Исключение составляют матрицы имеющие только одну строку или столбец (вектора), они всегда хранятся как строка, т.е. не добавляется ни каких дополнительных элементов за исключением недостающих элементов в конце строки.

Таким образом все 3x1 вектора с плавающей точкой хранятся как 4x1 вектора: (x,x,x,0).

14.2. Часто Задаваемые Вопросы по внутреннему устройству

14.2.1. Почему некоторые структуры имеют префикс `dx`, а некоторые префикс `d`?

Префикс `dx` используется внутренними структурами данных, которые не должны быть видны из вне. Префикс `d` используется структурами являющимися частью публичного интерфейса.

14.2.2. Возвращаемые вектора

Существует два способа возврата векторов ODE, а именно:

```
    const dReal* dBodyGetPosition (dxBodyID);
    void dWorldGetGravity (dxWorldID, dVector3);
```

Почему? Второй способ “официальный”. Первый способ возвращает указатель на изменяемые внутренние структуры данных и является менее правильным с точки зрения API. Для стабильности API я считаю что возвращать заполненный вектор вместо указателя правильней по двум причинам:

1. Возвращаемое значение может быть каким либо образом обработано, таким образом не будет внутреннего “кэша” на который можно было вернуть указатель.
2. Внутренние структуры данных могут быть перемещены, что может создать проблемы, если пользователь хранит указатель для того что воспользоваться им (указателем) позже.

Поскольку два этих случая не могут случится в ODE – большинство возвращаемых векторов кэшируются и всегда имеют один и тот же адрес. Но в будущем может быть полезно сделать по другому. В текущем API, из соображений скорости, (напр. получение трансформации тела) возвращается указатель – тем самым нарушая мое собственное правило.

Данный перевод осуществил Татаринцев Денис Юрьевич(asmzx): asmzx@tut.by
14.10.2004
Исправлен: 19.10.2004