

Дэвид А. Д. Гоулд

MAYA

Полное руководство
по программированию

Подробное описание
языка MEL и C++ API

КУДИЦ-ОБРАЗ

COMPLETE MAYA PROGRAMMING

An Extensive Guide to MEL and the C++ API

David A. D. Gould



MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER SCIENCE

www.mkp.com

Моему отцу, привившему любовь к письму

Сергей Есенин

Софья Есенина

Софья Есенина

Софья Есенина

Софья Есенина

Аэвил А. А. Гоулд

Полное руководство по программированию Maya

Подробное описание языка MEL и интерфейса
C++ API

Перевод с английского

КУДИЦ-ОБРАЗ
Москва • 2004

Гоулд Дэвид А. Д.

Полное руководство по программированию Maya. Подробное описание языка MEL и интерфейса C++ API / Пер. с англ. - М.: КУДИЦ-ОБРАЗ, 2004. - 528 с.

Книга содержит всестороннее описание возможностей программирования в системе Maya, являющейся лидером на рынке приложений для работы с трехмерной компьютерной графикой. Основное внимание уделено в ней встроенному языку программирования MEL (Maya Embedded Language) и интерфейсу прикладного программирования (API) на языке C++. Вместе с тем, материал книги начинается с описания базовых принципов функционирования Maya, что позволяет получить полное представление о внутреннем устройстве системы. Эти знания помогут читателю быстро и эффективно приобрести опыт написания собственных программ.

Предлагаемое автором описание языка сценариев MEL стало глубоко и основательно, что может с полным правом считаться настоящим учебником по этому языку, достойным самого пристального внимания со стороны начинающих пользователей. Простота MEL и сходство его синтаксиса с языком C++ способствуют его легкому освоению даже теми, кто делает свои первые шаги в освоении пакета Maya.

Детальное описание методик создания подключаемых модулей, нестандартных команд и узлов с использованием C++ API превращает эту книгу в уникальное по своей лаконичности и четкости изложения пособие по расширению возможностей Maya. Каждый демонстрируемый прием автор сопровождает реальным практическим примером и подробно анализирует требуемый исходный код.

Рекомендуется как начинающим пользователям Maya, так и подготовленным специалистам, желающим расширить свои знания по организации системы и приобрести умения и навыки программирования для нее.

ISBN 1-55860-835-4

ISBN 5-93378-098-7

Гоулд Дэвид А. Д.

**Полное руководство по программированию Maya. Подробное описание языка
MEL и интерфейса C++ API**

Учебно-справочное издание

Перевод с англ. А. В. Петров

Лицензия ЛР № 071806 от 02.03.99. НОУ «ОЦ КУДИЦ-ОБРАЗ».

119034. Москва. Гагаринский пер., д. 21, стр. 1. Тел.: 333-82-11, ok@kudits.ru

Подписано в печать 24.05.04

Формат 70×90/16. Бум. газетная. Печать офсетная.

Усл. печ. л. 38,6. Тираж 2000. Заказ 1210

ISBN 1-55860-835-4

ISBN 5-93378-098-7

© НОУ «ОЦ КУДИЦ-ОБРАЗ», 2004

Издатель согласен, что перевод Произведения должен быть сделан верно и точно и никакие изменения не будут сделаны в тексте без письменного согласия Правообладателя.

Издатель согласен напечатать оригинальное английское название Произведения и имя Автора на титульной странице или обратной стороне титульной страницы каждого экземпляра своего издания, принять все необходимые меры для защиты авторских прав своего издания и воспроизвести следующее уведомление об авторских правах как на английском, так и на русском языке в каждом опубликованном экземпляре:

Copyright © 2002 by Academic Press

Translation Copyright © 2004

by «КУДИЦ-ОБРАЗ»

All rights reserved.

Издатель согласен напечатать на корешке и титульной странице каждой книги имя и логотип Morgan Kaufmann Publisher.

Отзывы критики на «Полное руководство по программированию Maya»

Дэвид Гоулд - эксперт в области применения Maya, программирования в этой системе и обучения работе с ней, и этого нельзя не заметить. Читатели, которым необходимо составлять программы для Maya, поймут, что эта книга им просто необходима. Ее должны прочитать даже те пользователи пакета, кто не намерен заниматься программированием всерьез, так как это чтение позволит лучше понять происходящее в самой Maya. Лаконичное, но при этом подробное изложение включает описание языка MEL и интерфейса C++ API и представляет интерес как для начинающих, так и для опытных программистов. Очень рекомендую!

Ларри Гритц (Larry Gritz), Exluna/NVIDIA

Эту книгу должны прочитать все, кто программирует в среде Maya, - и новички, и специалисты. Новичкам эта книга послужит полным и удивительно продуманным практическим руководством и введением в систему. Самая большая заслуга Дэвида все-таки в том, что он делится с читателями своими глубокими знаниями фундаментальных основ и архитектуры пакета, что позволяет даже специалистам в этой сфере научиться более эффективному применению богатых возможностей интерфейсов программирования Maya.

Филип Шнайдер (Philip J. Schneider), Disney Feature Animation

Написав рецензию на книгу Дэвида Гоулда *Complete Maya Programming*, я должен сказать, что эта книга является самым полным руководством по разработке сценариев и подключаемых модулей для Maya. Никогда прежде я не встречал такого краткого и ясно написанного пособия по программированию Maya. Любой пользователь, который отважится прочесть эту книгу, безусловно, извлечет из этого свою выгоду.

Крис Рок (Chris Rock), технический директор Large Animation Studio, Северная Калифорния

Если вам когда-нибудь хотелось раскрыть для себя панель инструментов Maya, эта книга – для вас. Получив четкие инструкции, описывающие каждый шаг вашей работы, вы совсем скоро научитесь настраивать и совершенствовать приложение, а также создавать свои собственные расширения, пользуясь для этого языком сценариев MEL или всеми возможностями C++ API.

Кристоф Эри (Christophe Hery), Industrial Light + Magic

Наконец, эта книга представляет собой содержащее пошаговые инструкции практическое руководство, которое, начиная с самых азов, демонстрирует приемы максимально эффективного применения Maya. Читатели «Полного руководства по программированию Maya» получат, прежде всего, полное представление о внутренней работе системы, а затем перейдут к изучению способов настройки и расширения Maya посредством сценариев и подключаемых модулей, позволяющих достичь нового уровня управляемости и эффективности.

Пользователи, не имевшие ранее опыта программирования, могут обратиться к простому языку сценариев Maya, MEL (Maya Embedded Language), а более подготовленные специалисты - работать с интерфейсом C++ API (Application Programming Interface). Сочетая в себе лучшие стороны фундаментального пособия для начинающих и подробного справочника для опытных разработчиков, «Полное руководство по программированию Maya» является настольной книгой каждого пользователя, стремящегося к совершенному овладению этим пакетом.

Что делает эту книгу уникальной?

Демонстрация приемов использования MEL для управления Maya, настройки ее интерфейса, автоматизации процедур и т. д.

Подробности применения C++ API для модификации функций Maya и разработки инструментов и средств, удовлетворяющих любым потребностям,

Наличие большого числа реальных примеров, иллюстрирующих практические аспекты программирования Maya.

Наличие множества сценариев на языке MEL, исходных текстов на языке C++, перечня ресурсов и словаря, доступных по адресу www.davidgould.com.

Исходные тексты

Обратите внимание на то, что все файлы, используемые в этой книге, доступны по адресу www.davidgould.com. Перед вами, хотя и неполный, перечень информации, доступной на этом сайте:

- ◆ исходный код сценариев на языке MEL и программ на языке C++ для всех примеров из этой книги;
- ◆ дополнительные примеры сценариев на языке MEL;
- ◆ дополнительные примеры исходного кода с использованием C++ API;
- * список допущенных в книге опечаток;
- * постоянно обновляемый словарь терминов;
- ◆ обновляемый список других Web-сайтов и сетевых ресурсов по данной тематике.

Предисловие

Пакет Maya - это, **безусловно**, очень мощный инструмент компьютерной графики. Но нельзя не сказать и о том, что его изучение подчас отпугивает. Только то, что его возможности чрезвычайно **широки**, делает трудноразрешимой проблемой даже изучение **пакета**, не говоря уже об овладении им на профессиональном уровне. Возможно, это никогда не было для вас столь очевидно, как в тот день, когда вы открыли его в первый раз. Я уверен, у вас возникло **ощущение**, будто вы стоите у подножья горы и готовы начать длительное **восхождение**. Вы будете рады услышать о том, что это путешествие может стать легче. Вы можете настраивать и расширять Maya так, как не могли никогда прежде. Вы можете автоматизировать или в немалой степени упрощать великое множество своих **повседневных** задач. Вы можете создавать инструменты, которые не только повысят вашу производительность, но и дадут **вам** гораздо большие возможностей управления. Всего этого и многоного другого можно достичь благодаря программированию в среде Maya.

На большинство людей даже простое упоминание программирования наводит страх и волнение. Оно и понятно: авторы многих книг по программированию всерьез опираются на имеющийся у **читателей** программистский опыт. Эта книга написана с учетом того, что вам, вероятно, еще не пришлось приобрести какой бы то ни было опыт, и призвана рассеять миф о том, что программирувать в Maya способны лишь профессиональные программисты. Прочно усвоив основные идеи, любой желающий сможет приступить к работе с системой на уровне, доступном лишь при использовании программирования. То, что когда-то казалось вам пугающим предприятием, предстанет перед вами как опыт, дающий большие возможности и открывающий широкие горизонты.

Под руководством опытного и терпеливого проводника любое путешествие становится не таким трудным. Роль этой книги, по сути, не **назидать**, а вежливо подсказать вам путь к пониманию того, как, в основе своей, работает Maya. Начав изучение с рассмотрения самого «сердца» **Maya**, вы узнаете, как выполняется управление данными и их обработка. Знание этих механизмов совершенно необходимо, так как они представляют собой основу, на которой построена вся работа пакета. Используя реализованный в Maya простейший язык **программи-**

рования MEL (Maya Embedded Language), вы приступите к изучению способов управления системой Maya и автоматизации множества операций. Затем познакомитесь с интерфейсом прикладного программирования (API) на языке C++. Имея базовые знания языка программирования C++, вы быстро научитесь разрабатывать собственные функции и инструменты. Изучая каждый новый реальный пример, вы все лучше будете **понимать**, как осуществляется доступ к функциям Maya и управление ими. Сочетая использование MEL и C++, вы вскоре сможете получить полный контроль над всеми аспектами деятельности Maya и способность расширять приложение ради удовлетворения любых собственных нужд.

Наряду с рассмотрением конкретных вопросов программирования в Maya, важное место в этой книге отведено тому, **почему** тот или иной элемент системы спроектирован именно так, а не иначе. Программирование зачастую позволяет решить задачу неограниченным числом разных **способов**. Важно, однако **понять**, что Maya обладает своей собственной, особой философией проектирования. В этой книге вам будет предложен ряд указаний по созданию программ таким образом, чтобы они прозрачно встраивались в Maya и работали в ней без каких-либо конфликтов. Осознавая причины выбора того или иного подхода, вы сможете перенести заложенную в него идею на решение своих собственных задач. Моя цель состоит в том, чтобы по окончании чтения этой книги вы могли не только вдохновенно рисовать в своем воображении **новые** волнующие перспективы, но и обрели необходимые знания и умения для их реального воплощения.

Об авторе

Дэвид Гоулд (David Gould), специалист с более чем десятилетним опытом работы в индустрии компьютерной графики, неотступно следует по пути художника и программиста. Его редкостная способность сочетать в себе техническое и творческое начало принесла ему множество наград и помогла добиться известности. Он играл ключевую роль в разработке уникального набора технологий, куда входит удостоенная приза лазерная система рендеринга для проекта Pangolin. Также он являлся создателем управляющего программного обеспечения для проектов Kuper и Monkey. Лично разрабатывал Illustrate! - ведущий инструмент на рынке средств рендеринга мультфильмов и технических иллюстраций. Этот продукт нашел свое применение, в том числе, в агентстве НАСА, компаниях British Aerospace, Walt Disney Imagineering и Sony Pictures Entertainment.

Карьера Дэвида охватывает великое множество компаний и континентов. В Париже он руководил производством научных трехмерных стереофильмов, в том числе получившего приз фильма *Inside the Cell*. В Лондоне разрабатывал запатентованную систему лицевой анимации. Продолжая обогащать свой опыт, он работал в Нью-Йорке, решая задачи в области видеопроизводства, где внес личный вклад в целый ряд известных коммерческих проектов.

Работая в компании Walt Disney Feature Animation в Лос-Анджелесе, Дэвид разрабатывал перспективную технологию анимации и моделирования, предназначенню для использования при создании анимационных художественных фильмов этой же студии. Далее, расширяя свою сферу деятельности, он перешел в компанию по производству программных средств Exluna в Беркли, основанную группой бывших исследователей-графиков из Pixar, в которую входил Ларри Гритц. Работая в Exluna, он принимал активное участие в проектировании и разработке совместимой с Renderman системы рендеринга Entropy, а также ряда других продуктов. Свою карьеру разработчика средств визуализации Дэвид продолжил в компании NVIDIA в Санта-Кларе (Калифорния), где оказывал помощь в создании будущих микросхем 3D-графики этой фирмы. Затем местом его работы стала компания Weta Digital (Новая Зеландия), в составе которой Дэвид трудится над фильмом-трилогией *The Lord of the Rings* («Властелин колец»). Его разностороннее участие в производстве, включая разработку шейдеров, работу со светом и создание компьютерных эффектов, отражает всю широту его таланта.

1. Введение

Говоря об изменениях в нашем мире, можно смело сказать, что ни одна сфера человеческой деятельности не развивается столь динамично, как индустрия компьютерной графики. Эта по-прежнему сравнительно молодая отрасль уже сумела проникнуть в самые различные области. Среди них моделирование, анимация и визуализация. Они не стоят на месте, и развитие каждой из них – путь совершенствования и прогресса. В результате сегодня мы наблюдаем такие потрясающие реалистичные и правдоподобные изображения, каких не видели никогда прежде. И никогда прежде научные имитационные эксперименты не отличались такой сложностью. И никогда прежде не было таких интерактивных игр с эффектом погружения, как сейчас. Никогда прежде, т. е. до наступления завтрашнего дня!

Если в индустрии компьютерной графики в чем-то и можно быть уверенными, то лишь в том, что изменения не просто неотвратимы, они продолжаются, и их интенсивность все увеличивается. Устойчивый рост прошлых лет меркнет в сравнении с нынешней скоростью появления и воплощения в жизнь новых изобретений и идей. Движущая сила этого прогресса - ненасытное желание людей получать более сложные изображения и испытывать более сильные ощущения. Это желание беспредельно, и индустрия компьютерной графики постоянно торопится, пребывая в погоне за «максимальными» переживаниями.

На переднем крае этой непрерывной волны изменений всегда находилась компания Alias | Wavefront. Ее приверженность стратегии, нацеленной на долгосрочные исследования и разработки, позволила создать ряд самых современных инструментов и технологий из числа тех, что появились на сегодняшний день. Одна из таких технологий - Maya. С момента выпуска в 1998 г, Maya стала тем пакетом компьютерной графики, на котором останавливают свой выбор ведущие анимационные студии и компании мира. Пакет Maya, обладающий обширным набором инструментов моделирования, динамики, анимации и рендеринга, находит применение на всех этапах создания видеопродукции.

Чтобы не отставать от высокого темпа изменений, Maya постоянно обновляется и совершенствуется. Даже несмотря на то что пакет уже обладает большими возможностями, авторы Maya осознали, что никогда не смогут предложить всех

средств и функций, которые будут нужны каждому. Они не смогли бы спроектировать их с учетом того, как любит работать тот или иной клиент. Поэтому они построили свой продукт так, чтобы пользователи могли свободно настраивать и расширять его в соответствии со своими потребностями. Все те, кто применяет Maya, вправе свободно изменять **пакет**, адаптируя его к нуждам своей среды и технологии производства. Начиная со смены основного графического интерфейса пользователя и заканчивая «бесшовным» внедрением новых возможностей, пользователи могут свободно превращать Maya в нечто совершенно иное.

Фактически пакет Maya можно считать открытой архитектурой для работы с компьютерной графикой. Воспринимайте Maya как **структуру**, предоставляющую множество трехмерных компонентов (**объектов**, анимации, динамики и т. д.), которые вы можете применять в своих собственных приложениях. К примеру, вы можете использовать Maya для создания собственных имитационных моделей или решения интерактивных задач в реальном масштабе времени. Хороший пример, который демонстрирует возможности **Maya**, показал инженер Alias | Wavefront Майк Тейлор (Mike Taylor). Благодаря применению простых сценариев, он сумел превратить Maya в трехмерный **тетрис**. Окончательный вариант интерфейса и способа взаимодействия с пользователем настолько **отличался** от первоначального, что было трудно поверить, действительно ли это Maya.

Ближе познакомившись с тем, до каких **пределов** простираются **возможности** настройки и расширения Maya, вы поймете, что ваши действия почти ничем не **ограничены**. Гибкость предлагаемого продукта состязается только с вашим воображением.

1.1. Возможности программирования Maya

Итак, что же именно можно запрограммировать в Maya? Этот раздел посвящен некоторым основным функциям Maya, к которым можно обращаться напрямую и которыми можно управлять при помощи интерфейсов программирования.

1.1.1. Настройка

Графический пользовательский интерфейс (GUI) Maya управляется средствами MEL - встроенного языка, на котором он полностью и написан. Язык MEL помогает создавать, редактировать и удалять любые элементы графического интерфейса пользователя. А значит, при помощи MEL вы тоже **можете** управлять интерфейсом Maya. Так, пользуясь своими сценариями на языке **MEL**, вы можете полностью заменить стандартный интерфейс. Часто возникает потребность в специа-

лизированной настройке фрагментов интерфейса Maya. К примеру, возможно, вы захотите разработать собственный интерфейс, позволяющий аниматорам устанавливать ключевые кадры, не утруждая себя изучением средства Channel Box (Окно каналов) или редактора Graph Editor (Графический редактор). Кроме того, вы можете скрыть или убрать многие элементы интерфейса Maya, понизив его сложность для конкретных пользователей.

Наряду с пользовательским интерфейсом, вы можете управлять внутренними настройками Maya. MEL позволяет вносить изменения в настройки Maya, действующие как в рамках одного проекта, так и на систему в целом. Например, вы можете обеспечить согласованное применение всеми пользователями одинаковых временных, угловых и линейных единиц измерения. В иной ситуации каждый пользователь, открывающий тот или иной проект, сможет работать с собственными настройками, имеющими силу в данный момент времени.

1.1.2. Интеграция

Часто Maya оказывается не единственным пакетом, применяемым в производственном процессе. При этом могут использоваться лишь некоторые возможности Maya, для решения же отдельных задач могут служить другие пакеты. В этом случае возникает необходимость передавать данные из внешних пакетов в Maya и обратно. Хотя в стандартную поставку Maya входят средства экспорта и импорта отдельных форматов, интерфейс программирования позволяет вам писать собственные модули экспорт и импорта данных. Такие модули транслируют данные одного пакета в формат, понятный другому, поэтому их также называют трансляторами. Так как Maya позволяет обращаться к целым сценам и связанным с ними данным, то вы можете выводить эти данные в любой нужной вам форме. Перед компаниями, выпускающими компьютерные игры, часто стоит задача переноса и преобразования разработок, сделанных в Maya (моделей, анимации и т. д.), в формат конкретной игровой платформы.

Кроме того, на основе функциональных возможностей Maya можно создать отдельное приложение. Так появляется возможность использовать смесь кода Maya и собственных функций для создания законченного программного решения.

1.1.3. Автоматизация

Нередко многие задачи повторяются. Для автоматизации повторяющихся действий вполне подходит программирование. Вместо того чтобы просить пользователя всякий раз решать задачу вручную, можно составить сценарий на языке MEL, который ее полностью автоматизирует. Будь то раскраска, размещение объектов или просто какая-то задача, требующая многократного решения, - все

это можно автоматизировать. По сути, в силу характерных для программирования обобщений, вы можете написать **сценарий**, выполняющий различные действия в зависимости от текущего контекста. Например, можно создать ряд объектов и поместить их поверх другого объекта либо прикрепить к его нижней **части**, причем автоматически, не требуя от пользователя ручной установки каждого из них.

1.1.4. Расширения

Несмотря на то что невероятно обширный набор возможностей Maya представляет предмет гордости этого пакета, потребность в нестандартных средствах и возможностях будет оставаться всегда. К **счастью**, Maya позволяет вам добавлять свои собственные возможности и функции. Более того, эти функции можно заставить прозрачным образом взаимодействовать с остальными компонентами Maya. С точки зрения пользователей, **средства**, которые они **создают**, ничем не отличаются от стандартных **средств** Maya. С помощью MEL и C++ вы можете создавать большие и сложные расширения пакета.

Существует лишь несколько ограничений, определяющих набор расширяемых функций. Maya позволяет создавать собственные **щедеры**, нестандартные модули динамики, частицы, деформаторы и контроллеры анимации, и это **далеко** не все. Фактически создаваемые расширения могут напрямую внедряться в граф зависимости **Dependency Graph**, самое «сердце» Maya. Встроив свое расширение один раз, в дальнейшем вы сможете обращаться к **нему** так же, как и к стандартным возможностям пакета.

1.2. Интерфейсы программирования

Художники обычно работают в Maya, пользуясь **графическим интерфейсом** пользователя. Он включает меню, диалоговые **окна**, кнопки и прочие **элементы**, предоставляющие пользователям наглядные средства **выполнения** тех или иных действий. Но есть и другой способ их **выполнения** действий. Возможности программирования Maya позволяют решать те же задачи, составляя и выполняя программы. Эти программы можно **разрабатывать**, пользуясь одним из двух интерфейсов программирования Maya: MEL или C++.

1.2.1. MEL

MEL - это акроним, который служит для обозначения встроенного языка Maya (Maya Embedded Language). Язык MEL - оригинальный язык программирования, **специально** созданный для работы в среде Maya. Благодаря упрощенной структуре и синтаксису, он более прост и используется более широко, нежели

интерфейс программирования на основе C++. Одно из **главных** достоинств MEL заключается в том, что он является **интерпретируемым языком**. В то время как обычные языки программирования требуют компиляции и сборки исходного кода, программа на интерпретируемом языке может выполняться сразу же. Эта способность немедленно выполнять записанные **инструкции** означает, что **MEL** особенно подходит для быстрого составления прототипов. Коль скоро шаг компиляции-сборки не требуется, вам будет проще создать проект и реализовать новую идею. Действительно, код на языке **MEL** можно написать, отладить и выполнить, не покидая Maya. Внешние компиляторы и отладчики становятся не нужны.

MEL - это интерпретируемый язык, поэтому у него есть недостаток: программа на нем **может** работать гораздо медленнее, чем аналогичная программа на C++. В результате компиляции исходного кода на C++ генерируются **естественные** машинные инструкции, и такая программа выполняется очень быстро. Интерпретируемый язык работает с исходным кодом, который интерпретируется «на лету». Когда Maya встречает инструкцию **MEL**, **пакет** должен интерпретировать ее и, **наконец**, преобразовать в естественный машинный формат. Даже несмотря на то что Maya проделывает большую работу, пытаясь ускорить этот процесс, зачастую MEL сильно отстает от C++ в отношении скорости. Однако во многих случаях дополнительные **преимущества**, **связанные** с быстрым созданием и **выполнением** программы на **MEL**, на деле **перевешивают** необходимость установки, сложность и издержки компиляции программы на C++. Все зависит от типа и сложности задачи, которую вы хотите решить.

1.2.2. C++

Для программирования Maya можно использовать **стандартный** язык C++. Хотя это и отпугивает тех, кто не знает этого языка, на сегодняшний день C++ является самым мощным инструментом, расширяющим возможности Maya. Используя C++, вы сможете создавать собственные подключаемые модули Maya, которые будут без проблем работать с остальными **компонентами** этого пакета.

Доступ к средствам программирования **осуществляется** через интерфейс прикладного программирования (API) на языке C++. Он состоит из ряда библиотек классов C++. Чтобы создать подключаемый модуль, просто напишите на языке C++ программу, которая использует и расширяет возможности базовых классов Maya. Как таковой процесс изучения программирования с использованием C++ API включает освоение назначения различных классов и приемов работы с ними. Реализация классов, к счастью, отличается **последовательностью** и **непротиворечивостью**, поэтому изучение некоторых более простых классов помогает

при изучении более **сложных**. Хотя количество классов может **поначалу** испугать, типичный подключаемый модуль Maya пользуется лишь малой их частью. Вообще говоря, постоянно используются лишь около трети существующих классов. Некоторые классы, известные лишь посвященным, применяются достаточно редко.

1.2.3. MEL или C++?

Теперь вы знаете о двух способах программирования работы Maya и должны сами решить, каким из них пользоваться. Делая свой выбор в пользу того или иного интерфейса программирования, в равной степени взвесьте оба варианта с позиции ваших конкретных нужд. На окончательный выбор могут повлиять такие внешние факторы, как сжатые сроки или особые требования к **производительности**. В общем, MEL содержит все необходимые функции. Он обладает широкими возможностями доступа к функциональности Maya и не требует обращения к C++. Работа с интерфейсом программирования на основе C++ обычно обусловлена потребностью в тех функциях, которые отсутствуют в интерфейсе MEL.

Кроме того, **немаловажно** понять, что выбор одного автоматически не исключает другого. Интерфейс C++ не является надмножеством интерфейса MEL; иначе говоря, интерфейс C++ не имеет всех возможностей, которые имеются в интерфейсе MEL, и тем более не превышает их. В **MEL** вы сможете отыскать ряд возможностей, которые недоступны через интерфейс C++, и наоборот. По существу, некоторые задачи можно решить лишь при совместном использовании обоих интерфейсов. Когда вы приобретете более богатый опыт разработки для **Maya**, хорошее знание API на основе C++ и MEL станет для вас **лучшим** средством выбора надлежащего решения конкретной проблемы. Уверенно ориентируясь в обоих интерфейсах программирования, вы сможете понять, когда один из них более соответствует тому или иному фрагменту задачи, и, соответственно, сможете получить оптимальное смешанное решение.

Простота применения

Выбор используемого интерфейса программирования может быть продиктован исключительно вашей программистской квалификацией. Хотя MEL обладает значительным сходством с языком C, он существенно отличается от него в тех деталях, которые делают его доступнее для менее опытных разработчиков. Благодаря изъятию таких элементов, как указатели, а также отказу от **выделения** и освобождения памяти, MEL становится гораздо проще в применении и понимании. Несмотря на то что он не предусматривает возможностей низкоуровневого доступа, предлагаемых языком C, необходимость этого при программировании Maya возникает нечасто. Кроме того, **отсутствие** подобных конструкций

также предотвращает некоторые наиболее распространенные причины программных сбоев и нестабильности общего характера.

Сценарии **MEL** более близки к *псевдокоду*, чем программы на других языках, поэтому их легче читать. Кроме того, MEL более «снисходителен» в том смысле, что не выполняет слишком много проверок типов. Это значит, что вы можете быстро составлять *сценарии*, не беспокоясь о *том*, какой тип имеют используемые переменные. Это может быть одновременно и благом, и мучением, поскольку иногда ведет к незаметным ошибкам, *которые*, как окажется позднее, трудно найти. Проверки типов, к счастью, можно ужесточить, явно указывая тип переменной при ее описании.

Если вы имеете немалый опыт программирования на **C++**, то, *возможно*, вам захочется воспользоваться интерфейсом на C++. Иерархия C++-классов Maya обеспечивает доступ к большому количеству функций пакета. Однако даже в том случае, если вы захотите написать весь код на *языке* C++, вам неизбежно придется время от времени заниматься программированием на MEL. Хорошее знание C++, безусловно, поможет в изучении MEL, так как синтаксис последнего очень напоминает С. Существенные различия между языками программирования С и MEL изложены в Приложении В.

Если *задача*, которую вы *программируете*, требует применения сложных структур данных, вам, скорее всего, потребуется использовать C++ API. Язык MEL содержит ограниченный набор различных типов переменных и не позволяет пользователям описывать собственные структуры данных.

Функциональные особенности

Поскольку интерфейс на основе C++ сложнее по сравнению с языком MEL, нетрудно предположить, что именно C++ API должен *обеспечивать* больше возможностей доступа и управления. Отчасти это так. Пользуясь интерфейсом на основе C++, можно создавать подключаемые модули и писать программы, обеспечивающие более тесную интеграцию с ядром Maya. Но все же неверно было бы полагать, что C++ API является надмножеством функций языка **MEL**. На самом деле, они дополняют друг друга.

Существует определенный набор функций, к которым можно обратиться из MEL, но нельзя обратиться из C++ API, и наоборот. Так что при выполнении ряда задач разработчик не имеет выбора и должен сочетать использование обоих языков. В действительности подобная практика весьма распространена и вы вполне можете встретить подключаемый модуль на C++, в котором иногда используются вызовы команд MEL. Однако чаще MEL обеспечивает достаточную функциональность, и написание модулей на C++ становится ненужным.

Но такое не всегда возможно. Интерфейс на основе C++ обладает некоторыми функциями, которые просто не имеют аналогов в арсенале MEL. Только посредством C++ API вы можете создавать свои собственные узлы. Создание подобных узлов позволяет разрабатывать нестандартные инструменты, деформаторы, формы, шейдеры, локаторы, манипуляторы и т. д. К тому же MEL не позволяет описывать пользовательские команды, хотя и поддерживает создание процедур, действие которых подобно действию некоторых команд.

Независимо от того, какой интерфейс программирования вы выберете для разработки ядра своих функций, вам придется использовать MEL, когда вы приступите к работе с любым графическим интерфейсом пользователя. поскольку MEL - это единственное средство управления интерфейсом. К счастью, команды MEL можно выполнять из интерфейса C++ API.

Межплатформенная переносимость

Учитывая широкое распространение Maya на многих вычислительных платформах, важно решить, имеет ли значение возможность переноса программы. Если ваша разработка ограничена одной платформой, это становится не столь существенным. Однако если не забывать о повсеместном использовании смешанных платформ для обработки данных и рендеринга, то, пожалуй, было бы разумным подумать о возможных последствиях решения, принятого сегодня.

Язык MEL является интерпретируемым, поэтому он был разработан с расчетом на легкую переносимость на другие платформы. В него входит немного функций, действие которых зависит от аппаратной конфигурации. По сути, почти все команды MEL работают независимо от конкретной платформы, на которой они запущены. Значит, разместить сценарий MEL на нескольких платформах обычно проще, так как он менее зависим от особенностей каждой из них. Для реализации всех функций графического интерфейса пользователя служит язык MEL, стало быть, вам никогда не придется иметь дело со своеобразием различных систем управления окнами. При помощи MEL вы можете разработать интерфейс на одной из платформ и знать о том, что на других он будет выглядеть и работать аналогично.

Хорошо известно, что разработка межплатформенных программных средств на C++ представляет трудности по целому ряду причин. Проблемы начинаются с того, что различные компиляторы C++ обладают собственными несовместимыми элементами. К тому же они в различной степени соответствуют стандарту ANSI C++. Наконец, не все компиляторы используют одни и те же стандартные библиотеки С. Применение только C++-классов Maya и библиотек, которые слабо зависят от других внешних наборов функций и расширенных свойств языка, должно облегчить разработку продуктов для нескольких платформ.

В тех случаях, когда межплатформенная совместимость играет немаловажную роль, попытайтесь как можно больше функций написать на языке MEL. Обращайтесь к C++ лишь тогда, когда это абсолютно необходимо.

Скорость

MEL - это интерпретируемый язык, и по этой причине он, вероятно, работает медленнее, чем C++. Однако это не всегда так, и скорость во многом зависит от сложности программы. При решении некоторых задач различие в скорости может быть не столь значительным, чтобы стать оправданием дополнительных усилий по написанию кода на C++. Что касается сложных операций, время выполнения которых действительно имеет решающее значение, то для них, безусловно, справедливо высказывание о том, что программа, на C++ выполняется быстрее. На практике прирост скорости может быть десятикратным. Это немаловажное соображение, если сцена состоит из больших и сложных геометрических объектов. Затратив дополнительные силы на программирование на C++, можно получить большой выигрыш в качестве взаимодействия с пользователем и общей производительности труда.

Разглашение информации

Может случиться так, что вы захотите передать некоторые свои функции третьей стороне, не раскрывая в точности деталей их реализации. Часто это означает, что сценарий или модуль содержит некоторые запатентованные алгоритмы или технологии. Вам бы хотелось защитить эту информацию, но в то же время позволить другим использовать свои наработки.

Так как сценарий на языке MEL - это, с одной стороны, исходный код, а с другой - инструкции, которые должны быть выполнены, вам не удастся отделить одно от другого. Чтобы пользователи могли запустить сценарий, они должны получить его текст на языке MEL. А когда у них будет текст, они смогут узнать, как он работает, открыв его в любом текстовом редакторе. В то же время подключаемый модуль на языке C++ представлен в машинном коде. Он является конечным результатом компиляции и сборки. Машинное представление окончательного продукта скрывает от пользователя подробности составления программы. Оно служит эффективной защитой методов и алгоритмов, использованных при написании программы.

Итак, если вы действительно против того, чтобы пользователи ваших сценариев на языке MEL смогли точно узнать о том, как они работают, лучше всего реализовать функции в виде подключаемого модуля на C++, содержащего немногочисленные дополнительные сценарии. Обратите внимание, что ваш модуль на языке C++ может по-прежнему вызывать команды MEL.

2. Основы организации Maya

Надежность любой постройки заложена в ее фундаменте. Это утверждение в равной степени применимо и к программированию Maya. Прочное усвоение основ организации пакета немало облегчит вашу работу в качестве программиста. Вам предстоит работать в среде Maya, поэтому так важно понять принципы ее функционирования. Любая система накладывает ряд ограничений: что в ней делать можно и чего нельзя. Несмотря на то что Maya – это, несомненно, чрезвычайно гибкий инструмент, помогающий достичь очень многого, существует ряд основополагающих правил, которые нужно всего лишь выучить, а затем придерживаться их.

Хотя вы, быть может, торопитесь и хотите, пропустив эту главу, сразу же приступить к программированию, я должен предостеречь вас от этого. Мне доводилось встречатьсяся со многими разработчиками в среде Maya, которые делали свою работу, однако годами применяя программу, попросту игнорировали или не знали основ ее функционирования. В результате их приложения не могли в полной мере использовать всю мощь пакета. Неоднократно они пытались приспособить свой подход к существующей идеологии Maya. К сожалению, при таком отношении к делу они неизменно терпели поражение. Ознакомившись с тем небольшим объемом материала, который приведен в этой главе, вы сможете овладеть некоторыми основными правилами Maya и, благодаря этому, создавать более быстрые приложения более высокого качества.

В основе нашего рассуждения лежит и другая проблема: Maya не запретит вам «сделать что-нибудь не так». Структура пакета отличается чрезвычайной гибкостью, поэтому Maya редко дает строгие указания по поводу того, как следует решать конкретную задачу. Это очень заманчивая перспектива, поскольку практически к любой проблеме можно подойти с разных точек зрения. Недостаток такой неограниченной гибкости заключается в том, что подчас введенному в заблуждение разработчику удается создать приложение, которое на деле не укладывается надлежащим образом в рамки, заданные самой Maya. Нередко это выливается в долгие мучительные часы, отданные отладке и тестированию сценариев, которые попросту отказываются работать или ведут себя весьма

странно. В книге читатели **найдут** описание первых **шагов**, направленных на формирование необходимого фундамента знаний.

Большая часть материала, представленного в этой **главе**, нацелена на всесторонний обзор внутренней работы Maya. Этот материал в равной мере подходит как для художников, **так** и для программистов. Хотя **здесь**, по сути, не обсуждаются специальные вопросы программирования, изучение этой главы жизненно необходимо для лучшего понимания системы Maya, которое в дальнейшем поможет вам составлять более **качественные** программы. В **следующих** главах, посвященных языку MEL и интерфейсу C++ **API**, мы самым подробным образом расскажем о программировании пакета Maya.

2.1. Архитектура Maya

Если прежде чем обратиться к Maya вы работали с другими средствами трехмерной графики, то вы вспомните о том «внутреннем перевороте», который должны были пережить, переключившись на работу с Maya с вещей, привычных для вас по работе с другим пакетом. Хотя большинство 3D-пакетов, в конечном **счете**, пытаются решать одни и те же общие задачи, подход, который они исповедуют, может существенно различаться. Действия, которые нужно выполнить для совершения той или иной операции, часто являются специфичными для конкретного пакета и поэтому образуют некую **заранее** определенную структуру рабочего процесса. К счастью, для Maya характерна очень гибкая последовательность операций, не накладывающая **чрезвычайно** много ограничений. По большей части, *ты* или иную задачу можно решить более чем одним способом. Значит, вы можете использовать любой подход, лучше других **соответствующий** вашим нуждам. Со временем вы **узнаете**, как создавать свои собственные, нестандартные последовательности операций.

2.1.1. Обзор

Целостное представление системы Maya и ее декомпозиция на основные компоненты позволяют получить схему, **показанную** на рис. 2.1.

Как **пользователь** системы Maya вы взаимодействуете с ее **графическим** пользовательским интерфейсом. Вы выбираете пункты меню, изменяете параметры, анимируете и передвигаете объекты и т. д. Во время вашего взаимодействия с интерфейсом пользователя Maya, на самом деле, инициирует команды языка **MEL**. Они посылаются командному ядру (Command Engine), где интерпретируются и выполняются. Как вы, возможно, знаете, пакет Maya можно запускать и в пакетном

режиме. Графический интерфейс пользователя в нем отсутствует, и команды MEL передаются на выполнение непосредственно компоненту Command Engine.

Большинство команд MEL работают с графом зависимости **Dependency Graph**. Это происходит потому, что на интуитивном уровне **Dependency Graph** можно представить как полную *сцену*. Сцена содержит все важные данные и информацию, образующую трехмерный мир, включая объекты, анимацию, динамику, материалы и т. д. Граф зависимости не только описывает то, какие данные относятся к текущей сцене, но его структура и общая схема определяют способ обработки данных. Компонент **Dependency Graph**- это своего рода «сердце» и «мозг» Maya. Это очень важный компонент, и сейчас вы узнаете о нем более подробно.

2.2. Граф зависимости

2.2.1. Структура приложения

Чтобы лучше понять то, чем архитектура Maya отличается от других приложений трехмерной графики, важно выдвинуть на первый план структуру самых «типичных» 3D-приложений. Понимание отличительных черт пакета Maya определенно повлияет на то, каким образом вы будете строить собственные подключаемые модули.



Рис. 2.1. Система Maya

Как это делали в прошлом

Вообще говоря, любое приложение трехмерной графики предоставляет пользователю определенный набор инструментов для решения задач моделирования, анимации, освещения и рендеринга. Ясно, что эти задачи весьма отличаются друг от друга. Операции, выполняемые при моделировании, совсем не те, что выполняются при рендеринге. Данные и сведения, собираемые, используемые и сохраняемые при моделировании, также существенно отличаются от данных рендеринга.

Каждая из областей обладает своим собственным набором данных и операций, а из этого логически вытекает то, что их проектирование и реализацию можно осуществлять порознь. На деле каждую сферу можно описать как отдельный и обособленный модуль. Каждый модуль сохраняет и работает с конкретным типом данных. Модуль моделирования, например, может работать с геометрическими данными и данными о процессе моделирования. Для решения каждой задачи: моделирования, анимации, освещения и рендеринга - можно создать отдельный модуль. В каждой из этих задач реализуется вполне определенное множество функций. Результат - ясный и понятный интерфейс программирования каждого модуля.

Весьма иронично, что та самая ясность интерфейса не позволяет ему быть настоящему гибким. Скажем, вы проектируете 3D-приложение, куда входит система моделирования. Как только вы смоделировали статический объект, его можно оживить, передав модулю анимации. Тот позволил бы ему перемещаться в пространстве. Затем модуль рендеринга получил бы от модуля анимации анимированный объект и сформировал изображение на экране. В силу того что каждый модуль обладает четким интерфейсом, он точно знает, как ему общаться с другими модулями. А что произойдет, если вам захочется подключить систему деформации, которая изменяет форму анимированных объектов? Если модуль рендеринга знает лишь о том, на каком языке общаться с модулем анимации, вам придется либо видоизменять его и адаптировать к новым функциям, либо создавать новый модуль. В том и другом случае необходимость добавления новых функций означает, что модули анимации и рендеринга требуют обновления. Стало быть, простое подключение новых возможностей породит «эффект волны», которая захлестнет вашу систему и вызовет множество исправлений. Такой способ проектирования терпит крах, так как не позволяет легко добавлять новые функции без внесения изменений во все задействованные в этом модули. Нужен более универсальный и гибкий подход.

Как это сделано в Maya

В конце концов, вам захочется добавлять новые возможности так, чтобы не пришлось затрагивать ни один из существующих модулей. К тому же вы захотите изменять набор функций или интерфейс модуля, избегая ручного обновления всех зависимых компонентов.

В поисках лучшего решения вам стоит спросить себя: к чему же, в основе своей, действительно стремятся приложения трехмерной графики? На самом базовом уровне они лишь генерируют какие-то данные, которые затем обрабатываются при помощи серии операций. Конечным результатом этих шагов создания и обработки данных *может*, к примеру, стать итоговая полигональная сетка или ряд элементов изображения. Этапы преобразования фрагмента данных из начального состояния в окончательный результат можно легко определить как совокупность последовательных операций. Тогда результат одной операции подается на вход следующей и т. д. Каждая операция некоторым образом *изменяет* данные, а затем передает их следующей операции для дальнейшей обработки. Следовательно, процесс в целом можно представить себе так: данные приходят на один конец серии операторов и покидают другой конец в измененном виде. Подобное представление часто называют *моделью потока данных*. Кажется, что данные «перетекают» от одной операции к другой.

Проанализировав тенденции 3D-приложений, нетрудно обнаружить, что каждый отдельный модуль, на самом деле, можно разбить на более мелкие операции. Эти операции меньшего размера способны обрабатывать данные и передавать их следующим операциям. По сути, все функции, реализуемые обособленным модулем, можно инкапсулировать в серию взаимосвязанных операторов. Эти операторы принимают на вход одни данные и передают на выход другие. Будучи соединены последовательно, они образуют канал, или *конвейер*. Данные передаются первому оператору конвейера, а затем, после той или иной обработки, покидают последний оператор на другом конце. При правильном подборе операторов на одном конце конвейера можно поместить *трехмерную*-модель, а на другом получить ряд элементов изображения. Продолжая обобщать этот подход, можно без труда прийти к тому, чтобы любые данные приходили на один конец конвейера и любые данные покидали другой конец. На самом деле, не обязательно требовать даже того, чтобы они относились к трехмерному пространству. Можно поместить на вход конвейера текстовый файл и получить на выходе отредактированную цепочку символов.

Компания Alias | Wavefront реализовала ядро Maya, пользуясь такой парадигмой потока данных. Названное ядро физически представлено графом зави-

сности – компонентом Dependency Graph.¹ Ради простоты я буду обозначать Dependency Graph сокращенно, DG. Прежде чем двигаться дальше, должен отметить, что, с технической точки зрения, в основу DG положена *дву направленная модель*, а не строгая модель потока данных. Позднее я максимально подробно объясню различие между ними, а пока парадигма потока данных позволит сформировать интуитивное представление о механизме работы Maya.

DG – это поистине «сердце» Maya. Он предоставляет в ваше распоряжение все только что упомянутые базовые строительные блоки. Он позволяет создавать произвольные данные, подаваемые на вход серии операций и служащие сырьем для получения обработанных данных на другом конце конвейера. Данные и операции над ними инкапсулируются в DG как узлы. Узлы заключают в себе любое количество ячеек памяти, которые содержат данные, используемые Maya. Кроме того, в состав узла входит оператор, способный обрабатывать данные узла и получать в результате нечто иное.

Распространенные задачи трехмерной графики могут быть решены путем соединения ряда узлов. Данные первого узла поступают на вход следующего, тот так или иначе их обрабатывает и выдает новые данные, затем поступающие на вход другого узла. Таким образом, данные проходят по сети узлов, начиная с самого первого узла и заканчивая последним. На этом пути узлы могут свободно обрабатывать и редактировать данные, как им будет угодно. Кроме того, они могут порождать новые данные, которые затем передаются в другие узлы. Каждый тип узлов предназначен для выполнения *малого*, ограниченного множества *различных* операций. Узел, управляющий ориентацией одного объекта, который должен указывать на другой, реализует лишь эту конкретную функцию. Он не может, к примеру, деформировать объект. Для решения именно этой задачи будет создан узел деформации. Проектируя узлы для решения конкретных единичных задач, можно сохранить их простоту и повысить управляемость.

Для выполнения какого-то сложного преобразования данных создается сеть подобных простых узлов. Количество и способ соединение узлов не ограничены, что дает возможность создавать сети произвольной сложности. Во многом это похоже на конструктор Lego, в котором путем соединения простых блоков можно создавать сложные конструкции; тщс и узлы Maya можно связывать и объединять в сложные потоки данных.

Свидетельством гибкости графа DG является тот факт, что при помощи узлов создаются и обрабатываются все данные (*моделирование, динамика, анимация*,

¹ Технология Dependency Graph защищена патентами США №5.808.625 и №5.929.864. – Примеч. авт.

тонировка и т. д.) в среде Maya. Этот подход к построению сложных вещей из простых строительных блоков придает Maya реальную силу и гибкость. В то время как проекты прежних 3D-приложений предусматривали наличие отдельных модулей, Maya предлагает обширный набор различных узлов. Объединив достаточное количество этих узлов в сеть надлежащим образом, можно решить практически любую вычислительную задачу. Каждая функция, связанная с моделированием, динамикой, тонировкой и рендерингом, обеспечивается наличием сети взаимосвязанных узлов. При таком универсальном подходе нетрудно заметить, что в систему Maya, не имеющую постоянного множества интерфейсов или понятия обособленных модулей, можно с легкостью встраивать новые возможности.

Для внедрения в состав Maya новых функций достаточно лишь создать **новый** узел. Подобные узлы можно создавать так, что они выглядят как собственные узлы Maya, а значит, их можно безболезненно встраивать в структуру графа DG. Это, в сочетании с возможностью описания собственных команд, дает практически безграничные возможности расширения Maya. С добавлением мощного языка сценариев **MEL** в Maya почти не осталось элементов, которые нельзя было бы расширить или настроить в соответствии со своими нуждами.

2.2.2. Сцена

Сцена - это общее понятие, используемое для описания совокупного состояния трехмерной графики в данном 3D-приложении. Сцену образуют все модели, анимация и текстуры, а также все источники света, камеры и иная дополнительная информация, такая, как настройки рендеринга и т. д. Сценой в пакете Maya является граф DG. Это одно и то же. Записывая сцену на диск или считывая ее с диска, Maya просто сохраняет или считывает граф зависимости. DG описывает всю сцену как сеть взаимосвязанных узлов.

В обычном приложении вся информация, которая составляет сцену, хранится в отдельных специальных структурах. Эти структуры, как правило, находятся в центральной базе данных. Граф DG представляет собой множество взаимосвязанных узлов, где содержатся данные, поэтому сцена сохраняется не централизованно, а в виде распределенной сети. Совместно все узлы образуют распределенную базу данных. Можно добавлять новые узлы в сеть и удалять существующие, поэтому по своей структуре это очень динамичное образование.

Как при таком распределении информации по всем соединенным узлам получить конкретный фрагмент данных? Если другие системы сохраняют свои сведения в центральной базе данных, то конкретный фрагмент можно легко

найти. Несложно, например, направить сцене запрос на перечисление всех источников света. Когда же все данные содержатся в сети узлов, запросы такого типа могут представлять трудности. К счастью, Maya оберегает пользователей от изучения особенностей функционирования DG и предоставляет удобные функции доступа к сцене, позволяющие работать с ней так, будто она содержится в центральном хранилище данных. Интерфейс программирования DG не требует знания схемы его организации и структуры. К примеру, вы можете быстро и легко получить список всех объектов данного типа, не зная их расположения в сети.

Даже несмотря на то что различные интерфейсы программирования скрывают изрядную долю подобных **сложностей**, важно понимать, что реально происходит внутри. Вам предстоит неоднократно оказываться в ситуациях, когда более глубокое понимание применяемых в Maya методов хранения сцены поможет разрабатывать более надежные и быстрые приложения.

2.2.3. Отображение графа зависимости

Ниже приведен пример окна Hypergraph (Гиперграф). Окно **Hypergraph** – это видимое окно графа зависимости, ядро Maya в буквальном смысле слова. На рис. 2.2 показан результат лишь нескольких операций моделирования. Можно заметить, что количество узлов и связей между ними быстро **растет** даже после столь небольшого числа операций.

На первый взгляд DG может выглядеть устрашающее, однако знания, приобретенные при прочтении следующих глав, позволят вам быстро разобраться в том, что означает каждый узел и как он соединен с другими узлами. Вы уже будете замечать, что существуют различные типы узлов; суставы, множества, доводки и т. д. Эти узлы соединены с другими. Линии указывают на соединения между узлами, а стрелки обозначают направления потоков данных.

Если вы не слишком хорошо знакомы с окном Hypergraph, я настойчиво рекомендую вам почитать учебные пособия по Maya и изучить соответствующую документацию. В ходе своих занятий программированием вы будете бесконечное число раз пользоваться окном Hypergraph, и потому так важно **понять**, как оно работает.

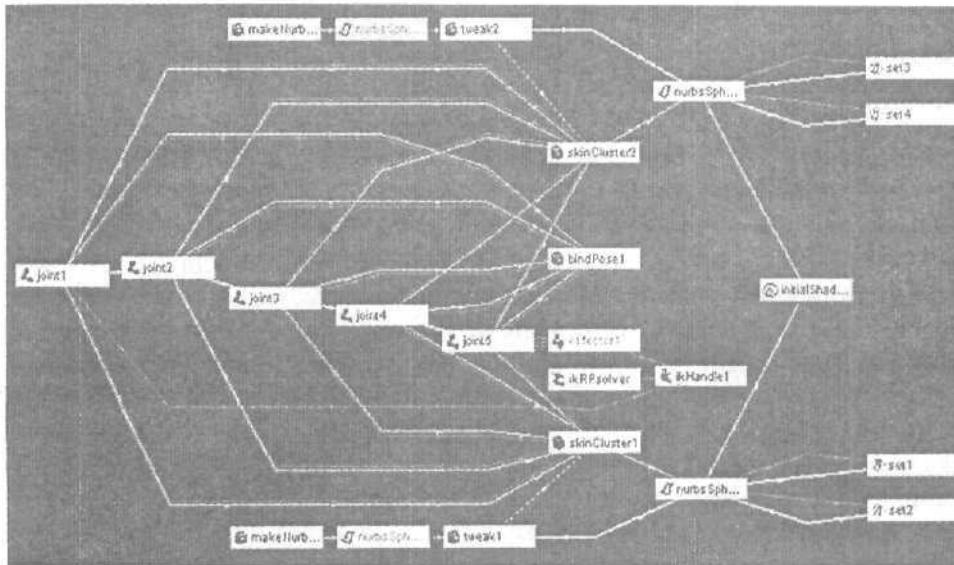


Рис. 2.2. Пример окна Dependency Graph

2.2.4. Поток данных

Далее предметом подробного изучения станет отдельно взятый DG небольшого размера. На рис. 2.3 изображен более простой граф зависимости, на котором показаны некоторые общие функции, присущие каждой сети узлов. Более крупные и сложные сети есть не что иное, как результат добавления большего числа узлов и соединений. Основополагающие средства создания узлов и соединений между ними будут теми же самыми.

Приведенная конфигурация узлов очень распространена, и вы, несомненно, будете видеть ее довольно часто. Она отражает типичный процесс анимации значений в Maya. Данный граф DG состоит всего лишь из трех узлов. Первым является узел `time` (время), вторым – узел `curveAnim` (кривая анимации); последним стоит узел `transform` (преобразование). Ближе познакомившись с узлами различных типов, вы, скорее всего, захотите больше узнать о том, как они работают. В главах, посвященных MEL и C++ API, будет сказано, где найти информацию о всевозможных типах узлов и соответствующих атрибутах данных.

Двигаясь по стрелкам вдоль линий, соединяющих узлы, вы заметите, что узел `time1` связан с узлом `nurbsSphere1_translateX`, который, в свою очередь, соединен с `nurbsSphere1`. Интуитивно можно понять, что информация о времени пе-

редается анимационной кривой, а затем анимационная кривая передается узлу преобразования. Из рисунка видно, что выход `time1` служит входом кривой анимации. Его значением является текущее время. По сути дела, узел `time1` входит в состав каждой сцены. По известному значению текущего времени кривая анимации находит некоторое выходное значение. Ее выход является результатом расчета анимационной кривой в данный момент времени. Поэтому, коль скоро время изменяется, кривая анимации рассчитывается в разные его моменты. Если кривая изменится, то значение результата тоже претерпит изменение, а это отразится на анимационной кривой. Данное значение передается узлу преобразования как значение `translateX`. Оно определяет положение объекта по оси x. При изменении этого значения положение `nurbsSphere1` вдоль оси x станет иным. Таким образом, положение сферы, в итоге, зависит от результатов расчета анимационной кривой в данный момент времени. Вследствие изменения текущего времени узлы заново рассчитывают собственные значения, и в результате сфера движется вдоль оси x. Интуитивно можно представить, что данные начинают свое движение с первого узла слева и движутся до тех пор, пока не достигнут последнего узла справа, и это перемещение данных сопровождается их обработкой. Поток движется по стрелкам слева направо.



Рис. 2.3. Типичная конфигурация анимации DG

2.2.5. Узлы

Так как узлы действительно представляют собой фундаментальные строительные блоки Maya, нужно представить их более подробно. Для того чтобы понять, что они содержат и как они работают, я буду изображать их в схематической форме. Базовый вариант узла показан на рис. 2.4.

Каждый узел имеет одно или несколько связанных с ним свойств. Такие свойства обычно называют *атрибутами*. Атрибут – это конкретное свойство данного узла. Узел `transform`, к примеру, содержит атрибут `translateX`, в котором хранится текущее положение объекта на оси x. Каждый узел типа `makeNurbSphere` имеет атрибут `radius`, а также многие другие. В этом примере узел имеет два атрибута: `input` и `output`. По идеи, данные атрибутов содержатся внутри УЗЛОВ. В главе об интерфейсе C++ API вас ждет более подробное обсуждение организации хранения данных в узлах.

Наряду с атрибутами все узлы содержат функцию `compute`. Хотя вы не увидите ее на экране в Maya, можете считать, что ее имеет каждый узел. Функция `compute` должна принимать один или несколько входных атрибутов и вычислять один или несколько выходных. Как упоминалось ранее, основная задача узла состоит в хранении данных и, возможно, той или иной их модификации. Данные сами по себе просто хранятся в атрибутах узла. И именно функция `compute` выполняет всю работу по их модификации. Она имеет представление о том, как на основе исходных данных получить выходной результат.

Зная тот факт, что `сцена` – это не что иное, как граф DG, который представляет собой ряд взаимосвязанных узлов, можно прийти к интересному логическому заключению относительно функции `compute`. Отдельные реализации этой функции можно рассматривать как «нейроны» в сети узлов, образующих «мозг» Maya. Концептуально последний не отличается от мозга человека. В то время как в нашем мозге нейроны объединяются посредством `дendritov`, в мозге Maya вместе объединяется именно ряд узлов, которые, благодаря своим функциям `compute`, генерируют поток данных, меняющихся на пути от одного узла к другому.

«Интеллект» Maya – непосредственный результат совместного действия всех функций `compute`. Граф DG может состоять из множества различных узлов, поэтому вы можете на деле создавать разные варианты «мозга», порождающего и обрабатывающего данные. Создавая свои собственные узлы и составляя соответствующие функции `compute`, вы фактически сможете встраивать в этот механизм свои собственные компоненты. Ваш нестандартный узел может быть установлен в одну или несколько позиций графа зависимости. После этого, благодаря наличию в нем определенной функциональности, он внесет свой вклад в обработку данных.

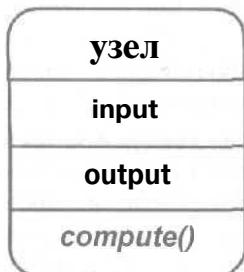


Рис. 2.4. Базовый вариант узла

2.2.6. Атрибуты

Ранее было сказано, что все данные хранятся в **каждом** узле как атрибуты. Атрибут – это некоторая область узла, где **содержится спределенная** информация. Атрибут имеет **имя** и **тип данных**. Имя служит для краткого описания атрибута и в дальнейшем используется для его обозначения. Например, имя **radius** присвоено одному из атрибутов узла **makeNurbSphere**, который управляет радиусом сферы. Тип данных определяет, какие данные можно размещать в атрибуте. Если, к примеру, атрибут имеет тип **long**, то он **может** содержать целые значения. Иначе, если атрибут имеет тип **float**, его значениями могут быть вещественные числа. Эти два типа относят к **простым типам данных**, так как в них можно записать только одно значение. Неполный перечень простых типов включает **boolean**, **byte**, **char**, **short**, **long**, **float**, **double** и др. Они соответствуют основным числовым типам, которые **имеются** на компьютере. Узел на рис. 2.5 содержит один атрибут **size** типа **float**.



Рис. 2.5. Простой узел

Поддерживаются и более сложные типы данных. В качестве атрибута узла может храниться целая геометрическая сетка или поверхность неравномерных рациональных би-сплайнов (NURBS). На самом деле, в виде атрибутов хранятся все данные, будь то простые или сложные. Возможность получения атрибутов, состоящих из простых или сложных данных, позволяет добиться поразительной гибкости.

Кроме того, Maya разрешает объединять простые типы данных в более сложные структуры. Если, к примеру, взять три элемента данных типа **float**, то можно описать точку (позицию) в трехмерном пространстве. Точка состоит из трех атрибутов типа **float**: **x**, **y**, **z**. Сама по себе она может считаться **родительским атрибутом** по отношению к **атрибутам-потомкам** **x**, **y**, **z**. Родитель как такой тоже является атрибутом, поэтому вы можете обращаться как к нему, так и косвенно к его потомкам, либо к каждому из **потомков** в отдельности. На рис. 2.6 узел имеет атрибут **pointA**, состоящий из трех атрибутов **x**, **y**, **z**.



Рис. 2.6. Узел, содержащий составной атрибут

Maya именует атрибут **pointA** *составным*, так как тот возник в результате объединения других атрибутов. Атрибут **pointA** считается родителем, а атрибуты **x**, **y**, **z**- потомками. Ограничения на количество и типы **потомков** одного атрибута не накладываются. Поэтому вы можете строить произвольные «деревья» атрибутов.

Наряду с созданием составных атрибутов, можно создавать *массивы* атрибутов. Массив атрибутов - это атрибут, содержащий *последовательность* из одного или нескольких других атрибутов. Например, вы можете запомнить массив точек, которые описывают положение ряда частиц. Узел на рис. 2.7 содержит атрибут с именем **points**. Он представляет собой массив точечных атрибутов. Атрибут **point** - это составной атрибут, образованный тремя атрибутами-потомками: **x**, **y**, **z**. Этот простой пример на практике иллюстрирует создание относительно сложной иерархии объектов.

Итак, атрибут - это очень гибкое средство хранения произвольных данных. В атрибутах могут содержаться как простые данные, так и более сложные их комбинации. Атрибуты позволяют устанавливать отношения «родитель – потомок», поэтому вы можете свободно создавать иерархии произвольной сложности. Пользуясь этим наряду с возможностью сохранения массивов иерархий, вы почти всегда сможете представить любую сложную структуру данных при помощи атрибутов Maya.

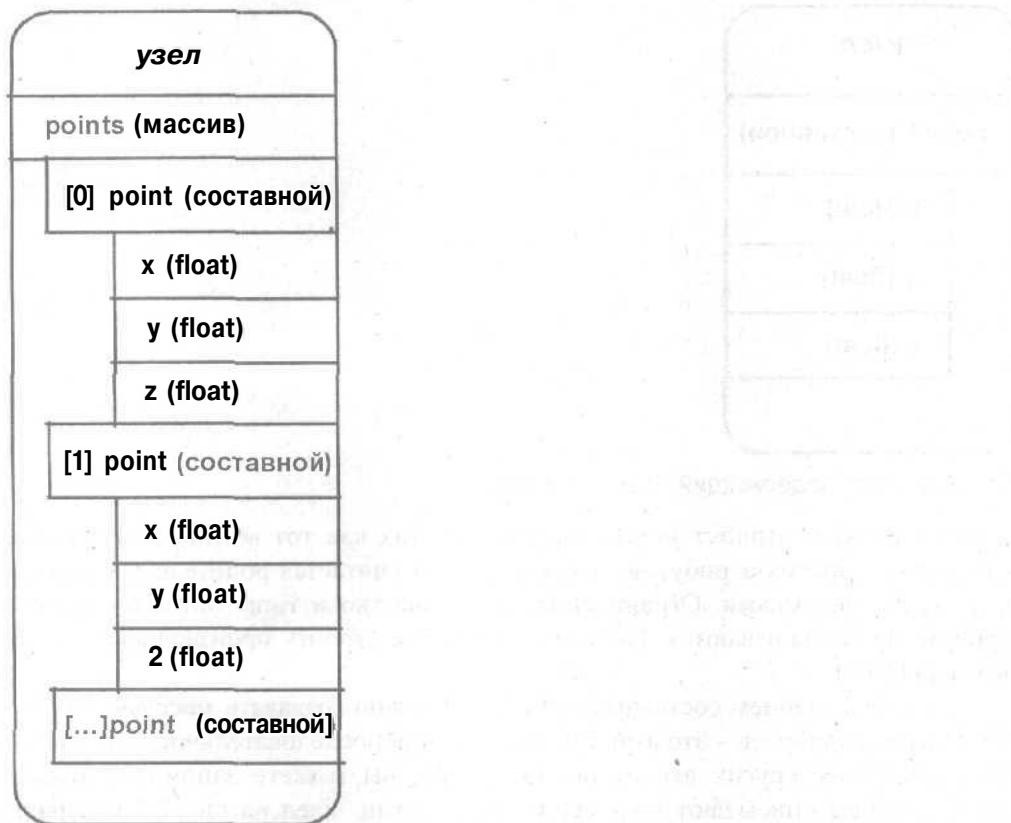


Рис. 2.7. Узел, содержащий массив составных атрибутов

2.2.7. Функция compute

Ранее было сказано, что функция `compute` - это фактически «мозговой центр» узла. Атрибуты обеспечивают хранение его данных, но помимо этого они ничего не делают. Чтобы сделать нечто полезное, данные узла нужно так или иначе обработать. К примеру, чтобы заставить персонаж ходить, его нужно перемещать во времени. Мышцы и кожа персонажа должны со временем изменяться. Начиная свою работу с персонажа в статической позе, система должна изменять и деформировать его с течением времени. В каждом кадре персонаж должен деформироваться по-разному. Результат такой деформации в каждом кадре - это результат обработки статичного персонажа, поданного на один конец конвейера

DG. с получением на втором конце другого, полностью деформированного персонажа.

По своей сути функция `compute` принимает один или несколько входных атрибутов и вычисляет результат, представленный выходным атрибутом. Ее можно рассматривать как функцию языка программирования вида:

выход = compute(вход0, ..., входN).

Так как входы и выходы являются атрибутами, которые могут принимать различную форму, функция `compute` способна работать с широким спектром разнообразных данных. Например, она может принять на вход одну и возвратить другую, новую NURBS-поверхность. Функция `compute` может вычислить некоторую меру входной поверхности и возвратить единственное значение, представляющее, скажем, ее площадь. Таким образом, гибкость, обеспечиваемая общим характером атрибутов, означает, что функция `compute` способна создавать произвольно сложные данные, а также принимать произвольно сложные данные на вход.

Очень важно отметить тот факт, что и входные, и выходные атрибуты функции `compute` являются локальными атрибутами узла. По сути, для решения поставленной задачи функции `compute` достаточно проверять лишь атрибуты своего собственного узла. Эта функция никогда не пользуется информацией из других узлов или прочих источников. Все данные, к которым она обращается в процессе вычислений, должны исходить только из самой функции и ниоткуда больше. Это очень важное ограничение будет неоднократно повторяться по ходу книги, так как оно носит фундаментальный характер применительно к разработке «хороших» узлов.

Что касается внутренней организации функции `compute`, то она может выполнять любые действия, необходимые для вычисления окончательного результата. Maya никогда не узнает обо всех подробностях процесса вычисления результата. Функцию `compute` можно рассматривать как «черный ящик». Зная выход такой функции, невозможно определить, с чего начинался расчет результата. Преимущество данного подхода состоит в том, что вам никогда не придется изучать внутреннее устройство узла при желании его использовать. Узел, который выполняет сопряжение двух граней, может иметь очень сложную внутреннюю структуру, однако вы никогда об этом не узнаете. Пользуясь таким узлом, вам следует знать только то, что при наличии данного множества входных форм функция `compute` порождает конкретную выходную форму.

С точки зрения внешнего наблюдателя, узел публикует лишь используемые им атрибуты: внешние и внутренние. Фактически эти атрибуты описывают *интерфейс*

узла. Пользуясь узлом, вы обращаетесь к нему, считывая и устанавливая его атрибуты. Вам никогда не удастся получить непосредственный контроль над функцией `compute`. При заданных входных атрибутах `radius`, `start sweep` и др. узел `makeNurbSphere` построит NURBS-поверхность сферической формы. Изменяя входные атрибуты узла, вы управляете созданием NURBS-поверхности. Для того чтобы сделать сферу крупнее, нужно просто увеличить значение атрибута `radius`. Тогда узел построит сферу большего размера. Узел создаст сферическую поверхность даже в том случае, если вы не знаете, как это происходит.

Преимущество такого проекта состоит в том, что когда вы, к примеру, создаете свой собственный узел `makeMyNurbSurface`, вы вправе реализовать такой метод построения поверхности, какой пожелаете. Если позднее вы выберете более эффективный метод, то сможете использовать его, не разрушая граф DG и его сеть. Важно лишь то, что ваша функция `compute` действительно строит поверхность. А то, как она это делает, полностью зависит от вас.

Зависимые атрибуты

Несмотря на то что в моем описании фигурируют атрибуты, являющиеся входом и выходом, важно отметить, что Maya не делает подобного различия. Для нее не существует таких понятий, как входные или выходные атрибуты. Любые атрибуты просто содержат данные.

Однако необходимость логического разделения атрибутов на входы и выходы существует. Входной атрибут – это атрибут, поступающий в функцию `compute` в составе узла. Выходной атрибут – это атрибут, значение которого рассчитывается функцией `compute` на основе одного или нескольких входных атрибутов. Чтобы показать, как при помощи узла можно найти объем сферы, обратимся к узлу `sphereVolume`. Он показан на рис. 2.8.

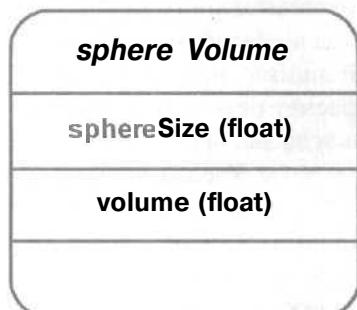


Рис. 2.8. Узел `sphereVolume`

Узел имеет два атрибута: **sphereSize** и **volume**, каждый из которых хранится как число типа float. Атрибуты, как всегда, всего лишь содержат значения. И нельзя ничего сказать о том, какой из этих атрибутов служит входом, а какой – выходом.

Известно, что объем сферы зависит от ее размера, и потому можно легко прийти к выводу о том, что значение атрибута **volume** должно зависеть от значения атрибута **sphereSize**. Значит, атрибут **volume** зависит от атрибута **sphereSize**. Атрибут **volume** можно считать выходным атрибутом, атрибут **sphereSize** – входным. Ответственность за расчет значения атрибута **volume** возлагается на функцию `compute`. Вычисление объема можно схематично записать так:

volume = compute(sphereSize).

Заметим, что способ вычисления объема вами не описан. Как и в случае с любым узлом, вам совершенно не нужно знать, как работает функция `compute`. Чтобы пользоваться узлом, надо знать лишь о том, какие входные и выходные атрибуты он имеет. Остальное – забота функции `compute`.

Теперь стало ясно, что вы установили внутреннюю связь между обоими атрибутами. Атрибут **volume** является результатом функции `compute`, принимающей на вход атрибут **sphereSize**. Если **sphereSize** изменится, то **volume** потребует повторного вычисления. Нужен механизм описания этого отношения зависимости между атрибутами **sphereSize** и **volume**.

Задание такого отношения зависимости является делом конкретного узла. На уровне внутренней реализации эта задача решается при помощи функции `attributeAffects` из состава C++ API. Функция сообщает Maya о том, что один атрибут влияет на другой, фактически это и означает задание подобного отношения зависимости между парой атрибутов. В рассматриваемом примере вызов функции примет следующий вид:

```
attributeAffects( sphereSize, volume );
```

Здесь указано, что атрибут **sphereSize** оказывает влияние на значение атрибута **volume**. Единственный вызов функции закрепил два немаловажных положения. Во-первых, он уведомил Maya о том, что атрибут **volume** является результатом вычислений, т. е. его значение может быть получено только путем вызова функции `compute`. Во-вторых, он сообщил Maya о том, что при вычислении **volume** входом функции `compute` должен служить атрибут **sphereSize**.

Задание этого отношения зависимости для всех выходных атрибутов позволяет определить, как узел выдает новые данные. Всякий раз, когда запрашивается значения атрибута **volume**, оно будет найдено на основе атрибута

sphereSize. В то же время при изменении **sphereSize** атрибут **volume** также будет рассчитан заново.

Заметьте, что я не упомянул о том, как на самом деле узел вычисляет объем. В узле может найти применение любой алгоритм, пригодный для получения этого значения. Чтобы пользоваться узлом, вам даже не нужно знать, как рассчитываются атрибуты. Вы можете просто запросить значение атрибута **volume**, и узел его вычислит, а затем вернет результат. Если вы намерены заняться созданием своих собственных узлов Maya, то уже можете понять, на что уйдет большая часть времени - на составление функции `comprute`.

Хотя в этом простом примере был всего один входной атрибут, который служил основой расчета одного выходного атрибута, на практике функция `comprute` может принимать любое количество входных атрибутов и рассчитывать исходя из этого любое количество выходных. Требуется лишь задать отношение зависимости между рядом входов и заданным выходным атрибутом. Эта же схема применима и к другим выходам. Кроме того, выходной атрибут может использоваться как входной.

Опишем узел **boxMetrics**, входом которого является объект прямоугольной формы; узел рассчитывает площадь верхнего основания (**areaOfTop**) и объем этого объекта (**volume**). На рис. 2.9 показана схема узла. Описание объекта составляют его ширина, высота и глубина, т. е. атрибуты **width**, **height** и **depth**.



Рис. 2.9. Узел **boxMetrics**

Все атрибуты узла описаны как данные типа `float`, и между ними пока нет никаких внутренних взаимосвязей. Все они – просто ячейки памяти, содержащие по одному значению с плавающей запятой.

Для расчета параметров объекта будут использоваться следующие формулы:

areaOfTop = width x depth;

volume = areaOfTop x height.

Коль скоро объект имеет прямоугольную форму, площадь его верхнего основания фактически равна площади нижнего основания. Поэтому, выразив эту мысль в своей функции `compute`, вы получите:

areaOfTop = compute(width, depth);

volume = compute(areaOfTop, height).

Обратите внимание, что для расчета атрибута **volume** вам нужно вычислить значение **areaOfTop**. Значение **height** используется непосредственно. Итак, отыскание значения **volume** зависит от расчета **areaOfTop**. Как вы опишете это отношение в Maya? Отношение зависимости для атрибута **areaOfTop** будет сформулировано так:

```
attributeAffects( width, areaOfTop );
attributeAffects( depth, areaOfTop );
```

Отношение зависимости для атрибута **volume** записывается аналогично.

```
attributeAffects( areaOfTop, volume );
attributeAffects( height, volume );
```

Что касается атрибута **volume**, то делать что-то особенное не потребуется. Нужно лишь, как и прежде, указать, от каких атрибутов он зависит. Maya автоматически обеспечит вычисление значения **areaOfTop** прежде, чем оно будет использоваться при расчете **volume**.

В случае изменения значений **width** или **depth** значение **areaOfTop** вычисляется заново. Поскольку **areaOfTop**, помимо прочего, влияет на объем. Maya выполнит перерасчет значения **volume**. Аналогично, при изменении **height** значение **volume** опять-таки будет рассчитано вновь. Так как **areaOfTop** не зависит от значения **height**, изменение последнего не затронет площадь основания объекта.

Время

Прежде чем продолжать изложение материала, следует заметить, что узел не обязательно должен что-нибудь вычислять. Ситуация, когда узел лишь содержит один или несколько атрибутов, является совершенно нормальной. Эти узлы ведут себя как простые хранилища данных. Узел **time** – яркое тому подтверждение.

ждение. Maya хранит текущее время в узле **time** с именем **time1**. Схема узла **time** отражена на рис. 2.10.



Рис. 2.10. Узел time

Узел **time** содержит значение времени в атрибуте **outTime**. Как только вы пе-редвинете бегунок текущего кадра или щелкнете по кнопке **Play** (Воспроизвести), Maya установит значение **outTime** узла **time1**, равное времени текущего кадра. Узел **time** не производит никаких вычислений. Это не говорит о том, что он не содержит функции **compute**. Функцию **compute** содержат все узлы. Но так как вы не задали никаких отношений зависимости между атрибутами, функция **compute** не будет вызываться никогда.

2.2.8. Соединение узлов

Теперь вы знаете, что такое узлы и для каких целей они предназначены. Несмотря на то что при наличии множества входных значений узлы могут вычислить почти все, что угодно, отдельного узла вы не получите много. Реальная способность узлов производить сложные вычисления приходит по мере их связывания в сети. Тогда каждый из узлов, как и прежде, решает свою вычислительную задачу, но теперь результаты этих расчетов передаются другим узлам, которые также занимаются вычислениями. Соединяя узлы, вы можете создавать цепочки вычислений, приводящие к окончательным результатам, которые представляют собой выходные атрибуты узлов в самом конце этих цепочек.

Возвращаясь к исходному примеру с анимационным графом DG, вы можете углубить свои изыскания и увидеть, как же на самом деле узлы связаны друг с другом. Рис. 2.11 показывает исходную конфигурацию узлов DG.



Рис. 2.11. Конфигурация узлов анимации

Схематично этот DG можно представить так, как показано на рис. 2.12. Здесь вы можете ясно увидеть атрибуты каждого узла и способ их соединения. Узел **time** содержит единственное значение **outTime**, соответствующее текущему времени. Оно передается атрибуту **input** узла **animCurveTL**. Этот узел содержит кривую анимации. Так называется **кривая**, которую вы изменяете в редакторе **Graph Editor**, чтобы анимировать то или иное значение. Этот узел рассчитывает атрибут **output**, передаваемый атрибуту **translateX** узла **transform**. Узел **transform** – это достаточно крупный узел с большим количеством атрибутов. Для простоты показаны лишь атрибут **translate** и его потомки **translateX**, **translateY** и **translateZ**.

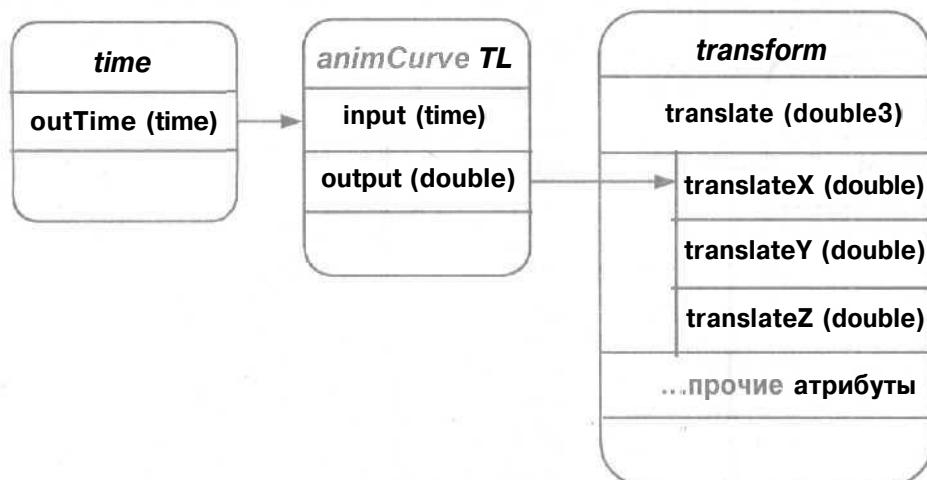


Рис. 2.12. Схематичное представление DG

Из этого рисунка видно, что процесс соединения узлов состоит в связывании их атрибутов. Поток данных и информации берет начало в одном из атрибутов одного узла и заканчивается в другом атрибуте другого узла. Обратите внимание, что атрибут может быть связан лишь с иным атрибутом того же типа. Атрибут **outTime** узла **time** имеет тип **time**. Его можно соединить с атрибутом **input** узла **animCurveTL**, поскольку тот тоже имеет тип **time**. Однако соединить **outTime** с атрибутом **translateX** узла **transform** было бы невозможно, так как последний имеет другой тип, а именно **double**. Не вдаваясь в подробности, можно сказать, что в Maya предусмотрен механизм преобразования некоторых типов.

Узлу - и это важно - никогда неизвестно о том, что он соединен с другими узлами. Он не располагает сведениями о том, с какими узлами соединен или в каком **именно** месте сети DG расположен. По сути, узлу известны лишь собственные входные и выходные атрибуты. Он не **знает**, участвуют ли они в соединениях. Поток данных, передаваемых одним узлом другому, обрабатывается системой Maya, поэтому узел, как и прежде, лишь вычисляет свои выходные атрибуты. Распространив этот принцип «ограниченности знаний» на все узлы, вы можете использовать их как настоящие строительные блоки. Узлы **никогда** не должны знать о контексте, в котором они применяются.

Один атрибут может быть связан со многими. Значение атрибута передается всем связанным атрибутам других узлов. Пример такой конфигурации представлен на рис. 2.13.

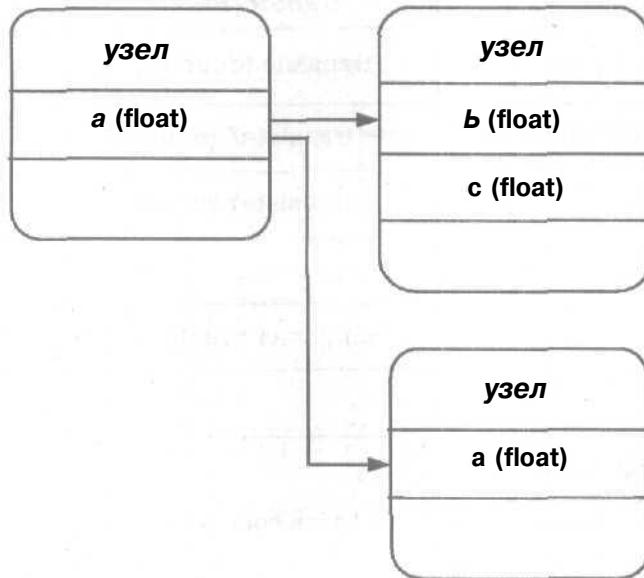


Рис. 2.13. Связь единичного атрибута со многими

В то же время запрещено, как это показано на рис. 2.14, подавать несколько атрибутов на вход одного. Такое ограничение понятно, поскольку атрибут содержит только одно значение, а устанавливая множество связей с участием этого атрибута, вы создаете **ситуацию**, в которой узел не способен выбрать используемое соединение. Какое значение при такой конфигурации должен принять атрибут **a**? Будет ли это **b** или **c**? Правильного ответа **не** существует.

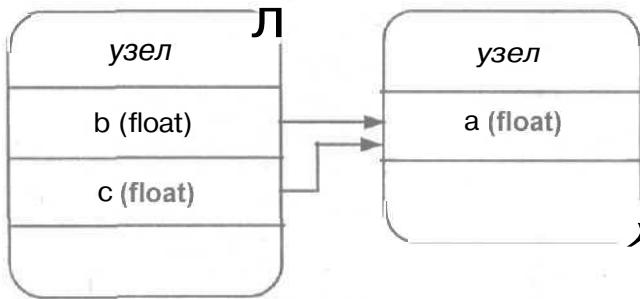


Рис. 2.14. Запрещенное соединение

Если атрибут не является целевым атрибутом соединения, его значение хранится в пределах узла. Если соединение построено так, что данный атрибут принимает значение от другого, то его значение заимствуется из атрибута, создавшего это соединение. Атрибут отбрасывает значение, которое содержалось в нем ранее, и принимает значение входящего соединения. Аналогично, если соединение между двумя атрибутами разрывается, то значение, хранившееся в предыдущем целевом атрибуте, является последним, которое имел атрибут до разрыва соединения. Поэтому, если значение, полученное от *связанного* атрибута, составляло 1,23, а затем соединение было разорвано, то атрибут сохранил это значение 1,23. Теперь, когда соединение отсутствует, изменение атрибута-источника не повлияет на атрибут, оказавшийся в изоляции.

2.2.9. Узлы ориентированного ациклического графа (ОАГ)

До сих пор узлы описывались в общих чертах. Те из них, что входят в состав DG, следует точнее называть *узлами зависимости*. Такое название логично, поскольку эти узлы являются узлами графа зависимости. При необходимости узлы можно добавлять в граф и удалять из него. Любые атрибуты одного узла могут быть соединены с любыми атрибутами другого, что позволяет создать простую или сложную сеть взаимосвязанных узлов. Благодаря возможности свободно соединять узлы, можно получить сеть произвольной структуры.

Это обстоятельство обеспечивает немалую гибкость; но что если вы захотите придать тем или иным узлам некую логическую структуру? В компьютерной графике часто употребляется понятие *иерархии*, позволяющее описывать отношения «родитель – потомок», которые возникают между 3D-объектами. К примеру, вы намерены собрать модель автомобиля, пользуясь колесами как дочерними объектами. Во время движения автомобиля колеса тоже будут передвигаться. Такое отношение нельзя описать при помощи графа DG.

Для разрешения данной проблемы служат специальные узлы ОАГ, или ориентированного ациклического графа. *Ориентированный ациклический граф* - это, по сути, технический термин, обозначающий иерархию, в которой узел не может быть родителем и потомком самого себя. Важно понимать тот факт, что узлы ОАГ являются узлами DG. Узлы ОАГ присутствуют в DG подобно всем прочим узлам. Они всего лишь имеют особый вид, *обеспечивающий логическую связь «родитель - потомок» с другими узлами*.

Соотношение между узлами ОАГ и DG может сбить с толку большинство пользователей. Основная причина этого заключается в том, что в Maya не предусмотрено никаких визуальных *средств*, позволяющих увидеть истинные отношения между ними. Инструмент **Hypergraph** позволяет просматривать узлы или в виде иерархии ОАГ [Scene Hierarchy (Иерархия *сцены*)], или как узлы графа зависимости [Up- and Downstream Connections (Восходящие и нисходящие соединения)], но не дает возможности увидеть *те и другие* одновременно. Если бы такое было возможно, это соотношение могло бы выглядеть примерно так, как показано на рис. 2.15.

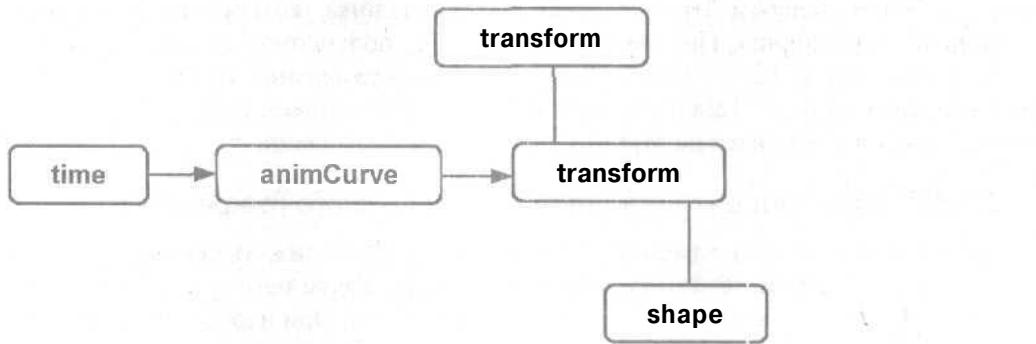


Рис. 2.15. Узлы ОАГ и графа зависимости

Узлы **transform** и **shape** - это узлы ОАГ, подробным *изучением* которых мы зайдемся немного позднее. Сейчас достаточно понять, что они вводят отношение «родитель - потомок». Имеются два узла **transform**, один из которых является потомком другого. Дочерний узел **transform**, в свою очередь, обладает потомком **shape**. Узлы **time** и **animCurve** представляют собой стандартные узлы графа зависимости. Стрелки, как и прежде, показывают, как атрибуты различных узлов связаны друг с другом. Вертикальные *линии*, соединяющие узлы ОАГ, указывают на имеющиеся отношения «родитель - потомок».

Ввиду наличия дерева, образованного в результате отношениями «родитель - потомок», узлы ОАГ образуют иерархию «сверху - вниз». В то же время другие узлы зависимости могут иметь атрибуты, связанные с любыми узлами иерархии ОАГ. Поэтому граф DG можно «прочитать» в двух направлениях: сверху вниз, начиная с корневого узла ОАГ и спускаясь по всем узлам преобразования и форм, а также слева направо, начиная с самого левого узла и следя по всем связанным узлам в горизонтальном направлении.

Очень важно, прежде всего, понять, что узлы ОАГ - это просто узлы графа зависимости. Они являются лишь средством описания отношений «родитель - потомок», отсутствующим в обычных узлах графа DG. Во всех других отношениях они функционируют точно так же.

Преобразования и формы

Создавая 3D-объект в среде Maya, вы фактически получаете сочетание двух узлов ОАГ. Первый из них - узел **transform**, второй - узел **shape**, и оба они входят в состав объекта. Maya располагает широким спектром узлов **shape**, включая сетки, NURBS-кривые и поверхности, пружины, камеры, источники света, частицы и т. д. Отношение «родитель- потомок», действующее между узлами **transform** и **shape**, проиллюстрировано рис. 2.16.

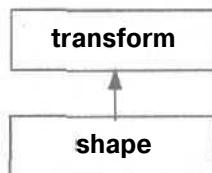


Рис. 2.16. Узел **shape**-потомок узла **transform**

Узел **transform** описывает местоположение объекта в пространстве, а узел **shape** определяет, как на самом деле он должен выглядеть. Задание этого отношения важно как для моделирования, так и для анимации. Чтобы вывести это отношение в окне **Hypergraph**, выберите опцию **Show Hierarchy** (Показать иерархию). Иерархия, соответствующая NURBS-сфере, изображена на рис. 2.17.

Узел **nurbsSphereShape2**, относящийся к категории **shape**, содержит NURBS-поверхность. Его атрибуты предназначены для хранения данных, которые ее описывают. Расположенный выше узел **nurbsSphere2** является узлом **transform**. Его цель - преобразование этой NURBS-поверхности. Обычно узел **transform** работает с формой, которая, по сути, описана в пространстве объектов, и переносит ее в мировое пространство. Такое простое объяснение подразумевает, что

данный узел **transform** действительно не имеет другого родителя той же категории **transform**. Наследование мы обсудим немного позднее. А сейчас важно понять, что узел **shape** *не может существовать* без узла **transform**. Это ясно, поскольку при отсутствии **transform** Maya не сможет узнать, в какой точке трехмерного пространства следует поместить форму. Если вы хотите оставить форму в пространстве объекта, просто установите преобразование узла **transform**, равное *identity* (*тождество*), и оно не будет играть никакой роли.

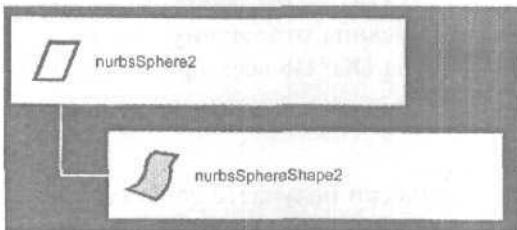


Рис. 2.17. Иерархия, соответствующая NURBS-сфере

Любой объект, который «желает» отобразить себя в трехмерном пространстве, должен быть построен при помощи такой структуры типа «преобразование - форма». Прекрасным примером являются локаторы. Не описывая физических объектов, которые можно визуализировать, они предоставляют графические средства реализации других операций. Для задания локаторов служит специальный **узел-локатор** категории **shape**. Узел **shape** всего лишь вычерчивает локатор в трехмерном окне экрана. Подобно другим узлам **shape**, он имеет предка - узел **transform**, который определяет его положение и **ориентацию** в пределах **сцены**.

И узел **transform**, и узел **shape** являются узлами ОАГ и представляют собой частный случай узла зависимости общего вида, предназначенный для поддержки отношения «родитель – потомок». Как было указано ранее, ОАГ – это сокращенное обозначение ориентированного ациклического графа. Выражение «*ориентированный ациклический*» на самом деле означает, что узел не может пребывать в двух местах иерархии при ее обходе, начиная с корня и заканчивая данным узлом. Это предотвращает возникновение циклических зависимостей, в которых какой-то узел может быть и потомком, и косвенным родителем данного узла. Понятие «*граф*» отражает то обстоятельство, что рассматриваемая иерархия не является, строго говоря, *деревом*. Каждый дочерний узел дерева обладает только одним предком. В случае с графом потомок может иметь несколько родителей, как это происходит при создании экземпляров объектов. Граф DG с тиражируемыми узлами содержит множество путей, ведущих от корневой вершины до кон-

крайнего тиражируемого узла. Каждый из этих путей представляет уникальную иерархию. Число возможных иерархий соизмеримо с числом различных родителей.

Отношение наследования

NURBS-сфера из предыдущего примера имела всего одного предка, собственный узел **transform**. Что если вы захотите изменить ее, пользуясь другими узлами преобразования? Возможно, вам даже захочется поместить эти преобразования в иерархию, с тем чтобы потомки трансформировались своими родителями. Так будет описана *иерархия преобразований*. Сами формы по-прежнему существуют под влиянием преобразований непосредственных родителей. Именно родительские преобразования могут становиться потомками других преобразований. Это дает возможность выстраивать произвольные иерархии узлов **transform**. Концептуально такую иерархию можно считать деревом. На рис. 2.18 представлена **transform-иерархия** простого человекоподобного робота.

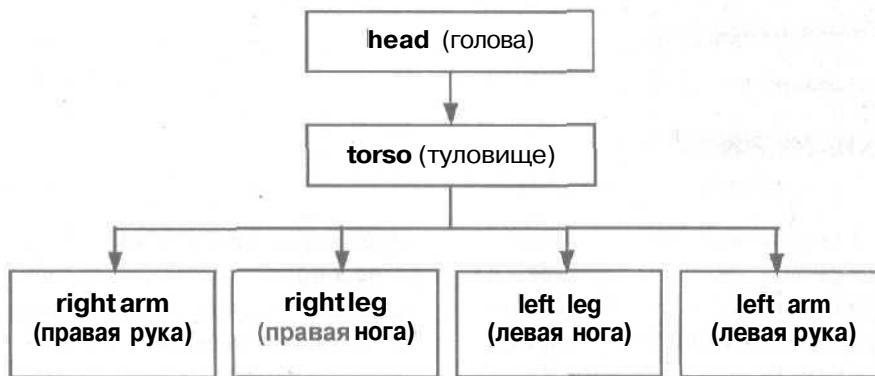


Рис. 2.18. Иерархия человека-робота

Эта иерархия не отражает реального положения дел. Физическая организация робота в трехмерном пространстве отличается от той, что показана на схеме. Голова физической модели может находиться впереди туловища. Та же иерархия в окне **Hypergraph** показана на рис. 2.19.

За каждым из преобразований стоит связанная с ним форма. Двигаясь сверху вниз по серым линиям, соединяющим узлы, можно обнаружить, что в качестве корня дерева в иерархии определен **transform-узел head**. На уровень ниже находится **transform-узел torso**. Еще ниже лежат четыре узла **transform**: **leftArm**, **rightArm**, **leftLeg** и **rightLeg**.

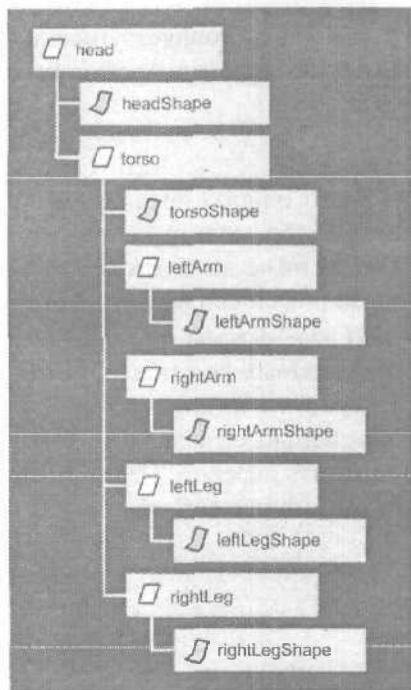


Рис. 2.19. Иерархия узлов

При наличии множественных родительских преобразований окончательное положение формы в мировом пространстве уже не определяется лишь узлом **transform**, относящимся к непосредственному родителю. Перенос объекта в мировое пространство становится результатом действия всех узлов **transform**, лежащих в иерархии выше данной формы. Так формируется *путь преобразования*. Чтобы определить итоговое преобразование из объектного пространства в мировое, начните с преобразования, относящегося к непосредственному предку формы, а затем найдите все преобразования, лежащие на пути к корневому узлу. Когда такое преобразование будет найдено, умножьте его *матрицу* на *матрицу* текущего преобразования. В случае с узлом **rightLegShape** его итоговое преобразование имеет вид:

итоговая матрица преобразования = матрица rightLeg x матрица torso x матрица head.

Эта итоговая матрица **transform** применяется ко всем точкам формы **rightLegShape** для их переноса из объектного пространства в мировое.

В данном примере все узлы **transform** располагали дочерним узлом **shape**. Возможны случаи, когда узел **transform** не имеет потомка **shape**. Фактически *группирующие* возможности Maya реализованы исключительно при помощи узлов **transform**. Выполняя группировку набора объектов, вы заставляете Maya создать новый узел **transform** и поместить в него все выделенные объекты в качестве потомков, т. е. на уровень ниже.

Пути по ориентированному ациклическому графу

Тот факт, что иерархия ОАГ является структурой типа «дерево», означает, что каждый узел ОАГ является *листом* этого дерева. Вам предстоит неоднократно обращаться к конкретному узлу иерархии. Для создания ссылки на тот или иной узел Maya предлагает такое средство, как *путь по ОАГ*. Путь по ОАГ – это точное описание того, как из корневого узла можно перейти в конкретный узел, двигаясь по промежуточным узлам дерева.

Если бы вам пришлось начинать движение с самой вершины дерева и спускаться по его узлам, замечая при этом, какие узлы вы обошли, прежде чем оказались в интересующем вас месте, список вершин в ваших заметках стал бы описанием пути к этому узлу. Итак, каким будет путь по ОАГ к узлу **leftLegShape**? Первым на пути лежит узел **head**, за ним **torso**, затем **leftLeg** и, наконец, **leftLegShape**.

Для обозначения пути к заданному узлу в Maya используется специальная форма записи. Каждое из названий узлов отделяется вертикальной чертой (**|**). Поэтому путь по ОАГ к узлу **leftLegShape**, записанный в нотации Maya, имеет следующий вид:

| head | torso | leftLeg | leftLegShape.

Заметьте, что узлу **head** предшествует пустой родитель. Он указывает на корневой узел сцены – предка всех входящих в сцену узлов. Он не выполняет преобразования форм, лежащих на нижних уровнях, а служит глобальным корневым узлом, потомками которого являются все объекты сцены. Далее следует узел **head** и т. д.

Пространство Underworld

Пространство *Underworld* является тем понятием, о котором мало известно и которое можно легко неверно интерпретировать. Несмотря на то что по-английски его название звучит безрадостно и даже зловеще (*underworld* – преисподня), в действительности оно представляет собой еще один тип геометрического пространства, где вы можете размещать свои объекты. Обычно положение объекта задается точкой в декартовых координатах (*x*, *y*, *z*). Эта точка может присутствовать в любом количестве концептуальных пространств, таких, как

объектное, родительское, мировое и др., однако везде она остается всего лишь точкой.

Поверхность некоторых объектов также можно рассматривать как пространство. Оно отличается тем, что описывается при помощи двух координат (u, v), которые задают точку в *параметрическом пространстве*. Параметрическим пространством обладают не все *объекты*. Типичным объектом, имеющим параметрическое пространство, является *NURBS-поверхность*. *NURBS-поверхность* – это своеобразный «лоскуток», позиция в параметрическом пространстве которого может меняться от $(0; 0)$ в левом нижнем углу до $(1; 1)$ в правом верхнем. Хотя такой вариант наиболее *распространен*, *возможны и другие* пределы изменения параметров «лоскутка». Точка на поверхности определяется координатами (u, v) , лежащими в интервале от 0 до 1. К примеру, центр параметрического пространства имеет координаты $(0,5; 0,5)$. На рис. 2.20 показано параметрическое пространство и его проекция на «лоскуток» NURBS.

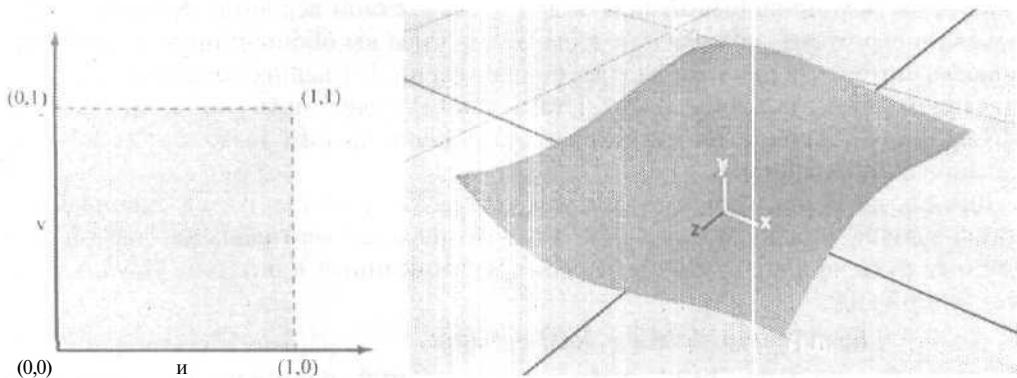


Рис. 2.20. Параметрическое пространство

Так как для вывода на экран объект по-прежнему должен быть задан в трехмерной системе координат, точка поверхности с *параметрическими* координатами (u, v) пересчитывается и в *результате* преобразуется из точки в трехмерной системе. В этой точке объект и изображается. *Преимуществом* параметрических координат является то, что они «сдвигаются» вместе с поверхностью. При изменении формы поверхности положение относительно самой поверхности остается неизменным. К тому же вы можете плавно перемещать *объект по* поверхности, изменяя его *параметрические координаты*, при этом точка будет всегда гарантированно пребывать на этой поверхности. Вот почему *кривые* усечения NURBS-поверхностей, всегда плотно прилегающие к последним, описываются в параметрических координатах.

В следующем примере будет создана плоскость NURBS. Затем мы ее *оживим*, выделив эту плоскость и выбрав пункт главного меню **Modify | Make Live** (Изменить | Оживить). На поверхности чертится кривая. Так как NURBS-плоскость является «живой», изображенная кривая фактически задана в параметрическом пространстве Underworld. Представление этой конфигурации в окне **Hypergraph** показано на рис. 2.21. Обратите внимание на пунктирную линию, соединяющую узел **curve1** категории **transform** с узлом **nurbsPlaneShape1** категории **shape**. Она указывает на то, что объект, лежащий в иерархии ниже NURBS-формы, принадлежит Underworld, а значит, задан в параметрическом пространстве.

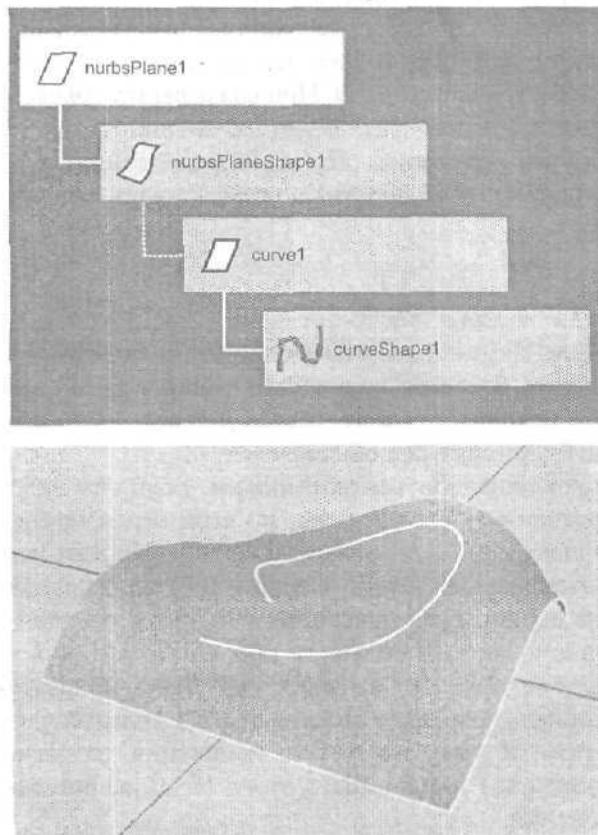


Рис. 2.21. Представление в окне **Hypergraph**: NURBS-плоскость с присоединенной кривой

По существу, пространство Underworld - это еще одно пространство координат, в котором вы можете определять положение объектов, поэтому Maya делает его частью ОАГ. Действительно, форма записи при указании пути по ОАГ к объекту в этом пространстве отличается очень незначительно. Вместо вертикальной черты (|) используется стрелка (->). Обнаружив стрелку в записи пути по ОАГ, знайте, что вы двигаетесь вниз по Underworld-пространству. На практике это означает переход от локального трехмерного пространства объекта к двухмерному параметрическому пространству. Полный путь по ОАГ к узлу формы curve1 в рассматриваемом примере имеет вид

```
| nurbsPlane1 | nurbsPlaneShape1 -> curve1 | curveShape1.
```

Вы можете задать вопрос, имеются ли другие пространства, в которых вы можете размещать объекты. На самом деле есть только два пространства – общепринятое декартово (*x*, *y*, *z*) и параметрическое (*u*, *v*). При связывании объекта с траекторией движения можно говорить о незначительном изменении параметрического пространства. В этом случае положение объекта определяется в одномерном параметрическом пространстве (*U*), простирающемся вдоль кривой траектории.

Экземпляры

Чрезвычайно важной, но иногда неверно воспринимаемой возможностью Maya является работа с экземплярами. Экземпляры становятся очень полезными тогда, когда вам необходимо получить большое количество точных копий заданного объекта. Если вы измените исходный объект, создав нужное число его экземпляров, то и все другие копии будут сразу же обновляться.

Вот пример ситуации, когда могут понадобиться экземпляры. Если вам надо смоделировать яблоню, сплошь усыпанную плодами, по всей вероятности, постройте единственную модель яблока и станете многократно тиражировать ее, развесивая фрукты на ветках. Потенциально можно создать 1000 одинаковых яблок. В результате вы 1000-кратно скопируете одинаковые очертания исходного яблока. Это приведет к потреблению гораздо большего объема памяти и увеличению размера файла сцены. Кроме того, это означает, что Maya придется больше заниматься обработкой данных, поскольку каждое яблоко является отдельным, самостоятельным объектом. К тому же любые изменения, затрагивающие единичное яблоко, будут касаться лишь одного этого объекта. Все остальные яблоки останутся без изменений.

А что если вам захочется изменить исходное яблоко и сделать так, чтобы это отразилось на всех прочих плодах? Что если вы хотите, чтобы каждое яблоко сохраняло форму исходного плода, но стало чуть меньше и было повернуто другим

боком? Этую возможность предоставляют экземпляры. Создавая экземпляр данной формы, вы порождаете новый узел **transform**, который непосредственно указывает на исходный. При отображении в окне просмотра объект имеет те же очертания, но при этом является результатом иного преобразования (переноса, масштабирования, вращения), заданного новым родителем **transform**.

На рис. 2.22 показано представление в окне Hypergraph NURBS-сферы **nurbsSphere1**, результатом копирования которой стал узел **nurbsSphere2**. Ясно, что вторая сфера **nurbsSphere2** имеет свой собственный уникальный узел **shape** с именем **nurbsSphereShape2**. Изменение формы **nurbsSphereShape2** не приведет к каким-либо изменениям исходной формы **nurbsSphereShape1**.

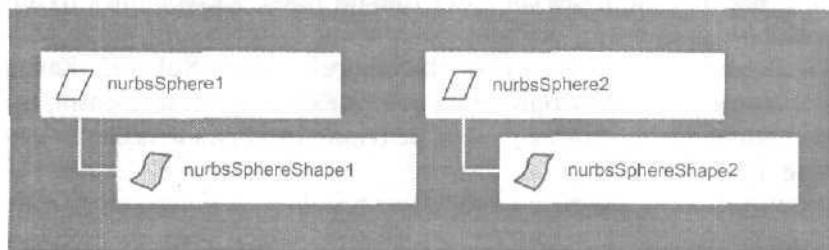


Рис. 2.22. После копирования

Во втором примере, показанном на рис. 2.23, первая сфера служит основой тиражирования, что опять-таки ведет к построению узла **nurbsSphere2** категории **transform**, однако на сей раз вы можете заметить, что новый узел не имеет собственного узла **shape**. Взамен он ссылается на исходный узел **shape** с именем **nurbsSphereShape1**, поэтому любые изменения, которые могут быть в него внесены, будут немедленно отражаться на втором экземпляре.

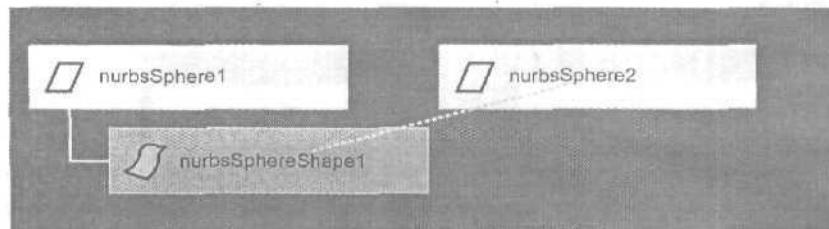


Рис. 2.23. Последтиражирования

Экземпляр - это лишь новый узел **transform**, который ссылается на совместно используемый узел **shape**. А так как он относится к категории **transform**, то вы можете преобразовывать новый экземпляр, как пожелаете.

Преимуществом применения экземпляров является сокращение необходимого объема памяти, поскольку в работе находится только одна копия формы. Все экземпляры пользуются этой единственной формой **согместно**. Если **речь** идет об очень **крупных** и сложных моделях, то подобное решение может дать значительную экономию. Например, батальная сцена с **большим** числом пехотинцев могла бы потребовать чрезвычайно много памяти, если бы каждый солдат располагал своим собственным узлом **shape**. Применение **экземпляров** гарантирует, что в памяти находится только одна модель солдата, которая используется всеми преобразованными экземплярами объекта в составе сцены.

К **сожалению**, с экземплярами связан один тонкий трюк, касающийся путей по ОАГ. Обратите внимание на то, что в примере с тиражированием у вас появились уже два узла **transform**, а именно **nurbsSphere1** и **nurbsSphere2**. Каков же путь по ОАГ к **shape**-узлу **nurbsSphereShape1**? Работая с экземплярами, вы действительно получили два возможных пути, ведущих от корня и заканчивающихся узлом **shape**. Вот они:

```
| nurbsSphere1 | nurbsSphereShape1;
| nurbsSphere2 ] nurbsSphereShape1.
```

Очень важно помнить о том, что, обращаясь к любому **узлу**, можно - при использовании экземпляров - получить несколько путей к нему. Поэтому вам не удастся сослаться на узел, лишь напрямую указывая его имя. Чтобы точно определить, к какому экземпляру вы обращаетесь, нужно пользоваться полным путем к узлу.

Тиражирование можно распространить и на сами **узлы transform**. В окне представления **Hypergraph** на рис. 2.24 показан более **сложный** пример использования экземпляров.

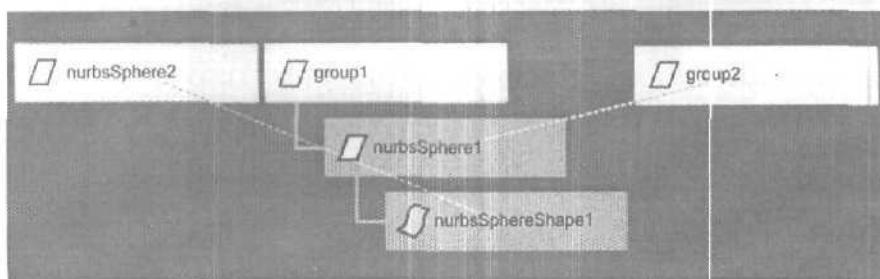


Рис. 2.24. Сложноетиражирование

На основе предыдущего примера, в котором первая сфера **nurbsSphere1** стала основой создания **nurbsSphere2**, исходное преобразование **nurbsSphere1** было помещено в группу. Это привело к созданию узла-предка **group1** категории **transform**. Данное преобразование **group1** было включено в процесс тиражирования, что позволило создать **transform**-узел **group2**. Обратите внимание, что преобразование **group2** ссылается на **transform**-, а не на **shape**-узел **group1**. Это показывает, что вы можете создавать экземпляры и других типов узлов ОАГ, в данном случае - узлов **transform**.

Последний пример еще раз заостряет внимание на важности непрестанного обращения к экземплярам в ходе проектирования приложения. Не забывайте о возможности существования множества путей по ОАГ к данному узлу, и это убережет вас от попадания в ловушку, когда, ссылаясь на корректный узел, вы можете спускаться по неверному пути. Подобного рода недоразумения часто повторяются в программном коде, так что их результатом могут стать трудно находимые ошибки. К счастью, и язык MEL, и интерфейс API содержат функции для определения того, участвует данный узел в тиражировании или нет, поэтому их нужно без ограничений внедрять в исходный код приложений.

2.2.10. Обновление графа зависимости

Получив прочные знания методов использования различных компонентов DG, вы можете перейти к изучению самой важной и насущной темы, а именно: как происходит обновление DG при изменении значений. Эта тема столь важна потому, что, не поняв ее, вы никогда не сможете отследить причины типичных проблем, связанных с обновлением узлов. Зачастую DG может становиться источником неочевидных проблем, когда, с точки зрения разработчика, он обновляется неожиданным образом. Это неправильное понимание может вылиться во многие часы, если не дни, потраченные в попытках выяснить, почему приложение отказывается работать.

Основная причина, по которой эта область вызывает так много путаницы, на самом деле достаточно проста. Создаваемые вами узлы совершенно не управляют тем, когда или каким образом они будут обновляться. Полный контроль над этим берет на себя Maya. Узел - лишь малозаметный персонаж большого романа. В основе **своей**, узел – это лишь функция **compute**. От него требуется вычислить некоторые **выходные** значения при наличии ряда **входных**. Узел никогда не знает о том, зачем он должен вычислять эти значения, а просто делает это. Поэтому, как разработчик, вы в действительности не управляете тем, когда и как обновляются ваши узлы. Помня об этом, продолжим наше движение к самому ядру DG и выясним, как именно происходит обновление.

Двунаправленная модель

Как упоминалось ранее, DG, по крайней мере, на концептуальном уровне, работает в соответствии с моделью потока данных. Кажется, что исходные данные приходят на один конец устройства, а после обработки покидают другой конец. Хотя эта умозрительная модель облегчает понимание возможностей соединения всех узлов и создания автомата, способного перерабатывать данные, фактическая реализация DG более элегантна.

На самом деле в основу графа DG положена *дву направленная модель*². Эта модель указывает на возможность *проталкивания* определенной информации по сети. Вместе с тем, информация из сети может и *вытягиваться*. Различие между этими операциями состоит в том, что, *проталкивая* данные, вы начинаете с текущего узла и ваши действия распространяются на все узлы, связанные с ним как его *выходы*. При вытягивании информации из узла действие запроса распространяется на все узлы, которые соединены с ним как его *входы*. Необходимость различать эти режимы информационных потоков обусловлена стремлением к эффективности. Двунаправленная модель предусматривает гораздо более эффективный механизм обновления по сравнению с чистой моделью потока данных.

На рис. 2.25 показаны пять взаимосвязанных узлов. Каждый из них может обрабатывать входные данные и получать результат, который отправляется на выход. В данном примере узел A генерирует некоторые *данные*, которые затем по выходным соединениям передаются узлам B и C. Узел C непосредственно не порождает никаких *данных*, а принимает входные данные, поступающие от узла A, и обрабатывает их перед отправкой результата узлам D и E.

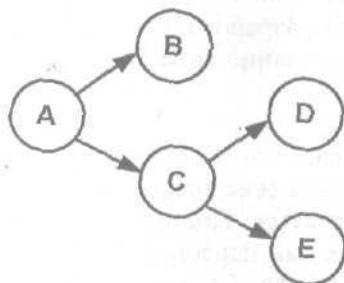


Рис. 2.25. Связанные узлы

² Дословно, модель «тяни – толкай» (push-pull model). Отмечая своеобразие английского термина, воспользуемся и мы этой метафорой для описания работы модели. - Примеч. перев.

Чтобы продемонстрировать различие между подходами на основе потока данных и двунаправленной модели, начнем с данных, находящихся в узле A, и посмотрим, как они распространяются по этой простой сети узлов.

При использовании подхода на основе потока данных *вы* начинаете с узла A, где происходит генерация данных. Затем эти данные передаются узлам B и C, где их обработка продолжается. Узел C передает результат обработки узлам D и E, которые, в свою очередь, сами выполняют некоторую обработку. Из этого описания ясно, что данные начинают свое движение в самом линии узле и распространяются по сети, причем каждый узел на этом пути реализует некоторую форму обработки данных. Когда узел A генерирует новые данные, вся сеть обновляется. По завершении обработки узлы B, D и E содержат окончательные результаты.

Теперь представьте *себе*, что обработка, которую выполняет каждый узел, чрезвычайно сложна. Скажем, к примеру, что каждомуциальному узлу на обработку входных данных и выдачу выходной информации требуется коло часа. Это означает, что если узел A генерирует некоторые новые данные, то до момента, когда все концевые узлы (B, D, E) будут содержать окончательные результаты, пройдет, в общей сложности, четыре часа.

А если вместо получения результатов B, D и E вы хотите получить только результат узла B? К сожалению, в этой модели все данные проходят по всем узлам. Это значит, что даже если вы хотите получить только результат узла B, вы должны по-прежнему ждать, пока все узлы закончат свою работу. Сеть узлов в этом примере намеренно упрощена, но можно легко представить более сложную сеть и те проблемы производительности, которые создаст применение этого подхода.

В идеале, хотелось бы выбрать конкретный узел и выполнить минимальный объем работы в сети для его обновления. Вот где двунаправленная модель становится очень полезной. Она разбивает процесс обновления сети на два четко различимых этапа. Первый этап - это распространение флага состояния, второй - фактическая обработка данных. Первый этап - «тяга», второй - «толкание».

Представьте, будто в каждом узле содержится флаг. Этот флаг, который можно назвать флагом необходимости обновления *needsUpdating*, указывает, что выход данного узла устарел, а потому требует обновления. Если флаг установлен, узел вычисляет свой выход заново. Если флаг не установлен, он просто выдает свое текущее выходное значение, не выполняя никаких повторных расчетов. Применение этого флага дает непосредственный прирост скорости. Если узел не требует обновления, всю связанную с ним обработку можно пропустить и передать текущий результат следующему узлу. Поскольку обновление сети состоит

из двух этапов, узлы на схеме отображаются в одном из двух состояний: обновленные или требующие обновления. Рис. 2.26 иллюстрирует способы изображения узлов в различных состояниях.

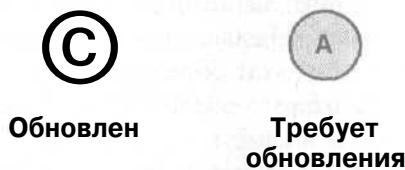


Рис. 2.26. Состояние узлов

Первый этап работы двунаправленной модели заключается в проталкивании флага статуса по всем узлам, участвующим в соединениях. Когда узел А выдает новое значение, данные не пересыпаются связанным с ним узлам; Однако начинается распространение флага состояния. Флаг указывает на то, что узлы требуют обновления. Первыми получат флаг и обновят свое состояние узлы В и С, затем флаг будет передан дальше. Данный этап охватывает все узлы, которые участвуют в соединениях. Наконец, как показано на рис. 2.27, эти узлы оказываются в новом состоянии.

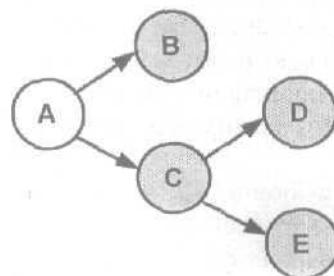


Рис. 2.27. После изменения узла А

Обратите внимание, что узел А не помечен как требующий обновления. Так как он выдает исходные данные, то и содержит наиболее актуальное значение, Потенциально устаревшие данные как раз содержат все остальные узлы, поскольку они напрямую или косвенно зависят от узла А.

В каждом из этих узлов не было никакой обработки данных, поэтому первый этап может быть выполнен очень быстро. Узлы будут оставаться без изменений до тех пор, пока вы не запросите содержащиеся в них данные. Что

если вы хотите узнать полученный в результате выход узла В? Определение результата В сравнимо с вытягиванием информации из узла. Стало быть, процесс вытягивания данных подобен запросу выходного значения узла.

Однако запрос выхода конкретного узла может неявно привести к образованию цепочки обновлений узлов. Этот процесс гарантирует правильность результата, выдаваемого данным узлом. Когда узел В по запросу должен **возвратить** собственный выход, он проверяет флаг **needsUpdating**. Поскольку тот установлен, узел знает, что его текущий выход устарел и потому должен быть вычислен заново. Он берет входное значение, в данном случае выход узла А, и повторно рассчитывает результат. Теперь выход обновлен, так что флаг **needsUpdating** сбрасывается. Сброс флага указывает на корректные выходные данные узла. После этого этапа «тэги» граф выглядят **так**, как показано на рис. 2.28.

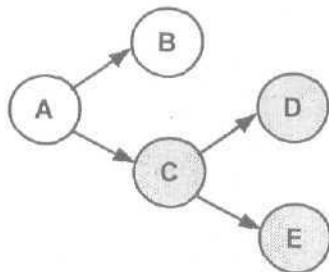


Рис. 2.28. После обновления узла В

Заметьте, что обновлен исключительно узел В. Коль скоро вы запросили лишь его выход, то и обновлен только он один. Узлы С, Д и Е не выполняли обработку данных, и их флаги **needsUpdating** все еще установлены. Двунаправленная модель позволяет выбрать единственный узел и запросить его **результат** без необходимости обновления узлов, которые не связаны с выбранным. Такой подход к обновлению исключительно тех узлов, которые вносят свой вклад в окончательный результат, - вот то, что делает двунаправленную модель более эффективной в сравнении с моделью потока данных.

Теперь посмотрите, что происходит при запросе результата узла D. Механизм его обновления в точности такой же, что и для узла В, однако имеются некоторые дополнительные запросы. Этап, реализуемый каждым узлом с целью самообновления, ничем не отличается от показанного прежде. Узел проверяет, установлен ли флаг **needsUpdating**, и если да, находит новое выходное значение. Если флаг не установлен, узел просто выдает свой нынешний выход. В данном случае флаг **needsUpdating** не установлен, поэтому можно гарантированно

предположить, что текущий выход узла является наиболее актуальным. Все узлы проходят этот несложный процесс, если запрашивается их выходное значение. Тот же процесс происходит **даже** в более длинных цепочках взаимосвязанных узлов.

Приняв запрос выходного значения, узел D обнаруживает, что ему требуется обновление. Тогда он запрашивает входные данные в узле C. Узел C, в свою очередь, видит, что ему тоже необходимо обновление. Он запрашивает свои входные данные в узле A. Флаг `needsUpdating` узла A не установлен, поэтому узел просто возвращает свое **текущее выходное значение**. Узел C принимает его и обновляется. Далее он сбрасывает свой флаг `needsUpdating` и выдает **полученный результат**. Затем узел D принимает новый вход и вычисляет новый выход, **после** чего также сбрасывает свой флаг `needsUpdating`. Как выглядит график в конце этапа «тяги», показано на рис. 2.29.

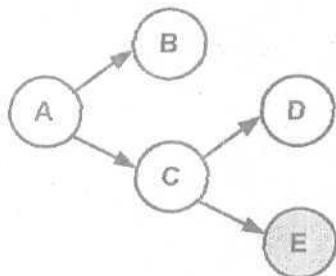


Рис. 2.29. После обновления узла D

Обратите внимание на то, что узел E не обновлен. Из-за отсутствия явных или неявных соединений с узлом D он не получил запроса на обновление. С другой стороны, узел C обновлен потому, что он соединен с узлом D и требовал **обновления**.

При получении запроса выхода узел запрашивает свои входные значения. Если входные узлы сами требуют **обновления**, процесс продолжается. Это может привести к появлению цепочки обновлений, затрагивающих некоторые или все узлы, которые в нее входят.

Обновление на практике

Теперь мы более внимательно рассмотрим реализацию двунаправленной модели в системе Maya. В этом разделе, содержащем полное описание принципов работы DG, вам потребуются все полученные ранее знания. Граф зависимости, представленный выше, показан на рис. 2.30.

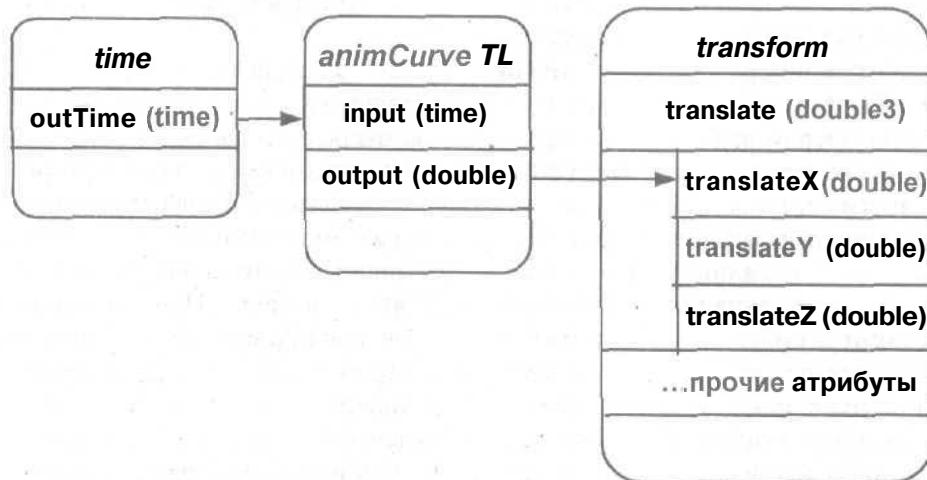


Рис. 2.30. Узлы и соединения графа зависимости

С каждым атрибутом в Maya связывается флаг, который функционирует точно так же, как признак **needsUpdating**, обсуждавшийся в предыдущем разделе. В Maya этот флаг носит название *бита изменений*. Бит изменений устанавливается, когда значение атрибута устарело и требует обновления. Когда бит изменений не установлен, атрибут содержит самое актуальное значение и, следовательно, не требует повторного вычисления. Если бит изменений атрибута установлен, то атрибут, как и на предыдущих схемах, будет отображаться на сером фоне.

В целях нашего обсуждения предположим, что граф DG уже обновлен. Это означает, что все атрибуты содержат актуальные значения, и бит изменений ни **одного** атрибута не установлен. Это как раз та *ситуация*, что показана на схеме рис. 2.30.

Теперь переставим бегунок текущего времени на другой кадр. Так как атрибут **translateX** анимирован, то вы можете предположить, что смена текущего времени приведет к обновлению атрибута; именно это и произойдет. Подход к решению этой задачи со стороны Maya не так прост, как может показаться.

При перемещении бегунка текущего времени единственный глобальный узел **time** с именем **time1** также изменился. Его атрибут **outTime** был обновлен и получил новое значение времени. Уведомление об этом путем установки соответствующих битов изменений рассыпается всем атрибутам, связанным с **outTime**. Заметьте, что бит изменений атрибута **outTime** установлен не будет, поскольку

Maya только что присвоила ему новое значение. Он содержит самые последние данные, а потому не будет помечен как измененный.

В данном примере атрибут **outTime** связан с атрибутом **input** узла **animCurveTL**. Бит изменений этого атрибута **устанавливается**. Следующий шаг связан с просмотром **атрибутов**, которые, в свою очередь, соединены с атрибутом **input**, поскольку им также требуется отправить уведомление. Атрибут **input** не имеет других соединений, поэтому уведомление дальше не распространяется. Однако он **влияет на** атрибут **output**. Это отношение **было** задано в тот момент, когда узел пополнился описанием воздействия атрибута **input** на **output**, для чего послужил вызов функции **attributeAffects(input, output)**. При изменении атрибута **input** атрибут **output** оказывается под его **влиянием**, а значит, требует повторного вычисления, так что бит изменений атрибута **output** устанавливается. Затем следует просмотр всех исходящих соединений атрибута **output**. С ним соединен атрибут **translateX**, поэтому его бит изменений также **устанавливается**. Так как атрибут **translateX** не имеет прочих соединений и не влияет на другие атрибуты своего узла, распространение бита изменений на этом завершается. Этап толкания окончен.

На рис. 2.31 показано состояние графа DG после описанного *распространения бита изменений*. В этот момент ни один из атрибутов, затронутых изменением **outTime**, не обновлен. Они все лишь помечены как измененные. Никакой дальнейшей обработки Maya не выполняет.

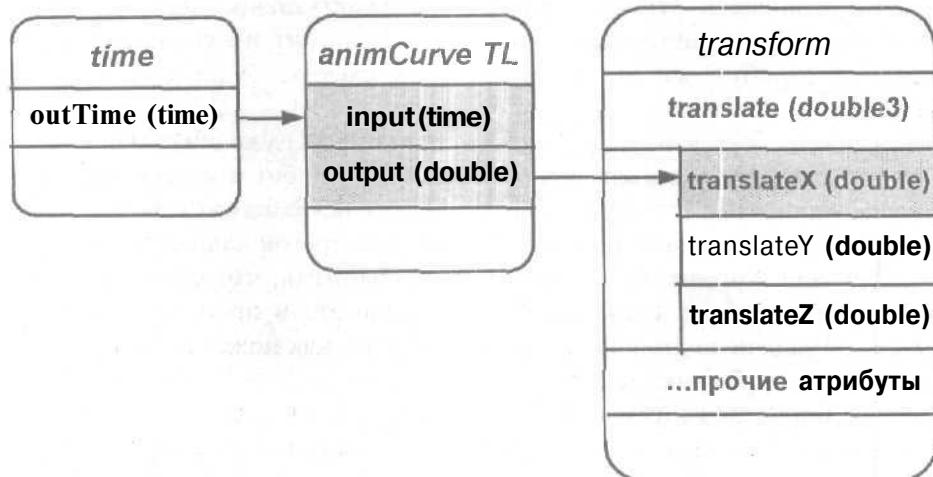


Рис. 2.31. После распространения бита изменений

При определенных обстоятельствах Maya сообщает графу DG о необходимости его обновления. Такими обстоятельствами, к примеру, может быть запрос значения атрибута, происходящий при запуске процесса рендеринга или отображении редактора **Attribute Editor** либо окна **Channel Box**. Есть много других причин обновления DG, но самой распространенной является ситуация, требующая обновления экрана. Косвенным результатом перестановки бегунка времени является отправка запроса перерисовки экрана, адресованного Maya. Прежде чем приступить к перерисовке всех объектов сцены, Maya требует от графа DG самообновления, так как некоторые из объектов, возможно, анимированы, а потому будут иначе выглядеть в другое время.

При обновлении графа зависимости Maya стремится к максимальной эффективности. Она запрашивает самые последние значения только тех узлов, которые на самом деле требуют обновления. Процесс начинается от узла, являющегося корнем ОАГ. Затем делается спуск «в глубину» по дереву узлов ОАГ, при этом Maya запрашивает признак видимости каждого узла. Если узел невидим, Maya игнорирует его и всех его потомков, поскольку признак видимости наследуется. Видимые узлы получают от Maya запрос, указывающий на необходимость обновления. Поэтому, когда в данном примере Maya достигает узла **transform**, она пытается обновить матрицу преобразования, что позволило бы поместить форму в нужной точке мирового пространства. Результатами запроса матрицы преобразования неявно становятся запросы последних знаний атрибутов **translate**, **scale** и **rotate**. Запрашивая значение **translateX**, вы начинаете этап «тяги» в ходе процесса обновления. Так как атрибут **translateX** имеет установленный бит изменений, его нужно обновить.

Коль скоро атрибут определяет свое значение на основании входного соединения, он запрашивает значение входа. Узел **animCurveTL** отвечает на этот запрос. Он обнаруживает, что запрашиваемый атрибут **output** изменен и требует обновления. Значение этого атрибута есть результат функции **compute**, поэтому производится ее вызов. Далее функция **compute** запрашивает значение атрибута **input**, необходимого ей для выполнения вычислений. Атрибут **input** помечен как измененный и поэтому нуждается в обновлении. Его значение основано на атрибуте **outTime**, который опять-таки запрашивается. Атрибут **outTime** анализирует свое состояние и обнаруживает, что его бит изменений не установлен, так что он просто возвращает значение, которое содержит в настоящем время.

Теперь цепочка запросов «сворачивается», и ряд запросов выполняется от конца к началу. Атрибут **input** принимает значение от **outTime** и сбрасывает собственный бит изменений. Функция **compute** принимает значение атрибута **input** и вы-

числяет значение **output**. Вслед за обновлением атрибута его бит изменений, как всегда, сбрасывается. Атрибут **output** передается на вход **translateX**. Возврат значения атрибута **translateX** означает выполнение исходного запроса.

Этот пример демонстрирует тот факт, что простой запрос значения атрибута, вероятно, может привести к длинной цепочке аналогичных запросов, каждый из которых, возможно, потребует дальнейших вычислений, выполняемых до тех пор, пока не будет получен окончательный результат. Объем работы, которой требует каждый запрос, напрямую зависит от числа атрибутов, расположенных далее в цепочке, а также от того, сколько из них имеют установленные биты изменений. Если бы в данном примере бит изменений атрибута **translateX** был сброшен, то никакая обработка бы не производилась. Действительно, если теперь вы запросите значение **translateX**, атрибут просто возвратит находящееся в нем значение, так как он уже не считается измененным.

Кроме специального запроса **Maya**, адресованного DG и указывающего на необходимость самообновления, граф изменений может частично рассчитываться путем обычных запросов текущего значения атрибута. Распространение процесса обновления будет происходить точно так же, как описано ранее. Вместо атрибута **translateX** вы всего лишь запросите иной атрибут любого другого узла. Однако шаги, направленные на получение актуального значения этого атрибута, останутся в точности теми же.

Обновление при анимации по ключевым кадрам

Занимательным побочным эффектом двунаправленной модели является возможность простого построения интерактивной системы анимации по ключевым кадрам. Системой анимации по ключевым кадрам называется система, где анимация объектов описывается серией анимационных кадров, заданных в различные моменты времени. Если для данного момента времени ключевой кадр отсутствует, то новое значение определяется путем интерполяции кадров, расположенных до и после текущего момента. Весьма интересно то, что этот побочный эффект применения битов изменений позволяет интерактивно передвигать или преобразовывать объект в 3D-пространстве, а затем, когда все готово, установить ключевой кадр для блокировки этого преобразования текущего кадра. Такая возможность очень важна для работы с объектами, анимация которых выполняется в интерактивном режиме.

При первом рассмотрении эта функция может показаться не таким уж «большим делом», однако подумайте над тем, что атрибут **translateX** управляет расположенным выше узлом анимационной кривой **animCurveTL**, и ответьте, как вообще

вы можете двигать объект, не изменяя анимационную кривую явным образом? Как можно взять объект, для которого уже созданы ключевые кадры, и в интерактивном режиме перемещать его по окну просмотра? Не должно ли значение **translateX** заимствоваться из узла анимационной **кривой**, с которым оно связано? Разве анимационная кривая перестает работать во время диалога с пользователем?

Ответ очень прост, и он показывает еще одно, отчасти малозаметное, следствие применения двунаправленной модели. Чтобы познакомиться с тем, какие события происходят «за кулисами», начните с такого упражнения в среде Maya.

1. Загрузите сцену **SimpleAnimatedSphere.ma**.
2. Щелкните по кнопке **Play**.
Сфера начнет двигаться вдоль оси **x**.
3. Нажмите на кнопку **Stop** (Остановить).
4. Выделите объект **nurbsSphere1**.
5. Откройте окно **Hypergraph** и щелкните по кнопке **Up and Downstream Connections**. То, что вы увидите на экране, будет напоминать рис. 2.32.
Узел анимационной кривой **nurbsSphere1_translateX** управляет атрибутом **translateX** узла **transform** с именем **nurbsSpherel**. Именно благодаря этому сфера и движется вдоль оси **x**.
6. Закройте окно **Hypergraph**.
7. Установите бегунок времени на 27-й кадр.
8. Убедитесь в том, что вы видите окно **Channel Box**.
9. Выделите объект **nurbsSpherel**, затем выберите инструмент **Move** (Перемещение). Передвиньте сферу в произвольную точку вдоль оси **x**.
Во время перемещения сферы проследите за значением ее атрибута **Translate X**. Как такое возможно? Параметр трансляции сферы по оси **x** берется из узла кривой анимации, но при этом вы можете его изменять. Преобразование сферы почему-то игнорирует анимационную кривую.
10. Перейдите к 28-му кадру.
Сфера быстро возвращается в исходное положение, являющееся результатом анимации 28-го кадра.
11. Перейдите к 27-му кадру.
Предыдущее положение, в которое вы перенесли сферу, потеряно. Сфера заимствует значение **Translate X** из исходной анимационной кривой.

12. Переместите сферу в новое положение вдоль оси *x*. Нажмите клавиши **Shift+w**.
27-й кадр становится ключевым.
13. Щелкните по кнопке **Play**.
Сфера анимируется, как и прежде, но теперь в 27-м кадре она перемещается в заданное вами новое положение. Очевидно, что перенос задан анимационной кривой. Ключевой **кадр**, который вами установлен, **теперь** учитывается именно так, как и ожидалось.



Рис. 2.32. Окно Hypergraph

Что это за магия в закулисных действиях **Maya**, которая позволяет в диалоговом режиме передвигать сферу даже тогда, когда ее положение управляетяется кривой анимации? По правде говоря, никакой магии нет. Maya по-прежнему выполняет те шаги обновления DG, речь о которых шла до сих пор.

Проследим эти шаги еще раз, но теперь будем наблюдать за тем, что происходит с каждым из атрибутов графа DG. Когда сцена загружена, Maya сразу же рассчитывает DG. В результате этого после обновления биты изменений всех атрибутов сбрасываются. По нажатию кнопки **Play** Maya увеличивает номер текущего кадра. Соответственно, с каждым новым **кадром** атрибут **outTime** изменяется, а биты изменений передаются другим **атрибутам** так, как описано ранее. При переходе от одного кадра к другому сцена нуждается в перерисовке, поэтому **Maya** требует от DG, чтобы он рассчитал себя **заново**. Выполняется запрос **translateX**, что в результате неявно порождает цепочку обновлений узлов, затронутых изменениями. Это в точности повторяет процесс, объяснение которого приведено в предыдущем разделе.

Воспроизведение было остановлено на 27-м кадре, когда вы передвинули сферу вдоль оси *x*. На практике эта операция означает, что входящему в состав сферы атрибуту **translateX** присвоено значение, полученное при интерактивном передвижении. Поскольку этот атрибут, как и **outTime**, установлен явно, его бит изменений автоматически сбрасывается. Подобный непосредственный сброс бита изменений является ключом к пониманию **того**, почему возможно **интерак-**

тивное перемещение сферы. Теперь на основании того, что бит изменений атрибута **translateX** не установлен, делается предположение, что атрибут имеет **текущее, обновленное значение**. Да, он действительно содержит значение, полученное при взаимодействии с пользователем, однако на самом деле должен иметь значение, взятое из атрибута **output** кривой анимации. Почему атрибут не принимает **этот** значение? При перемещении объекта в интерактивном режиме Maya получает множество запросов **обновления** экрана. Как всегда, она требует от **DG** самообновления, предшествующего перерисовке. Maya спускается вниз по **DG**, и значение **translateX** запрашивается повторно. Однако на этот раз бит изменений атрибута **translateX** сброшен, поэтому запрос значений всех его входных атрибутов выполнен не будет. При таком положении дел анимационная кривая никогда не получит запросов своего выходного значения, поскольку текущее значение атрибута **translateX**, т. е., результат перемещения в **диалоговом режиме**, является, по предположению, наиболее актуальным.

Далее вы перешли к 28-му **кадру**, который стал текущим. Сфера вдруг быстро вернулась на свое место, определяемое кривой анимации. Случилось же то, что, перейдя к другому кадру, вы неявно сменили текущее время. Значение **outTime**, как и прежде, изменилось, и в результате все атрибуты, которых это касается, включая **translateX**, установили собственные биты изменений. Значение **translateX** теперь помечено как измененное, а потому, как и в первоначальном варианте с поступлением запроса значения, атрибут должен получить соответствующее значение через другие узлы, которые вызвали обычную серию обновлений атрибутов.

Итак, возможность интерактивно передвигать ранее **анимированный** объект является прямым результатом сброса бита изменений атрибута **translateX**. Так как бит изменений был не установлен, дальнейшее обновление не **потребовалось**, а кривая анимации так и не получила запрос выходного значения. Использовавшееся значение было явно установлено при интерактивном передвижении.

В этом примере особое значение снова придается тому, что в определенных обстоятельствах **DG** может вести себя непредсказуемо. В действительности, он всегда следует одной и той же **процедуре** и, таким образом, его поведение всегда прогнозируемо. При запросе значения атрибута выполняется проверка бита изменений. Если атрибут не изменен, то текущее значение, которое он содержит, сразу же выводится. Если бит изменений установлен, атрибут предпринимает попытку получить самое последнее значение. Результатом этого может стать повторное вычисление атрибута при помощи функции **compute** или получение значения от одного из других атрибутов, с которыми он соединен. Этот процесс

может рекурсивно повторяться в отношении всех затронутых изменениями атрибутов, продолжаясь до тех пор, пока не будет получено окончательное значение. После сохранения нового значения в атрибуте его бит изменений сбрасывается, и при повторном запросе атрибут **может** просто выдать хранящееся в нем значение и тем самым избежать множества возможных повторных расчетов.

Двунаправленная модель представляет собой лишь один из многих внутренних механизмов, используемых **DG** во избежание выполнения ненужных вычислений. Некоторые из менее критичных механизмов будут изучены в следующих главах. Понимание того, как при помощи двунаправленной модели происходит обновление **DG**, несомненно, поможет вам избежать большого числа проблем при программировании Maya.

3. Язык MEL

3.1. Введение

Встроенный язык MEL (Maya Embedded Language) - наиболее простой и доступный интерфейс программирования Maya. Не зная языка MEL, вы пользуетесь им с того самого момента, как только открыли эту среду. Выделение объекта или отображение диалогового окна - прямой результат выполнения команды MEL. На самом деле, MEL управляет всем *графическим пользовательским интерфейсом (GUI)* Maya. При помощи этого языка можно обращаться практически к любой функции пакета. Читая данную главу, вы *будете* все лучше и лучше ощущать, насколько глубоко влияние MEL на все аспекты работы Maya, *и*, таким образом, *поймете*, когда и как можно обращаться к функциям Maya и управлять ими.

К счастью, MEL - относительно простой в изучении язык программирования. Приведем пример команды MEL, предназначеннной для создания сферы:

```
sphere;
```

Так как MEL является *языком сценариев*, то вы можете просто записать команду, а потом сразу же ее выполнить. Языки сценариев не требуют обычного этапа *компиляции и сборки*, необходимого в других языках программирования, подобных C и C++. Это означает, что вы можете быстро начать эксперименты и немедленно увидеть полученный результат. Если сценарий не будет работать, его можно быстро изменить и запустить снова. При использовании MEL весь циклический процесс реализации и тестирования ваших идей становится гораздо более простым и оперативным.

По вашему усмотрению сценарий на языке MEL может быть сколь *простым*, столь и *сложным*. Для решения часто повторяющихся, распространенных задач вы можете написать простой сценарий, а затем выполнить его по нажатию кнопки. Кроме *того*, можно составлять сценарии для решения очень сложных задач, а также для создания улучшенных вариантов *GUI*. Способность одновременно справляться с простыми и сложными задачами, избавляющая вас от *необходимо*

ности изучать другие языки программирования, относится к числу самых сильных сторон **MEL**.

Обладая прочными базовыми знаниями в области программирования на **MEL**, вы сможете управлять практически всеми аспектами функционирования Maya, а также разрабатывать инструменты и функции, которые сделают вашу работу более быстрой и продуктивной. Слегка поэкспериментировав, вы обнаружите, что даже небольшая порция MEL неплохо помогает в решении ежедневных задач в среде Maya.

Если вы знакомы с программированием на C, обратитесь к Приложению В «**MEL** для программистов на языке C», где приведен полный перечень отличий между этими языками. К счастью, **MEL** и C имеют много общего, поэтому хорошее понимание языка C определенно поможет вам в изучении MEL.

3.1.1. «За кулисами»

Пользуясь интерфейсом Maya, вы косвенно инициируете запуск команд **MEL**, которые, в свою очередь, делают конкретную работу. В ответ на ваши щелчки мышью и выбор пунктов меню Maya негласно запускает команды и сценарии MEL. Эти операции можно даже увидеть, вооружившись редактором **Script Editor** (Редактор сценариев).

Так как любое взаимодействие с графическим интерфейсом Maya происходит посредством языка **MEL**, то логично предположить, что многие из таких повседневных задач можно автоматизировать, записав эти операции, а потом воспроизводя их. Несмотря на то что Maya не предусматривает прямых средств записи всех сделанных вами нажатий клавиш и щелчков мышью, она может показать полученные в результате команды в окне **Script Editor**. Находясь в нем, вы можете выделять команды, а затем копировать и вставлять их в свои собственные сценарии. По сути дела, они могут использоваться как основа ваших личных настроек.

Так что же Maya делает «за кулисами»? Смотрите.

1. Выберите пункт меню **File | New Scene** (Файл | Новая сцена).
2. Откройте **Script Editor**.
3. В главном меню **Script Editor** выберите пункт **Edit | Clear All** (Правка | Очистить все).
4. В главном меню Maya выберите пункт **Create | Nurbs Primitives | Sphere** (Создать | Примитивы NURBS | Сфера).

На сцену помещается сфера, а в редакторе **Script Editor** отображаются следующие строки:

```
sphere -p 0 0 0 -ax 0 1 0 -ssw 0 -esw 360 -r 1 -d 3 -ut 0 -tol 0.01  
-s 8 -nsp 4 -ch 1;  
objectMoveCommand;
```

Выбор пункта меню влечет за собой выполнение этих команд MEL. Вызваны две команды, первая из которых создает сферу (с данным набором параметров), а вторая в качестве текущего инструмента устанавливает инструмент **Move**. Как упоминалось ранее, MEL применяется при любом взаимодействии с использованием GUI. Здесь перечислены команды, которые были выполнены при выборе пункта меню. А как насчет команд MEL, которые выполняют все операции, связанные с интерфейсом пользователя?

5. В меню **Script Editor** выберите пункт **Edit | Clear All**.
6. Выберите пункт меню **Script | Echo All Commands** (Сценарий | Эхо-повтор всех команд).
7. В меню Maya выберите пункт **Create | Nurbs Primitives | Sphere**.

Так как теперь в **Script Editor** можно наблюдать эхо-повтор всех команд, то и количество выведенных команд возросло. Показаны все выполненные вами операции с интерфейсом пользователя. Каждый выбор пункта меню приводит к запуску серии команд MEL.

```
editMenuUpdate MayaWindow|mainEditMenu;  
CreateNURBSSphere;  
performNurbsSphere 0;  
sphere -p 0 0 0 -ax 0 1 0 -ssw 0 -esw 360 -r 1 -d 3 -ut 0 -tol 0.01  
-s 8 -nsp 4 -ch 1;objectMoveCommand;  
// Result: sphere -p 0 0 0 -ax 0 1 0 -ssw 0 -esw 360 -r 1 -d 3 -ut 0  
-tol 0.01 -s 8 -nsp 4 -ch 1;objectMoveCommand //  
autoUpdateAttrEd;  
editMenuUpdate MayaWindow|mainEditMenu;
```

Хотя точный смысл команд понять нелегко, важно запомнить то, что почти все операции, производимые вами в среде Maya, на самом деле являются конечным результатом выполнения ряда команд языка MEL.

8. Чтобы отключить эхо-повтор команд, выберите пункт **Script | Echo .411 Commands**.

Такое использование редактора **Script Editor** с включенной опцией **Echo All Commands** - очень хороший способ узнать, какие команды и процедуры MEL вызываются при выполнении данной операции. Если вы хотите произвести подобную операцию в своем собственном сценарии, включите эхо-повтор и посмотрите, какие команды MEL были запущены. Затем их можно скопировать и вставить в собственный сценарий.

3.2. Язык программирования MEL

Этот раздел представляет собой введение в язык программирования, а также подробно рассказывает о синтаксисе и конструкциях языка.

3.2.1. Команды

При выполнении той или иной команды языка MEL фактически производится вызов одной из функций **Maya**, написанных на языке C++. Когда, пользуясь средствами пользовательского интерфейса, вы создавали сферу, то неявно обращались к команде Maya с именем **sphere**, которая фактически строила сферический объект. В главе, посвященной API на основе C++, вы узнаете, как создавать свои собственные команды, сейчас же мы сосредоточим свое внимание на изучении множества встроенных команд **Maya**.

Запуск команд

Среда **Maya** содержит довольно много различных областей, где можно вводить и выполнять команды на языке MEL. В зависимости от ваших потребностей некоторые из этих областей будут более удобными, чем другие.

1. Выберите пункт меню **File | New Scene** или нажмите клавиши **Ctrl+n**. Еще одним элементом интерфейса, где вы можете выполнять команды MEL, является командная строка **Command Line**. Она представляет собой одностороннее текстовое поле ввода. Это прекрасное место записи односторонних команд, не требующее открытия **Script Editor**. Страна **Command Line** расположена в нижнем левом углу и показана на рис. 3.1. Если ее не видно, выберите в главном меню пункт **Display | UI Elements | Command Line** (Отобразить | Элементы интерфейса пользователя Командная строка).
2. Щелкните по командной строке, а затем наберите следующий текст:
`sphere`

«Горячей клавишей», принятой по умолчанию для перевода курсора в командную строку, служит клавиша обратной кавычки (`).

3. Нажмите клавишу Enter.

На сцене появится **сфера**, имеющая параметры по умолчанию. Справа от **Command Line** расположена область **Command Feedback** (Отклик команд). В ней отображается результат последней выполненной команды. В данном случае здесь будет выведен следующий текст:

Result: nurbsSphere1 makeNurbSphere1

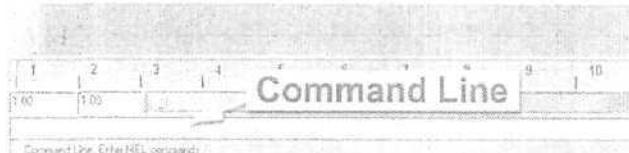


Рис. 3.1. Страна Command Line

Хотя строка **Command Line** идеально подходит для запуска кратких и лаконичных команд, обычно большинство ваших экспериментов будет проходить в редакторе **Script Editor**, поскольку в нем вы сможете запускать последовательность команд, записывая каждую из них на отдельной строке. То же самое можно делать и в **Command Line**, однако при использовании более сложных операторов это может сбить вас с толку, так как все операторы будут записаны в одной строке.

Окно **Script Editor**, показанное на рис. 3.2, состоит из двух основных частей. Верхняя, серая область **History Panel** (Панель истории команд) является местом вывода результатов команд MEL. Нижняя, белая область **Command Input Panel** (Панель ввода команд) предназначена для набора команд. Размер окна может изменяться, так что вы можете увидеть весь текст целиком.

1. Выберите пункт меню **File** → **New Scene**.
2. Откройте **Script Editor**.
3. Убедитесь в том, что режим **Echo All Commands** отключен, выбрав пункт **Script** | **Echo All Commands** в случае, если соответствующий флашок установлен.
4. Щелкните в любом месте панели **Command Input Panel** (белой области) в окне **Script Editor**.
5. Введите следующий текст:
`textCurves -t "Hello World"`

6. Нажмите клавиши **Ctrl+Enter**.

На сцене будет создан текстовый объект со словами «Hello World». Обратите внимание на то, что набранный вами текст **переместился** в область отклика в окне **Script Editor**, а под ним появился результат:

```
textCurves -t "Hello World";
// Result: Text_Hello_World_1 makeTextCurves1 //
```

Нажатие клавиш **Ctrl+Enter** или, аналогично, нажатие **Enter** на цифровой клавиатуре приводит к выполнению любого текста, находящегося в данный момент в **Script Editor**. Записывая многострочные команды, нажимайте обычную клавишу **Enter**, и курсор будет **перемещаться** вниз на следующую строку.

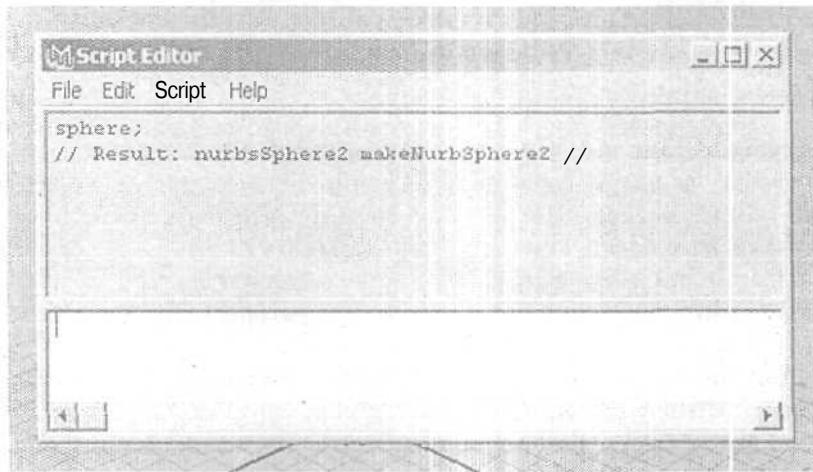


Рис. 3.2. Редактор сценариев

Заметьте, что Maya всегда выводит результат команды в окружении особого символа **(//)**. Когда вы запускаете текст из области **Command Input Panel**, Maya выполняет все операторы, которые там содержатся. Можно выполнить лишь часть операторов, для чего нужно выделить курсором те из них, которые вы намерены запустить, а затем нажать клавиши **Ctrl+Enter** или **Enter** (на цифровой клавиатуре). Этот метод также предотвращает автоматическое перемещение операторов из панели **Command Input Panel** в область **History Panel**.

Теперь хотелось сделать так, чтобы данная команда MEL стала доступной в любое время. Неплохим способом решения этой задачи является создание на панели **Shelf** (Полка) кнопки, щелчок по которой будет запускать команду.

1. Если панель **Shelf** не видна, выберите в главном меню пункт **Display | UI Elements | Shelf** (Отобразить Элементы интерфейса пользователя | Полка).
2. На панели **History Panel** в окне **Script Editor** выделите текст:
`textCurves -t "Hello World";`
Убедитесь в том, что вы выделили точку с запятой (;). Она используется для отделения операторов друг от друга.
3. Левой кнопкой мыши перетащите выделенный текст на панель **Shelf**.
На панели Shelf появится новая кнопка. Есть и другой способ: вы могли выделить текст, а затем выбрать в меню **Script Editor** пункт **File | Save Selected to Shelf...** (Файл Сохранить выделенный текст на полке...).
4. Выберите пункт меню **File | New Scene**.
5. Щелкните по вновь созданной кнопке на панели **Shelf**
Появится текст «Hello World».

Использование кнопок на панели **Shelf** является простым и оперативным средством выполнения команд MEL. Кнопки **Shelf** и команды MEL, которые они содержат, запоминаются при сохранении панели. Кроме того, вы можете отредактировать команды MEL, используемые каждой кнопкой.

Последнее и самое «долгоживущее» средство выполнения серии команд языка MEL - это сохранение их в виде файла. Команды, которые хранятся в файлах, носят название *сценариев* MEL. Сценарии - это обычные текстовые файлы с расширением .mel. Запуская такой сценарий, Maya открывает его и выполняет каждую указанную в нем команду.

Чтобы различать место, где заканчивается одна команда и начинается другая, нужно использовать точку с запятой (;), помещая ее в конце каждой команды. На самом деле, гарантировать, что каждая команда заканчивается точкой с запятой, - значит придерживаться хорошей практики программирования.

1. Выделите следующий текст на панели **History Panel** в окне **Script Editor**.
Снова убедитесь в том, что выделение содержит точку с запятой.
`textCurves -t "Hello World";`
 2. В меню **Script Editor** выберите пункт **File | Save Selected...** (Файл Сохранить выделенный текст...).
 3. Введите `myscript.mel` в качестве имени файла.
 4. Щелкните по кнопке **Save** (Сохранить).
- Команда сохранена в файле `myscript.mel`. Теперь вы можете выполнить файл сценария MEL полностью.

5. Выберите пункт меню **File | New Scene**.
6. На панели **Command Input Panel** в окне **Script Editor** наберите следующий текст:

```
source myscript.mel
```
7. Нажмите клавиши **Ctrl+Enter**.
Сценарий будет выполнен, и на сцене снова появится текст «Hello World». При чтении в память файл сценария загружается в Maya, где и выполняется. Расположение сценария также можно указать, выбрав в меню **Script Editor** пункт **File | Source Script...** (Файл | Текст сценария...).

Теперь вам должно быть ясно, что существует множество разных путей выполнения команд языка **MEL**. Перечислим различные методы и подробно расскажем о том, когда они могут быть или не быть самым подходящим решением.

◆ **Command Line**

Применяется для записи и выполнения простых команд, которые в длину обычно занимают одну строку. Пользуясь точкой с запятой для отделения команд друг от друга, можно записать в **Command Line** больше одной команды, однако на практике такая конструкция становится громоздкой при работе с любыми командами за исключением самых простых.

◆ **Command Shell**

Command Shell (Командная оболочка) - это окно, работающее подобно окну оболочки **Unix**. В нем вы можете выполнять команды языка **MEL**. Те, кто знаком с оболочками **Unix**, при использовании **Command Shell** могут почувствовать себя более комфортно. Для обращения в этому окну выберите в главном меню Maya пункт **Windows | General Editors... | Command Shell...** (Окна Универсальные редакторы... Командная оболочка...).

* **Script Editor**

Script Editor позволяет выполнять запись многострочных сценариев. После выполнения сценарий пополняет перечень на панели **History Panel**. Вы можете скопировать его и вставить обратно на панель **Command Input Panel** для дальнейшего редактирования и усовершенствования. Это основной метод создания простых, но функциональных сценариев. Когда сценарий протестирован и отложен средствами **Script Editor**, вы можете сохранить его в файл сценария для повторного использования в будущем.

◆ Shelf

Закончив написание и тестирование команд, вы можете связать их с элементом управления на панели **Shelf**. Команды, связанные с тем или иным элементом, можно отредактировать при помощи **Shelf Editor** (Редактора полки).

* Файлы сценариев

Запись сценариев в виде файлов – основной способ задания серии **команд**, которые вы хотите сохранить между сеансами работы с Maya. После записи в файл сценария эти команды могут использоваться в различных сценах и проектах.

* Прочие

Есть и другие способы выполнения команд MEL, включая «горячие клавиши», выражения и узлы сценариев (Script Nodes). Речь о них пойдет ниже в этой главе.

Аргументы команд

В предыдущем примере была выполнена команда `sphere`. При этом был построен сферический объект, имевший размер по умолчанию. Размер сферы, которую вы хотите получить, можно задать заранее. Для этого при вызове команды `sphere` нужно всего лишь **добавить описание** радиуса сферы. Чтобы создать сферу радиуса 2, воспользуйтесь командой

```
sphere -radius 2;
```

За командой `sphere` следует аргумент, имеющий значение радиуса: `-radius 2`. Этот аргумент содержит информацию о том, какой параметр требуется задать, а также о том, какое значение должно быть установлено. В этом случае аргумент есть `-radius`, а его значение равно 2. Если вы не укажете аргумент явным образом, как это сделано в случае с радиусом, то параметр будет использовать значение по умолчанию. Если при вызове `sphere` радиус не указан, будет принято значение радиуса по умолчанию, равное 1.

1. Выберите пункт меню **File | New Scene**.
2. В строке **Command Line** введите команду:
`sphere`
3. Нажмите клавишу **Enter**.
Будет создана сфера единичного радиуса.
4. Выполните следующую **команду**, набрав ее в строке **Command Line** и нажав клавишу **Enter**.
`sphere -radius 2`

Будет создана более крупная сфера радиуса 2.

Практически все аргументы команд имеют следующий общий вид:

-флаг значение

Флаг - это конкретный **параметр**, который вы хотите установить. За ним следует значение, которое вы намерены присвоить этому параметру. Можно установить и несколько параметров.

1. В строке **Command Line** выполните команду:

```
sphere -name MySphere -radius 3
```

Будет создана новая сфера радиуса 3 с именем **MySphere**. Если вы используете Maya в интерактивном режиме, как делали это до сих пор, было бы лучше, если бы вам не приходилось каждый раз набирать так много текста. К счастью, Maya предоставляет возможность работать с командными параметрами в двух видах - с использованием кратких и полных **имен**. Краткое имя состоит лишь из нескольких символов, а потому вы сможете набрать его **быстрее**, чем полное.

2. Более лаконичный вариант создания той же сферы.

В строке **Command Line** выполните команду:

```
undo; sphere -r 3 -n MySphere
```

Команда **undo** удаляет ранее построенную сферу. Затем создается новая сфера, для чего используется краткая форма имен параметров. Обратите внимание, что для разделения двух команд применяется точка с запятой.

Хотя краткая форма команд проще при их быстрой записи и **выполнении**, подобные команды не так просто понять с первого взгляда, как команды в их **полной форме**. В частности, при составлении сценариев предпочтительнее пользоваться длинными именами, так как это позволит читателю более четко понимать происходящее. Преимущества, связанные с применением полной формы записи, также касаются и сценариев, которые когда-нибудь будете читать только вы. Глядя на сценарий, составленный какое-то время назад, вы **почувствуете**, что длинные имена параметров проясняют картину, если вы **забыли**, чего именно хотели добиться. Полная форма сразу же позволяет **уяснить**, какой параметр устанавливается, и поэтому ей нужно отдавать предпочтение. Ё сравнивании с краткой формой.

Информация о Команде

Команда `sphere`, как только что упоминалось, принимает много различных аргументов. А какие еще аргументы она может принять? Подобные вопросы задаются очень часто, поэтому в Maya предусмотрена команда `help`. Она выводит краткий перечень аргументов, которые принимает команда, а также конспективное их описание.

1. Откройте редактор **Script Editor**.

Он будет использован потому, что длина текста, генерируемого командой `help`, превышает одну строку; пользуясь **Command Line**, вы увидели бы лишь последнюю строку текста подсказки, выведенную в области **Command Feedback**.

2. Выполните следующую команду:

```
help sphere
```

Помните, что для выполнения текста в окне **Script Editor** можно воспользоваться комбинацией клавиш **Ctrl+Enter** или клавишей **Enter** (на цифровой клавиатуре). Результатом будет отображение текста, выведенного по команде `help`.

```
help sphere
// Result:
Synopsis: sphere [flags] [String...]
Flags:
-e -edit
-q -query
-ax -axis           Length Length Length
-cch -caching       on|off
-ch -constructionHistory on|off
-d -degree          Int
-esw -endSweep      Angle
-hr -heightRatio    Float
-n -name             String
-nds -nodeState     Int
-nsp -spans          Int
-o -object           on|off
```

```

-p -pivot           Length Length Length
-po -polygon        Int
-r -radius          Length
-s -sections        Int
-ssw -startSweep   Angle
-tol -tolerance    Length
-ut -useTolerance  on|off
//
```

Команда `help` дает сравнительно краткий обзор данной команды. Описание синтаксиса содержит ее общую форму, включая различные необязательные флаги и аргументы, которые вы можете задавать. Далее приведены все флаги, в том числе их краткие и полные имена. Вы можете использовать любое из этих имен. Вслед за каждым флагом указан тип значений, которые ему можно присваивать.

Команда `help`, в частности, полезна в тех случаях, когда вы знаете, какую команду хотите использовать, но забыли некоторые из принимаемых ею параметров. Наиболее полным руководством по всем встроенным командам Maya является справочник Mel Command Reference. Он содержит полный список всех команд языка MEL, а также детальное описание их флагов и других параметров. Чтобы обратиться к документации по Maya, выполните следующие действия.

1. В редакторе **Script Editor** выберите пункт **Help | MEL Command Reference...** (Помощь Справочник по командам MEL...).

Из того же меню **Help** вы можете также выбрать пункт **Help on MEL...** (Справка по MEL...) и **Help on Script Editor...** (Справка по Script Editor...).

2. Щелкните по ссылке `s`, расположенной в горизонтальном алфавитном списке.
3. Найдите команду `sphere` и щелкните по ней.

Вы увидите полное руководство по команде `sphere`. Формат и стиль оформления страниц одинаковы для всех команд MEL. В частности, все параметры перечислены в разделе флагов, где приведено полное описание назначения каждого параметра. Также перечислены их значения по умолчанию. Несмотря на то что остальная часть страницы не нуждается в пояснениях, важно отметить раздел **Examples** (Примеры). В нем содержатся небольшие примеры, которые иллюстрируют варианты использования той или иной команды. Это особенно полезно, если вы не уверены в том, что касается применения команды или возможности установки конкретного параметра.

При стандартном отображении справочника MEL Command Reference команды перечислены в алфавитном порядке. Возможно, вы захотите увидеть все команды, связанные с решением конкретной задачи, например команды создания элементов пользовательского интерфейса.

1. В верхней части страницы справа щелкните по кнопке **Sort By: Function** (Сортировать по: Назначение).

На экране появится список всех возможных разделов языка MEL.

2. Щелкните по ссылке **NURBS** в разделе **Modeling** (Моделирование).

Среди множества команд **NURBS**-моделирования есть и команда **sphere**.

Поскольку MEL Command Reference так полезен при написании сценариев MEL, его нужно постоянно держать открытым. Быть может, вам стоит создать для него закладку в своем браузере. Еще один способ найти нужную команду – воспользоваться механизмом поиска.

1. В верхней части страницы щелкните по ссылке **Search** (Поиск).
2. Щелкните по элементу управления **Search In:** (Искать в), затем выберите из выпадающего списка строку **Maya Commands** (Команды Maya).
3. Наберите в строке поиска слово **sphere** и щелкните по кнопке **Search** (Искать). Вы увидите список всех разделов документации, где употребляется слово **sphere**.

Команда запроса

Результатом команды **sphere** был вновь созданный сферический объект. Используя ту же команду **sphere**, вы можете запросить некоторые свойства этой сферы. Команда **sphere** как таковая служит для двух целей – для построения сферического объекта, а также, позднее, для получения его свойств. При таком применении команда **sphere** может работать и с другими сферическими объектами сцены.

1. Выберите пункт меню **File | New Scene**...
2. В строке **Command Line** выполните команду
`sphere -radius 1.5 -name MySphere`
3. В строке **Command Line** выполните команду
`sphere -query -radius MySphere`
4. В строке **Command Feedback** будет выведен следующий результат:
Result: 1.5

Обратите внимание, что имя сферы должно быть задано. Запрашивая свойство сферы, вы должны точно указать, к какому сферическому объекту обращаетесь. По этой причине во время создания сферы ей было дано известное имя.

При первом обращении к команде `sphere` была построена сфера `MySphere`. Это пример вызова команды в *режиме создания*. В нем команда порождает новый объект. При втором обращении к команде `sphere` использовался флаг `-query`. С этим флагом команда `sphere` работает в *режиме запроса* и возвращает значение указанного параметра.

Флаг `-query` поддерживают почти все команды. Чтобы узнать, поддерживает ли данная команда режим запроса, загляните в MEL Command Reference. Тип возвращаемого значения зависит от конкретного запрашиваемого параметра. В предыдущем примере, запросив радиус сферы, вы получили число с плавающей запятой. Выполняя команду в режиме запроса, за один раз вы можете получить значение лишь одного параметра. Для запроса нескольких параметров выполните несколько команд запроса подряд.

Команда правки

Наряду с режимами создания и запроса, команды могут работать в *режиме правки*. В этом случае они способны изменять значения выбранных свойств объектов.

1. В строке Command Line выполните команду

```
sphere -edit -radius 3 MySphere
```

Теперь сфера имеет радиус 3.

Как и флаг запроса, флаг правки `-edit` служит для указания параметра, с которым работает команда. Однако в отличие от флага запроса, позволяющего запрашивать только одно значение в каждый момент времени, флаг правки допускает одновременное изменение нескольких параметров.

2. В строке Command Line выполните команду

```
sphere -edit -radius 4 -degree 1 MySphere
```

Теперь сфера стала больше и имеет ограниченную поверхность.

Изменение порядка сферы превращает ее в линейную, а не гладкую поверхность третьего порядка, какой она являлась ранее. Как и в режиме запроса, имя сферы должно быть указано для того, чтобы сообщить Maya, какую сферу вы собираетесь редактировать.

Режимы команд

Из нескольких предыдущих примеров становится ясно, что команда может, в общей сложности, работать в трех режимах: в режиме *создания*, *запроса* или *правки*. Если вы не укажете явным образом режим запроса или правки, команда будет работать в режиме создания. Если среди аргументов команды *задан* флаг *-query* или *-edit*, то используется соответствующий режим. Приведем некоторые примеры команды *sphere*, выполняя ее в *различных режимах*.

```
sphere -name Sphered; // Режим создания  
sphere -query -radius Sphered; // Режим запроса  
sphere -edit -radius 2 Sphered; // Режим правки
```

Немаловажно отметить тот факт, что смешивать разные режимы в одном вызове команды нельзя. Невозможно, к примеру, создать сферу и в то же время выполнить запрос к ней. Для этого разбейте такой оператор на два вызова команды *sphere*. Первый создаст сферу, а второй выполнит запрос к ней.

Не все команды поддерживают различные режимы. Чтобы найти подробные сведения о том, какие режимы поддерживает каждая команда, обратитесь к документации MEL Command Reference. В ней перечислены все флаги и поддерживаемые режимы команд. Они обозначаются значками С, Q, Е и М, что соответствует созданию, запросу, правке и множественному использованию. Последняя возможность предназначена для флагов, которые могут многократно использоваться в одной и той же команде.

3.2.2. Переменные

Все языки программирования предусматривают механизм хранения *значений*, которыми позднее можно воспользоваться снова. Такой механизм известен как *переменные*. В следующем примере показано, как можно получить результат выполнения команды MEL и поместить *его* в переменную.

1. Откройте редактор Script Editor.
2. Наберите в области ввода **Command Input Panel** следующий текст:

```
$rad = 2;  
sphere -radius $rad;
```
3. Для выполнения команд нажмите клавиши **Ctrl+Enter**.
Вы получите сферу радиуса 2.

Вместо явного задания радиуса сферы в команде `sphere` была использована переменная. Первая строка содержит описание переменной `$rad`, которой присвоено значение 2. Все переменные в языке MEL начинаются со знака доллара (\$). Во время вызова команда `sphere` использует значение, хранящееся в переменной `$rad`, для задания параметра радиуса.

Переменная может иметь любое имя с учетом некоторых незначительных ограничений. Имена переменных не могут содержать пробельных символов (пробелов, табуляции и т. д.) или других специальных знаков. Переменные не могут иметь имен, начинающихся с цифр, хотя цифры могут использоваться в имени после первого символа. Имена переменных *чувствительны к регистру*. Чувствительность имен к регистру предполагает, что использование вами букв верхнего и нижнего регистра имеет существенное значение. Так, в языке MEL имена `$RAD`, `$rad` и `$Rad` соответствуют различным переменным,

Типы переменных

MEL содержит ограниченный набор типов переменных: `int`, `float`, `string`, `vector` и `matrix`. Однако при составлении сценариев `иX` вполне достаточно для нужд большинства пользователей. Если тип переменной указан, а начальное значение `не` задано, переменной присваивается значение по умолчанию.

Целые числа

Для хранения целых чисел служит тип `int`. Переменные этого типа не могут содержать вещественных значений.

```
int $a = -23;  
int $b = 100;  
int $c = 270038;
```

Реальное количество разрядов, выделяемых для хранения значения типа `int`, зависит от платформы. Хотя, в общем случае, для надежности можно полагать, что минимальный размер типа `int` равен 32 битам, а значит, возможные значения лежат в диапазоне от -2 147 483 648 до 2 147 483 647; некоторые платформы выделяют для хранения `int` 64 разряда или более, тем самым обеспечивая более широкий диапазон значений. По умолчанию значение переменной типа `int` равно 0.

Вещественные числа

Для хранения вещественных чисел используется тип `float`. Такие числа могут содержать цифры после десятичной *запятой*.

```
float $a = 10002.34;  
float $b = 1.0;  
float $c = -2.35;
```

Как и в случае с `int`, количество разрядов, выделяемых для хранения `float`, напрямую зависит от платформы. Этот тип эквивалентен типу данных `double` в языке С. На большинстве платформ это означает, что переменные вещественного типа представлены 64 битами. Поэтому они могут хранить очень широкий диапазон значений³: от 2,2250738585072014e-308 до 1,7976931348623158e+308. Точность значений составляет 15 цифр. Значение переменной типа `float` по умолчанию равно 0.0.

Строки

Для хранения последовательности текстовых символов служат переменные типа `string`. Инициализировать строковую переменную некоторым текстовым значением можно следующим образом:

```
string $txt = "Hello there";
```

Если строка не получает значения явным образом, то используется значение по умолчанию, равное пустой строке. Это строка, которая не содержит текста.

```
string $txt; // Пустая строка
```

Присвоить переменной значение пустой строки вы можете непосредственно:

```
$txt = "";
```

Распространенной операцией над строковыми переменными является их склеивание (объединение). Для этого предназначен знак «плюс» (+).

```
string $name = "David";
string $txt = "My name is " + $name;
print $txt; // Результат = My name is David
```

Для подсчета количества символов в строке воспользуйтесь командой `size`.

```
string $welcome = "Hi";
int $numChars = size( $welcome );
print $numChars; // Результат: 2
```

Важно понять то, что MEL предварительно обрабатывает каждую строку и выполняет подстановку специальных символов там, где встречается обратная косая черта (\). Специальным символом называется символ, который часто невозможно ввести вручную, работая в текстовом редакторе, но который присутствует в таблице *ASCII*. Такие символы нередко называют управляемыми, или ESC-символами. Чтобы записать последовательность, содержащую символ новой строки, просто используйте пару символов \n. Они будут заменены

³ По абсолютному значению. - Примеч. перев.

ASCII - символом, эквивалентным переводу каретки. В следующем примере часть текста выводится на одной строке, а остальной текст - на другой.

```
print "Hi there\nBob";
```

Результат имеет вид:

```
Hi there
```

```
Bob
```

К числу других специальных символов относятся

\t табуляция

\r возврат каретки

\\" обратная косая черта

Обратите внимание: чтобы включить в строку символ обратной косой черты (\\"), вы должны записать его как \\\. Если этого не сделать, то MEL, встретив обратную косую черту, попытается выполнить подстановку **спецсимвола**. Это может вызвать особые трудности при записи полного пути к **файлу** в среде Windows, поскольку символы \ используются в ней для разделения каталогов. Стока

```
"c:\maya\temp"
```

после обработки препроцессором **MEL** примет такой окончательный вид:

```
"c:maya      temp"
```

Эта символьная строка содержит две ESC-последовательности, первая из которых – \\m, а вторая – \\t. Пара символов \\m не является корректной ESC-последовательностью и поэтому удаляется. Последовательность \\t заменяется символом табуляции. Для обеспечения сохранности **каждой** обратной косой черты нужно использовать запись \\\. Итак, исходная строка должна быть записана следующим образом:

```
"c:\\maya\\temp"
```

Окончательная строка после автоматических подстановок в **MEL** станет такой, как было задумано изначально:

```
"c:\maya\temp"
```

Размер текста, который можно сохранить в строке, ограничен только доступным объемом памяти, однако абсолютный верхний предел определяется максимальным значением типа **int**.

Векторы

Векторная переменная позволяет удерживать в памяти три значения с плавающей запятой. Обычно векторы служат для хранения точек и направлений в пространстве.

1. В окне **Script Editor** введите следующие команды:

```
sphere;  
vector $p = << 10.0, 5.0, 2.5 >>;  
move -absolute ($p.x) ($p.y) ($p.z);
```

2. Нажмите клавиши **Ctrl+Enter** для их выполнения.

Будет построена сфера, которая затем займет указанное положение ($x = 10$, $y = 5$ и $z = 2,5$).

Для доступа к отдельным компонентам вектора x , y , z , соответственно, используются конструкции $$p.x$, $$p.y$ и $$p.z$. Важно отметить, что при обращении к компонентам вектора их следует заключать в круглые скобки. Следующий пример иллюстрирует проблему отсутствия скобок.

```
vector $p = << 10.0, 5.0, 2.5 >>;  
print $p.x; // Ошибка  
print ($p.x); // Результат: 10.0
```

Другой характерной особенностью **MEL** является запрет на непосредственное присваивание числового значения компоненту вектора. Следующий код, например, приводит к ошибке:

```
$p.x = 3.0; // Ошибка  
($p.x) = 3.0; // Ошибка
```

Присвоить числовое значение компоненту вектора напрямую, по сути дела, нельзя. Такая операция должна выполняться косвенно, путем создания еще одного вектора и присваивания его исходному. Цель следующего примера- присвоить компоненту $$p.x$ значение 3,0.

```
vector $p = << 10.0, 5.0, 2.5 >>;  
$p = << 3.0, $p.y, $p.z >>; // Присвоить 3.0 компоненту x  
print $p // Результат = 3 5 2.5
```

Если вектор не получит значение явным образом, то будет использовано значение по умолчанию, равное $<<0.0, 0.0, 0.0>>$.

```
vector $p; // Автоматическая инициализация по умолчанию: «0.0, 0.0, 0.0»
```

Массивы

Иногда возникает необходимость хранения целого ряда переменных. В языке MEL вы можете описать **массив**, который образован несколькими переменными. Количество элементов массива часто неизвестно заранее, поэтому **MEL** допускает автоматическое увеличение числа элементов, если это необходимо. Далее показано несколько примеров описания массивов.

```
int $values[] = { 2, 5, 7, 1 };
string $names[3] = { "Bill", "Bob", "Jeff" };
vector $positions[2] = { <<0.5, 0.1, 1.0>>, <<2.3, 4.0, 1.2>> };
float $scores[]; // Пустой массив
```

При описании массива должны использоваться **квадратные скобки** ([]). Чтобы обратиться к отдельному **элементу** массива, в скобках нужно указать его индекс.

```
int $values[4] = { 2, 5, 7, 1 };
print ( $values[0] ); // Результат: 2
print ( $values[3] ); // Результат: 1
```

Индексы следуют от 0, поэтому индекс первого элемента массива равен 0, второго – 1 и т. д. Совокупный диапазон индексов **массива** – это ряд чисел от 0 до (**количество элементов** - 1). Чтобы найти общее число элементов массива, воспользуйтесь командой **size**.

```
int $values[3] = { 4, 8, 9 };
int $numElements = size( $values );
print $numElements; // Результат: 3
```

Типичной операцией является присоединение элемента к существующему массиву. Для этого используйте команду **size** следующим образом:

```
int $values[2] = { 5, 1 };
$values[ size($values) ] = 3; // Присоединить 3 к массиву
print $values; // Результат: 5 1 3
```

Кроме того, элемент можно добавить, непосредственно обратившись к нему.

```
int $values[]; // Пустой массив
$values[0] = 2; // Первому элементу присвоено значение 2
$values[2] = 7; // Третьему элементу присвоено значение 7
print $values; // Результат: 2 0 7
```

Изначально массив был пустым, т. е. не содержал элементов. Путем непосредственного присваивания значения первому **элементу** (с индексом 0) размер массива был увеличен с целью включения в него нового элемента. Тогда в **мас-**

сив входил всего один элемент. В следующей строке устанавливается значение третьего элемента (с индексом 2), поэтому размер массива увеличивается до трех. Так как значение третьего элемента было указано, он стал равен 7. Заметьте, что вы не задали значение второго элемента (с индексом 1). Увеличивая размер массива, Maya присвоила значение по умолчанию всем элементам, которые не прошли явной инициализации. Коль скоро это массив целых чисел, то второй элемент автоматически получает значение по умолчанию, принятое для целых, т. е. 0.

По мере добавления элементов массив занимает больше пространства в памяти. При работе с крупными массивами, содержащими множество элементов, это может привести к выделению немалого ее объема. Если элементы массива уже не нужны, то лучше удалить их, чтобы освободить память. Команда `clear` удаляет все элементы массива и освобождает память, которую он использовал.

```
int $values[4] = { 6, 9, 2, 1 };
print $values; // Результат: 6 9 2 1
clear( $values );
print $values; // Результат:
```

К сожалению, удалять элементы из массива напрямую нельзя. Вместо этого вы должны создать новый массив, содержащий элементы, которые вы хотите оставить, а затем скопировать их обратно в исходный массив. В следующем фрагменте кода этот метод продемонстрирован на примере удаления из массива третьего элемента (с индексом 2).

```
int $values[] = { 1, 2, 3, 4 };

// Удалить третий элемент
int $temp[];
int $i;
for( $i=0; $i < size($values); $i++ )
{
    if( $i == 2 ) // Пропустить третий элемент (с индексом 2)
        continue;
    $temp[size($temp)] = $values[$i];
}
$values = $temp;

print $values;
// Результат: 1 2 4
```

Заметьте, что массивы могут быть только одномерными. Для двухмерных массивов фиксированного размера используется тип `matrix`, описанный ниже. При работе с динамическими многомерными массивами вам придется упаковать их в одномерный. Пусть в следующем примере нужно создать двухмерный массив, т. е. такой `массив`, что вы сможете обращаться к его `элементам` по индексам строк и столбцов. Далее будет создан одномерный `массив`, который `аккумулирует` двухмерные данные. Он имеет две строки и четыре столбца. В одномерном массиве первая строка 1, 2, 3, 4 предшествует второй строке 5, 6, 7, 8.

```
int $pixs[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

Чтобы определить, чему равен единственный индекс такого массива, который соответствует данной строке и данному столбцу, используется следующая формула:

индекс = строка x количество_столбцов + столбец.

Помните, что все индексы начинаются с 0. Поэтому для доступа к элементу в строке 1 (второй строке) и столбце 3 (четвертым `столбце`) нужно выполнить следующие действия:

```
int $nCols = 4;
int $index = 1 * $nCols + 3;
print $pixs[$index];
// Результат: 8
```

Эту формулу для доступа к элементам можно обобщить в процедуру, которая сможет обращаться к любой строке и любому столбцу `массива`. Процедуры будут описаны позднее.

Матрицы

Матрицы очень похожи на массивы, за исключением того что они имеют два измерения. В то время как массивы можно считать `одной` строкой элементов, матрицы содержит как строки, так и столбцы. Однако в отличие от массивов, они не могут изменять свой размер после описания. По сути дела, матрицы не могут расширяться за счет `дополнительных` элементов. К тому же они могут содержать лишь значения типа `float`.

В описании матрицы указывают количество строк и столбцов, которые она будет содержать.

```
matrix $m[2][1]; // Установить размеры матрицы: 2 строки, 1 столбец
```

Если размеры матрицы по каждому изменению не указаны явно, возникает ошибка.

```
matrix $m[2][]; // ERROR: size not specified (ошибка: размер не указан)
matrix $m[][]; // ERROR: size not specified
```

Матрицы инициализируются с помощью конструкций, синтаксически схожих с аналогичными конструкциями для векторов. Элементы матрицы указываются построчно. В следующем примере первая строка матрицы содержит значения 3, 4, 6, 7, вторая - значения 3, 9, 0, 1.

```
matrix $m[2][4] = << 3,4,6,7; 3,9,0,1 >>;
```

Обращение к элементам матрицы производится почти так же, как к элементам массивов, за исключением необходимости указания индекса столбца.

```
matrix $m[2][4] = << 3,4,6,7; 3,9,0,1 >>;
print( $m[0][0] ); // Результат: 3
print( $m[1][3] ); // Результат: 1
```

Так как матрица имеет фиксированное количество строк и столбцов, то при попытке обращения к элементу за пределами диапазона возможных индексов отображается ошибка.

```
matrix $m[2][4] = << 3,4,6,7; 3,9,0,1 >>;
print C $m[2][0];
// Error: line 2: Index value of 2 exceeds dimension limit (2) of $m. //
// Ошибка: строка 2: Значение индекса 2 превышает размер (2) $m по
данному измерению. //
```

Если при описании матрица не получает значений явным образом, каждый ее элемент автоматически инициализируется значением 0.0.

```
matrix $m[2][1]; // Автоматическая инициализация: «0.0; 0.0»
```

Автоматическое определение типов переменных

В отличие от более строгих языков программирования, MEL не требует указания типа переменной при описании. Он определяет его по значению, которое ей присваивается.

```
$rad = 2.1;
```

Так как значение 2.1 может быть представлено только как число с плавающей запятой, переменной \$rad будет назначен тип **float**. Отсюда понятно, насколько важно знать принципы, по которым Maya различает переменные разных типов. Возьмем следующий пример:

```
$a = 2;
$b = 2.0;
```

Даже притом что оба эти значения могут храниться как целые числа, равные 2, переменная \$b получит тип **float**, поскольку второе значение имеет

вид 2.0. Такое автоматическое распознавание может привести к некоторым малозаметным ошибкам в ваших сценариях. Если вы работаете с прямоугольным объектом, а именно его точкой (3.43, 0, 0), то выполнение следующих операторов приводит к неожиданному результату.

```
$i = 5;  
$i = `getAttr box.translateX`;  
print $1;  
//Результат: 3
```

Несмотря на то что полученный результат равен 3, правильное значение координаты объекта по оси x составляет 3.43. Есть две причины, по которым результат стал именно таким. После создания переменной \$i ей было присвоено значение 5. Поскольку вы не указали явным образом тип этой переменной, Maya предположила, что это – целое число, коль скоро значение 5, очевидно, целое. В следующем операторе вы запрашиваете значение координаты объекта по оси x. Maya возвращает корректное значение 3.43, однако оно сохраняется в целочисленной переменной \$i. Учитывая, что переменная \$i является целой, в ней нельзя сохранить все десятичные цифры, а только целую часть. Поэтому, в конечном итоге. \$i принимает значение 3.

На самом деле, переменная \$i должна была получить тип float, с тем чтобы она могла хранить все десятичные цифры координаты по оси x. Общее правило гласит: тип переменной лучше всего явно указывать при ее описании. Это служит гарантией того, что вы не зависите от Maya, которая автоматически определяет тип переменной. Предыдущие операторы теперь следует читать так:

```
float $i = 5;  
$i = `getAttr box.translateX`;  
print $i;  
//Результат:3.43
```

Так как это может привести к некоторому замешательству и трудно находимым ошибкам, тип переменной лучше всего явно указывать при ее описании. Следующие примеры иллюстрируют сказанное.

```
float $a = 34; // присвоить значение 34.0  
int $b = 2.3; // присвоить значение 2
```

Коль скоро переменная типа float может содержать любое вещественное число, включая его дробную часть, она без проблем принимает значение 34. С другой стороны, переменная типа int не может принять значение с десятичными цифрами.

рами после запятой. Тип `int` может хранить лишь целую часть числа, поэтому `.3` отбрасывается, остается лишь значение `2`.

Как только тип `переменной` – явно или `неявно` – установлен, его уже нельзя изменить. В следующем примере переменная `$a` неявно приобретает тип `float` в результате присваивания ей значения `2.3`. С этого момента в `$a` можно хранить только данные этого типа. Если вы присвоите ей значение, тип которого отличен от `float`, Maya попытается привести его к вещественному типу. Присваивание переменной `$a` значения типа `string` сбрасывает переменную в `0`, так как осмысленное преобразование `string` в значение `float` невозможно. К счастью, Maya предупреждает о подобных преобразованиях.

```
$a = 2.3;      // $a неявно принимает тип float
$a = "notes"; // попытка превратить $a в строку
// Warning: $a = "notes";
//
// Warning; Line 3.13 : Converting string "notes" to a float value of
0.//
// Предупреждение: Стока 3.13 : Преобразование строки "notes" в значение
О типа float.//
```

Важно понимать, какие возможные значения могут принять переменные каждого типа. Общее правило гласит: для хранения всех чисел должны применяться значения вещественного типа, поскольку это именно тот `формат`, который обычно использует Maya. Для подсчета количества или организации циклов по спискам можно воспользоваться типом `int`.

Автоматическое преобразование типов

Другой, еще более таинственной причиной проблем в сценариях является автоматическое преобразование типов. Трудно их бывает отследить потому, что они требуют досконального `понимания` механизмов приведения типа `данных` одной переменной к типу данных другой переменной и обратно. Подобное понимание опирается на знание того, как каждый тип данных хранит соответствующую информацию. Возьмем, к примеру, следующие операторы:

```
int $i = 7.534;
print $i;
// Результат: 7
```

Как упоминалось ранее, тип данных `int` может хранить только целочисленные значения. Если вы попытаетесь присвоить такой переменной значение `7.534`, дробная часть будет попросту отброшена, а целая сохранится. В результате пере-

менная получит значение 7. К сожалению, Maya не предупреждает о таких потенциально опасных преобразованиях.

Операторы, которые кажутся простыми, могут работать неправильно лишь из-за того, что различные типы данных функционируют по-разному. В результате выполнения этих операторов будет выведен 0.

```
float $i = 1 / 3;  
print $i;  
// Результат: 0
```

Разве значение не должно составлять 0.333...? Причина, по которой этот код оказался некорректным, состояла в том, что начальное вычисление проводилось по правилам арифметики целых чисел. Maya предположила, что 1 и 3 в первом операторе были целыми. Поэтому для вычисления частного применялась целочисленная арифметика: обычно $1/3$ - это 0.333..., но при использовании целых чисел целая часть результата равна 0, и именно это значение сохраняется в памяти. Код работал бы правильно, если бы операторы были записаны так:

```
float $i = 1 / 3.0;  
print $i;  
// Результат: 0.33333
```

Почему же теперь вычисления приводят к нужному результату? Maya просматривает операнды деления следующим образом. Значение 1 интерпретируется как int. Значение 3.0 интерпретируется как float. На уровне программной реализации Maya не выполняет операций над смешанными типами данных; вместо этого, она приводит все операнды к одному типу, а затем производит вычисления. Так как вещественный тип обладает большей точностью по сравнению с целочисленным, Maya преобразует целое число к вещественному, после чего выполняет операцию над значениями с плавающей запятой, поэтому результат оказывается более точным.

Читателю, наверное, ясно, что проблема автоматического преобразования типов в целом весьма сложна и запутанна. Ради создания качественных программ не нужно овладевать всеми ее хитросплетениями. Однако она может иметь важное значение при отладке сценариев, которые ведут себя весьма неожиданным образом, невзирая на то, что вы их полностью проверили и они не содержат ни синтаксических, ни логических ошибок. Есть вероятность того, что вина ляжет именно на автоматическое преобразование типов. Если вы подозреваете, что это так, выведите на экран значения, полученные НЕ каждом шаге вычислений, с целью проверки результатов. Если они окажутся неверными, разбейте вычислительный процесс на более мелкие части и проверьте результаты каждой

из них. В конце концов, вы обнаружите те операторы, что послужили причиной ваших проблем. Общее правило таково: выполняйте все математические расчеты с использованием типа float, если необходимость работы с другими типами не продиктована очевидными соображениями.

В табл. 3.1 показано, как Maya выполняет автоматическое преобразование разных типов. Эта таблица должна послужить вам общим руководством по интерпретации и хранению значений различных типов.

ТАБЛИЦА 3.1. СОГЛАШЕНИЯ ОБ АВТОМАТИЧЕСКОМ ПРЕОБРАЗОВАНИИ ТИПОВ			
Начальный тип	Конечный тип	Пример	Характер преобразования
int	float	float \$i = 3; // Результат: \$i = 3.0	точное
int	vector	vector \$v = 3; // Результат: \$v = <>3.0,3.0,3.0>	точное
float	int	int \$i = 3.45; // Результат: \$i = 3	только целая часть
float	vector	vector \$v = 3.45; // Результат: \$v = <>3.45,3.45,3.45>	точное
vector	int	int \$i = <<1,2,3>> // Результат: \$i = 3	только целая часть длины
vector	float	float \$i = <>1,2,3>> // Результат: \$i = 3.741657	точное

3.2.3. Комментарии

Вы можете пояснить свои сценарии, делая комментарии к ним. Иногда, просто взглянув на сценарий, невозможно понять, как он работает или почему он был спроектирован именно так, а не иначе. Лучший способ ответить на эти вопросы - добавить комментарии. Они позволяют другим людям лучше понимать ваши намерения. Кроме того, комментарии могут послужить хорошей справкой и для самого разработчика, поскольку зачастую нетрудно забыть, почему данный фрагмент кода был реализован именно таким образом.

Язык MEL поддерживает два типа комментариев. Первый - односторочный комментарий. Он начинается с двух левых косых (//). Комментарием считается весь текст до конца строки, следующий за этими символами.

```
int $avg; // Среднее число частиц в составе сцены
```

Второй тип комментария - многострочный. Чтобы записать комментарий, который занимает более одной строки, просто заключите его в специальные скобки (/*, */).

Л

Этот сценарий - часть расширенного пакета визуализации.

Copyright (c) 2002 Smarty Pants Inc.

*/

Многострочные комментарии нельзя вкладывать друг в друга. Далее приведен пример, где один комментарий содержится внутри другого. Такой код приведет к синтаксической ошибке.

Л

```
int $avg;
```

Л

Для расчета среднего значения используется сумма

по всем сферам в составе сцены.

*/

```
$avg = average( $input );
```

*/

Комментарии могут быть очень удобны для «включения» и «выключения» различных фрагментов сценария. Если, к примеру, вы разрабатываете сценарий для выполнения конкретной задачи, а затем хотите внести в него изменения и усовершенствования, то исходные разделы кода **обычно** копируют и помечают как комментарий. Так позднее вы сможете восстановить начальный вариант кода, если это понадобится.

Основная цель комментария - дать читателю ответ на все вопросы «как» и «почему» относительно того, что вы делаете. Подробный ответ на вопрос «что» не столь важен, поскольку это становится очевидным при просмотре кода. В следующем примере комментарий является излишним, так как понять, что происходит, вы сможете, просто взглянув на оператор.

```
$total = $total + 1; // Прибавить к значению единицу
```

Лучше было бы объяснить, почему это было сделано.

```
$total = $total + 1; // Увеличить на 1 с учетом текущего года
```

3.2.4. Операторы

Теперь, когда вы познакомились с различными типами данных, поддерживаемыми языком MEL, обратимся к операциям, которые вы можете выполнять с этими типами данных.

Оператор присваивания

Самый распространенный оператор – это оператор присваивания (`=`). Он используется для того, чтобы придать переменной некоторое значение.

```
int $score;  
$score = 253;
```

Арифметические операторы

Арифметические операторы выполняют все основные математические операции над числовыми значениями, включая сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`) и др.

Сложение и вычитание

Сложение и вычитание выполняются так, как и должны выполняться в отношении числовых типов.

```
int $a = 3;  
int $b = 5;  
print ($a + $b); // Результат: 8
```

```
float $a = 2.3;  
print ($a - $a); // Результат: 1.7
```

```
vector $va = <<1, 2, 3>>;  
print ($va + <<0.5, 0.5, 0.5>>); // Результат: 1.5 2.5 3.5
```

```
matrix $m[2][1] = << 3; 5 >>;  
matrix $n[2][1] = << 2; 1 >>;  
print ($m + $n); // Результат: << 5; 6 >>
```

Как упоминалось ранее, оператор сложения (`+`) имеет особое значение применительно к переменным `string`. Результат сложения двух строк – это строка, являющаяся комбинацией обеих строк; вторая строка присоединяется к первой.

```
string $a = "Hi";
string $b = "there";
print ($a + $b); // Результат: Hi there
```

Сложение – единственный арифметический оператор, работающий со строками. К примеру, вам не удастся воспользоваться оператором вычитания для удаления одной строки из другой.

Умножение

Умножение работает со всеми числовыми типами **данных**.

```
float $v = 8.2;
print ($v * 2); // Результат: 16.4
```

Результатом умножения двух матриц является одна матрица, удовлетворяющая обычным правилам умножения матриц.

Использование оператора умножения (*) в **отношении** двух векторов позволяет получить **скалярное произведение** этих векторов. Оно эквивалентно перемножению всех компонентов векторов с последующим сложением результатов.

$$\text{склярное произведение}(a, b) = a.x \times b.x + a.y \times b.y + a.z \times b.z$$

Например:

```
vector $a = << 1, 2, 3 >>;
vector $b = << 6, 2, 1 >>;
print ($a * $b); // Результат: 13
```

Умножение матрицы на вектор с целью преобразования, к сожалению, запрещено. Эта операция не поддерживается языком MEL. Взамен вы должны явно описать ее как серию операций умножения и сложения.

Деление и взятие остатка

Деление работает со всеми числовыми типами **данных**. Результатом деления является точно такое значение, какое и следовало ожидать, за исключением деления целых чисел. Целые числа не содержат дробной части, поэтому никакую операцию **деления**, которая приводит к остатку, нельзя представить одним целым числом. Результат деления двух целых чисел есть целая часть частного.

```
int $a = 12;
print ($a / 3); // Результат: 4
```

```
int $a = 12;
print ($a / 5); // Результат: 2 (точное значение 2.4)
```

В то время как оператор деления дает целую часть частного, дробную его часть можно получить при помощи оператора взятия **остатка**, или деления по модулю (%).

```
int $a = 13;
print ($a % 5); // Результат: 3
```

Операторы деления и взятия остатка можно использовать в отношении переменных типа **matrix**. Единственное условие их применения состоит в том, что правая часть оператора должна представлять собой скалярное значение **int** или **float**.

```
matrix $m[2][3] = << 3, 6, 8; 4, 8, 2 >>;
print ($m / 2); // Результат: << 1.5, 3, 4; 2, 4, 1 >>
```

Векторное произведение

Оператор векторного произведения (^) служит для расчета *векторного произведения* двух векторов. Векторное произведение двух векторов есть вектор, расположенный перпендикулярно (по нормали) к обоим исходным **векторам**⁴. Уравнение векторного произведения имеет вид:

$$\begin{aligned} \text{векторное произведение}(a, b) = & (a.y \times b.z - a.z \times b.y, \\ & a.z \times b.x - a.x \times b.z, \\ & a.x \times b.y - a.y \times b.x). \end{aligned}$$

В некоторых языках оператор ^ служит для возведения числа в степень. В MEL для этого применяется команда **pow**.

Логические значения

Прежде чем перейти к изучению операторов логики и отношений, важно понять, что же такое *логическое значение*. Логические (булевы) переменные могут пребывать в одном из двух состояний: **true** (истина) или **false** (ложь). Поэтому они применяются для составления сравнений. Если нечто, как удалось определить, равно **true**, вы можете выполнить то или иное действие. Аналогично, вы можете выполнить действие, если нечто будет равно **false**.

⁴ Более точно, векторное произведение двух векторов должно не только удовлетворять условию ортогональности обоим исходным векторам, но и образовывать с ними правую тройку, если исходные векторы не лежат на параллельных прямых. Напомним, что тройка векторов называется правой, если направление кратчайшего поворота от первого вектора ко второму видно из конца третьего вектора против часовой стрелки. - Примеч. перев.

В MEL логические значения, на самом деле, хранятся как значения типа int. Целочисленные значения констант true и false соответственно равны 1 и 0. Это демонстрируют следующие операторы:

```
$a = true;
$b = false;
print ($a + " , " + $b);
```

Результат: 1, 0

Вместо true могут использоваться дополнительные ключевые слова yes и on. Подобным образом, вместо false могут использоваться ключевые слова no и off. Полный список логических значений приведен в табл. 3.2.

ТАБЛИЦА 3.2. КЛЮЧЕВЫЕ СЛОВА ДЛЯ ОБОЗНАЧЕНИЯ ЛОГИЧЕСКИХ ЗНАЧЕНИЙ

Логическое значение	Целочисленное значение
true, yes, on	1
false, no, off	0

Так как константа true имеет целочисленное значение 1, то любое целое, не равное 0, трактуется как истина в логическом смысле. Это демонстрирует следующий пример,

```
int $a = 5;
if( $a )
    print "a is true" // "а есть истина"
```

Результат:

```
a is true
```

Коль скоро любое целое, отличное от 0, считается истинным, то важно проявлять осторожность при выполнении сравнений с константой true, имеющей значение 1.

```
int $a = 3;
```

```
if( $a )
    print "a is true"
```

// Не выполняется, так как сравнение принимает вид 3 == 1, что ложно

```
if( $a == true )
    print "a is true"
```

Операторы отношений

Очень распространенной задачей программирования является сравнение двух переменных. К примеру, вы хотите выполнить некое действие, если расстояние до объекта превышает известное предельное значение. В виде псевдокода это можно описать так:

```
если расстояние больше чем предел то
    выполнить действие
конец
```

На языке MEL это можно записать следующим образом:

```
if( $distance > $limit )
    action();
```

Если оператор в составе `if(...)` является истинным, то оператор, записанный ниже, будет выполнен. Если же оператор в `if(...)` окажется ложным, то следующий оператор будет пропущен. Таким образом, `if` позволяет вам выполнить одно действие, если оператор сравнения возвращает `true`, и другое действие, если оператор сравнения возвращает `false`. Это продемонстрировано на примере:

```
int $a = 10;
if( $a == 10 )
    print "a is 10"; // "а равно 10"
else
    print "a is not 10"; // "а не равно 10"
```

Оператор, следующий за предложением `else`, выполняется в том случае, когда сравнение оказывается ложным. Если действие включает в себя более одного оператора, его следует заключать в скобки `({}, {})`. Таким образом, операторы объединяются в один логический блок. Следующий код иллюстрирует сказанное.

```
int $a = 10;
if( $a == 10 )
{
    print "a is 10";
    int $b = $a * 2;
}
else
    print "a is not 10";
```

Операторы if также могут вкладываться друг в друга. Написание вложенного оператора if означает, что в исходный if включается еще один оператор такого же типа.

```
int $a = 10;
if( $a < 11 )
{
    if( $a > 5 )
        print "a is between 6 and 10"; // "a лежит в интервале от 6 до 10"
    else
        print "a is less than 6"; // "a меньше 6"
}
else
    print "a is greater than 10"; // "a больше 10"
```

Количество операторов if, которые вы можете вкладывать друг в друга, не ограничено. Существует также оператор else if, однако его применение может привести к написанию менее понятного кода, так что его следует избегать, отдавая предпочтение созданию вложенных операторов if с ограничивающими скобками.

Как всегда, результат оператора отношения является значением логического типа. Поэтому результатом любого сравнения будет значение true или false. Полный перечень операторов отношений приведен в табл. 3.3.

ТАБЛИЦА 3.3. ДОПУСТИМЫЕ ОПЕРАТОРЫ СРАВНЕНИЯ

Отношение	Символ	Пример
меньше	<	a < b
меньше или равно	<=	a <= b
больше	>	a > b
больше или равно	>=	a >= b
равно	==	a == b
не равно	!=	a != b

Далее приведены некоторые примеры их использования.

```
$homeTeam = 234;
$visitingTeam = 123;
```

```
if( $homeTeam > $visitingTeam )
    print "Home team won!";           // "Победили хозяева!"
if( $homeTeam < $visitingTeam )
    print "Visiting team won!";      // "Победили гости!"
if( $homeTeam == $visitingTeam )
    print "Tie";                   // "Ничья"
```

Обратите внимание на то, что символ отношения сравнения, используемый для проверки равенства двух переменных, представляет собой пару знаков равенства (==), а не один такой знак (=). Вспомните, что единичный знак равенства применяется в качестве оператора присваивания. При выполнении следующих операторов переменная \$a получила бы значение 2, а не участвовала бы в сравнении с данной константой.

```
$a = 5;
if( $a = 2)
    print "Second"; // "Второй"
```

Для предотвращения этого код нужно изменить.

```
$a = 5;
if( $a == 2)
    print "Second";
```

Такая оплошность *очень* распространена, но, несмотря на это, Maya не предупреждает о совершении подобной ошибки. Лучший способ избегать этого – помешать значение литерала слева от оператора сравнения.

```
if( 23 == $score )
```

Если и в этом случае вы будете невнимательны и выполните присваивание, MEL укажет вам на ошибку как на попытку присвоить значение элементу, не являющемуся переменной.

```
if( 23 = $score )
```

```
***  
// Error: Line 2.2: Syntax error //
// Ошибка: Стока 2.2: Синтаксическая ошибка //
```

Логические операторы

Иногда возникает желание сравнить более двух переменных, после чего выполнить то или иное действие. Например:

```
float $result = 2.9;
float $worldRecord = 2.87;
float $firstPlace = 2.9;
float $personalBest = 2.8;

if( $result == $firstPlace && $result > $worldRecord )
    print "Winner and beat world record";
    // "Стал победителем и побил мировой рекорд"

if( $result < $firstPlace && $result > $personalBest )
    print "Lost race but beat personal best";
    // "Проиграл гонку, но установил новый личный рекорд"
```

Пользуясь логическими операторами, вы можете комбинировать несколько операций сравнения и выполнять определенные действия. Полный список логических операторов представлен в табл. 3.4.

ТАБЛИЦА 3.4. ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Логический оператор	Символ	Пример
или		-1 0
и	&&	a && b
не	!	!a

При использовании оператора «или» (`||`) результат является истинным, если `a` или `b` имеет истинное значение; в противном случае результат является ложным. Это продемонстрировано на примере:

```
int $a = 10;
int $b = 12;
if($a>3 || $b>5)//true или true = true
    print "yes"; // Будет выведено на экран

if($a==5 || $b==12)//false или true = true
```

```
print "yes"; // Будет выведено на экран  
  
if( $a == 3 || $b == 2 ) // false или false = false  
    print "yes"; // Не будет выведено на экран
```

При использовании оператора «и» (`&&`) результат является истинным лишь тогда, когда *a* и *b* имеют истинное значение; в противном случае результат является ложным.

```
int $a = 10;  
int $b = 12;  
if( $a == 10 && $b == 12 ) // true и true = true  
    print "yes"; // Будет выведено на экран
```

```
if( $a == 10 && $b == 2 ) // true и false = false  
    print "yes"; // Не будет выведено на экран
```

```
if( $a == 3 && $b == 2 ) // false и false = false  
    print "yes"; // Не будет выведено на экран
```

Оператор «не» (`!`) инвертирует результат сравнения. Если результат был истинным, он становится ложным, и наоборот. Проанализируйте следующий код:

```
int $a = 10;  
int $b = 12;  
  
// Сравнение дает истинный результат, однако "не" (!)  
// "переворачивает" значение, и оно становится ложным  
if( !($a == 10 && $b == 12) )  
    print "no"; // Не будет выведено никогда  
  
// Сравнение дает ложный результат, однако "не" (!)  
// "переворачивает" значение, и оно становится истинным  
if( !( $a > 10 ) )  
    print "a is not greater than 10"; // Будет выведено на экран  
    // "а не больше 10"
```

Два следующих оператора эквивалентны.

```
int $a = 0;
if( $a == 0 )
    print "a is false"; // "аложно"
```

// Оператор "не" 0) "переворачивает" логическое значение \$a

// Результатом инверсии 0 становится 1, т. е. истина

```
if( !$a )
    print "a is false";
```

Приоритет операторов

Понимать степень приоритета конкретного оператора очень важно. Незнание порядка, в котором происходит вычисление значения выражения, может приводить к непонятным ошибкам. Код кажется правильным и работает просто замечательно, однако конечный результат оказывается неверным. Часто причина заключается в неверном истолковании порядка, в котором производится вычисление операторов. Рассмотрим следующий пример кода.

```
int $a = 10 + 2 * 5;
```

Из математики вы знаете, что операция умножения должна выполняться раньше операции сложения, поэтому результат равен 20. Если бы умножение не имело приоритета относительно сложения, то результат составил бы 60. Приоритет оператора определяет то, выполняется ли он до или после другого оператора, т. е. от приоритета зависит порядок вычисления выражения.

Хотя большинству пользователей хорошо известно старшинство арифметических операторов, это не относится к другим конструкциям языка MEL. Какой приоритет имеют следующие операторы, иначе говоря, каков порядок их вычисления?

```
$height > 2 || $width == 1
```

Не зная приоритетов, вы, скорее всего, стали бы вычислять эти операторы слева направо в порядке их появления. Пользуясь скобками для группировки, запишем в итоге:

```
((($height > 2) || $width) == 1)
```

Это может не привести к желаемому результату. В действительности, оператор `>` имеет более высокий приоритет, поэтому он вычисляется первым. Следом идет оператор равенства `==`, за которым следует оператор «или» `||`, что означает следующую группировку операторов;

```
((($height > 2) || ($width == 1))
```

Полный перечень операторов и их приоритетов приведен в табл. 3.5. Первыми перечислены операторы с самым высоким приоритетом, далее – менее приоритетные операторы.

Там, где в одном предложении встречаются операторы с равным приоритетом, они группируются слева направо в порядке их появления. Например, сложение и вычитание имеют одинаковый приоритет и потому в следующем коде выполняются слева направо.

```
int $a = 2 - 3 + 4; // Результат 3
```

Не исключено, что в некоторых ситуациях вам придется подменять приоритет оператора, принятый по умолчанию. Для этого служит оператор группировки (). Как и в математике, он имеет наиболее высокий приоритет и, таким образом, доминирует над приоритетом любого другого оператора. В ранее приведенном коде оператор группировки может, к примеру, использоваться для того, чтобы обеспечить выполнение сложения *прежде*, чем вычитания.

```
int $a = 2 - (3 + 4); // Результат -5
```

Сомневаясь относительно приоритета оператора, всегда используйте оператор группировки, чтобы явно указать желаемый порядок вычисления выражения. Кроме того, оператор группировки поможет вам четко передать свои *намерения* тем, кто будет читать ваш код.

ТАБЛИЦА 3.5. ПРИОРИТЕТ ОПЕРАТОРОВ

Приоритет операторов

0, []

!, ++, --

*, /, %, ^

+, -

<, <=, >, >=

==, !=

&&

||

?:

=, +=, -=, ЧЛ

Оператор группировки можно использовать для обеспечения полного вычисления выражения до его передачи команде. Следующие операторы приводят к синтаксической ошибке.

```
int $a = 2;  
int $b = 1;  
print $a + $b;  
// Error: print $a + $b;  
//  
// Error: Line 4.10: Syntax error //  
// Ошибка: Стока 4.10: Синтаксическая ошибка //
```

Причина ошибки заключается в том, что последний оператор фактически вычисляется так:

```
(print $a) + $b;
```

Maya пытается прибавить к переменной `$b` результат выполнения команды `print`. Очевидно, что такой исход не предполагался. Чтобы явным образом выразить свои намерения, воспользуйтесь оператором группировки.

```
print ($a + $b);
```

3.2.5. Организация циклов

Если одну и ту же задачу требуется выполнить много раз, вам необходим **цикл**. Эту возможность предоставляют следующие конструкции языка.

For

Как бы вы рассчитали сумму элементов заданного массива чисел? По логике, процесс вычислений должен оперировать текущей суммой и прибавлять к ней каждое значение из массива. Чтобы обойти каждый элемент, используется цикл `for ("для")`. Следующий пример также служит иллюстрацией того, как вычислить произведение всех элементов.

```
float $nums[3] = { 2, 5, 6 };  
float $sum = 0;  
float $prod = 1;  
int $i;  
for( $i = 0; $i < size($nums); $i = $i + 1 )  
{
```

```
$sum = $sum + $nums[$i];
$prod = $prod * $nums[$i];
}
print $sum; // Результат: 13
print $prod; // Результат: 60
```

Оператор цикла `for` состоит из четырех частей.

```
for( вып_нач; условие_проверки; вып_кажд_итер)
    операция;
```

Оператор `вып_нач` вызывается один раз, перед запуском цикла. Оператор `условие_проверки` позволяет определить, когда цикл прекращает свою работу. Цикл завершается, если оператор возвращает ложное значение. Выполнение и проверка данного оператора производятся один раз до входления в цикл, а также в конце каждого полного прохода по нему. Оператор `вып_кажд_итер` запускается в конце каждого повтора цикла. На каждой итерации выполняется предложение `операция`.

While

Альтернативным вариантом цикла `for` является цикл `while` ("пока"). Он состоит всего из двух частей.

```
while( условие_проверки)
    операция;
```

Цикл `while` продолжает выполнение `операции` до тех пор, пока `условие_проверки` сохраняет истинное значение. Цикл `while` в точности повторяет цикл `for`, за исключением того, что он не содержит операторов `вып_нач` или `вып_кажд_итер`. Пользуясь циклом `while`, можно так переписать предшествующий оператор `for`:

```
float $nums[3] = { 2, 5, 6 };
float $sum = 0;
int $i = 0;
while( $i < size($nums) )
{
    $sum = $sum + $nums[$i];
    $i = $i + 1;
}
```

Do-While

Последний метод организации цикла- оператор **do-while**. Подобно оператору **while**, он состоит всего лишь из двух частей.

```
do
{
    операция;
} while(условие_проверки);
```

Оператор **do-while** работает точно так же, как и цикл **while**, за исключением того, что операция однократно выполняется до первого **условия_проверки**. Это позволяет, как минимум, один раз полностью выполнить цикл до проверки необходимости его завершения.

Бесконечные циклы

При работе со всеми методами организации циклов **очень** важным является правильное составление **условия_проверки**. Вполне возможно, что неудачно составленные операторы **условие_проверки** или **вып_кажд_итер** могут породить бесконечный цикл. Это происходит, если **условие_проверки** никогда не принимает ложное значение, а значит, цикл никогда не останавливается. В таком случае единственным выходом является аварийное завершение и повторный запуск процесса Maya. Лишь даже по этой причине так важно сохранять текущую сцену, а также любые рабочие сценарии перед их запуском.

Continue

Оператор **continue** позволяет пропустить оставшиеся операторы текущей итерации цикла и начать новую итерацию. Оператор **continue** можно использовать во всех циклах различных типов. Приведем пример использования **continue** в цикле **for**.

```
string $names[] = { "Jenny", "Bob", "Bill", "Paula" };
int $i;
for( $i=0; $i < size($names); $i = $i + 1 )
{
    if( $names[$i] == "Bill" )
        continue; // Не выводить имя Билла

    print ($names[$i] + "\n");
}
```

Результат:

Jenny
Bob
Paula

Break

Оператор `break` служит для немедленного выхода из цикла. Им можно воспользоваться в цикле любого типа. В следующем фрагменте кода работа цикла `while` прекращается в тот момент, когда одно из значений в массиве оказывается больше 10.

```
int $nums[] = { 2, 4, 7, 12, 3, 10 };
int $i = 0;
while( $i < size($nums) )
{
    if( $nums[$i] > 10 )
        break;

    print ($nums[$i] + " ");
    $i = $i + 1;
}
```

Результат:

2 4 7

Оператор `break` особенно полезен тогда, когда вы находитесь внутри серии вложенных условных операторов и хотите выйти из цикла, в котором они содержатся.

3.2.6. Процедуры

Как только вы приступите к написанию сравнительно длинных сценариев на языке MEL, вам станет труднее справляться с тем уровнем **сложности**, который вскоре возникнет. Удобный способ противостоять этой сложности состоит в разбиении сценария на более мелкие и легко управляемые части. Хотя **система** в целом, возможно, окажется достаточно сложной, отдельные ее компоненты вы **сможете** понять без особых проблем. Такой подход, заключающийся в разделении задачи компьютерного программирования на более мелкие части, известен как *структурное программирование*.

Принцип структурного программирования состоит в том, чтобы разбить программную задачу на мелкие фрагменты в соответствии с их функциональным назначением. Если бы вы писали программу изготовления тортов, то, вероятно, разработали бы ее как состоящую из двух частей. Первая часть имела бы дело с подготовкой полуфабриката. Вторая часть занималась бы собственно выпечкой. Если бы позднее вы решили создать систему, которая печет печенье, то вероятность того, что вам удалось бы еще раз использовать немалую часть «выпекающей» части программы, была бы достаточно велика.

Цель структурного программирования - разделить программу на функциональные части меньшего размера с возможностью их повторного применения. MEL позволяет решить эту задачу при помощи *процедур*. Процедура похожа на команду тем, что вы вызываете ее в своем сценарии, и она выполняет ту или иную операцию. Процедура способна принять на вход произвольное количество переменных. Помимо этого, она может выдать и некое выходное значение. Процедуру можно рассматривать как устройство, которое вы снабжаете информацией и которое обрабатывает ее с получением результата.

Далее приведена очень простая процедура, которая выводит на экран ваше имя.

1. Откройте редактор Script **Editor**.
2. Наберите на панели Command Input Panel следующий текст. Подставьте свое имя вместо имени David.

```
proc printName()  
{  
    print "My name is David";  
}
```

3. Нажмите клавиши **Ctrl+Enter** для выполнения.

Ничего не произошло. То, что вы сейчас сделали, - это лишь описание процедуры. Чтобы ее выполнить, вы должны ее вызвать.

4. Наберите, а затем выполните следующий текст:

```
printName;
```

Результат:

My name is David

Хотя эта процедура работает просто превосходно, что произойдет, если вы захотите вывести имя другого человека? Что если вы не знаете имени этого лица заранее? В таком случае спроектируйте процедуру, которая принимает имя на вход. После чего вы его напечатаете.

5. Наберите, а затем выполните следующие операторы:

```
proc printName( string $name )  
{  
    print ("My name is " + $name);  
}
```

Maya выдаст предупреждение следующего содержания:

```
// Warning: New procedure definition for "printName" has  
// a different argument list and/or return type. //  
// Предупреждение: Новое определение процедуры "printName" имеет  
// иной список аргументов и/или тип возвращаемого значения. //
```

Maya просто сообщает вам о том, что новая введенная вами процедура printName перекроет предыдущую версию.

6. Наберите, а затем выполните следующий текст:

```
printName( "Bill" );
```

Результат:

My name is Bill

Входным значением новой процедуры служит строковая переменная \$name. При этом вызове на вход процедуры передается значение "Bill". Переменная \$name принимает значение этой строки. После чего процедура выводит ее содержимое на экран.

Процедуры способны принимать на вход неограниченное число параметров. Наряду с этим, они могут возвращать выходное значение. В следующем упражнении описана процедура с именем sum, которая складывает два числа и возвращает результат.

7. Выполните следующий код:

```
proc int sum( int $a, int $b )  
{  
    int $c = $a + $b;  
    return $c;  
}
```

```
int $res = sum( 4, 5 );  
print $res;
```

Результат

9

Процедура `sum` принимает два входных целочисленных параметра `$a` и `$b` и возвращает единственное выходное значение целого типа.

Процедура `sum` является хорошим примером описания процедуры общего вида:

```
прис возвр_тип имя_процедуры( аргументы )
{
    операторы;
    возврат_результата;
}
```

При описании процедуры, которая возвращает результат, его тип должен быть указан в операторе `возвр_тип`. Если процедура не возвращает значения, то `возвр_тип` можно оставить пустым. `имя_процедуры` может быть произвольным. Однако в нем не могут содержаться пробельные символы, и первый знак не может быть цифровым. Процедура может не принимать `входных` параметров, а может принимать столько `параметров`, сколько необходимо. Типы и имена параметров перечисляются в операторе `аргументы`. Если процедура возвращает значение, она должна `содержать`, по крайней мере, один оператор `возврат_результата`. Если подобных операторов не обнаружится, Maya предупредит вас об этом. Такой оператор не обязательно должен находиться в конце процедуры, однако выход из нее будет невозможен без возврата значения результата. Если процедура не возвращает значения, оператор `возврат_результата` можно опустить.

Описание одной процедуры внутри другой невозможно.

3.2.7. Область видимости

Важно понимать то, что все переменные и процедуры имеют четко очерченную *область видимости*. Область видимости переменной определяет ее доступность из других областей сценария. Также она косвенно влияет на *время жизни* переменной. Каждая переменная характеризуется определенным периодом, в течение которого она доступна. За его пределами она не существует и потому является недоступной. Проще всего области видимости рассматривать в качестве блоков. Блоком считается раздел `кода`, содержащийся между скобками `(,)`. Следующий фрагмент является блоком:

```
(  
int $a;  
$a = 3 + 5;  
)
```

В любое время можно создать новый блок, поставив вокруг него пару скобок. Все переменные, описанные внутри данного блока, существуют только на протяжении времени его работы. Когда при достижении закрывающей скобки блок завершается, все переменные, которые в нем описаны, уничтожаются. А значит, доступ к ним уже невозможен. По этой причине переменная \$a, описанная в предыдущем фрагменте кода, уничтожается при достижении закрывающей скобки. Кроме того, одни блоки можно создавать внутри других.

```
// Блок 1
{
int $a = 12;

// Внутренний блок 2
{
    int $b = 10;
    print $a;
}

print$b;
```

При выполнении этого кода на экран будет выведена следующая ошибка:

```
// Error: print $b;
//
// Error: Line 7.9: "$b" is an undeclared variable. //
// Ошибка: Стока 7.9: переменная "$b" не описана. //
```

Переменная \$b оказалась неописанной потому, что она определена во внутреннем блоке, который уже завершился. Переменные, созданные в том или ином блоке, доступны и в блоках, внутренних по отношению к первому. Стало быть, переменная \$a, описанная в блоке 1, в данном случае доступна и во внутреннем блоке 2. Однако переменные, созданные во внутреннем блоке, недоступны для внешних блоков по той простой причине, что при завершении внутреннего блока все это переменные уничтожаются, а следовательно, внешний блок никогда не получит возможности обратиться к ним. Поэтому попытка вывести в блоке 1 значение переменной \$b терпит неудачу. К тому времени внутренний блок завершен, а значит, переменной \$b уже не существует.

Глобальные и локальные переменные

Переменные \$a и \$b считаются *локальными* переменными соответствующих блоков. Статус локальной переменной означает, что она доступна только в пределах своего блока и любого внутреннего блока, который может в нем находиться. Иногда возникает желание создать переменную, которая могла бы использоваться и другими блоками, описанными снаружи. К примеру, вы хотите запомнить имя текущего пользователя и сделать его доступным во всех своих сценариях. Именно для этой цели создана специальная область видимости - *глобальная*. Когда переменная описана как глобальная, она доступна в любое время в любом месте среды Maya. Описание глобальной переменной подразумевает ее размещение в «самой внешней» области видимости (вне каких-либо скобок) и использование ключевого слова global.

```
global string $currentUserName;
```

Чтобы получить доступ к глобальной переменной из блока, вам нужно сообщить MEL о том, что вы хотите к ней обратиться.

```
global string $currentUserName;
```

```
{  
    global string $currentUserName;  
    $currentUserName = "John Doe";  
}
```

Что произойдет, если вы не сообщите явно о своем намерении воспользоваться глобальной переменной?

```
global string $currentUserName;  
  
{  
    $currentUserName = "John Doe";  
}
```

MEL создаст внутри блока новую локальную переменную с именем \$currentUserName, и она будет использоваться вместо глобальной. Если же вы явно сообщите о намерении сослаться на глобальную переменную, MEL будет знать о том, какой переменной вы хотите воспользоваться. Применение локальной переменной предполагается всякий раз, когда иное не указано явно.

Инициализация глобальных переменных

Важно понимать то, как происходит инициализация глобальных переменных. Как и все переменные, они принимают значение по умолчанию, если вы не инициализировали их явно. Так как следующая переменная `$depth` имеет тип `int`, она принимает начальное значение 0.

```
global int $depth;
```

Кроме того, можно указать и явное значение переменной. Явное присваивание начальных значений переменных всегда предпочтительно. Такая практика делает непосредственно очевидным то, какое значение должна иметь переменная.

```
global int $depth = 3;
```

Важно знать и о том, что глобальные переменные инициализируются только *один раз*. Это происходит в тот момент, когда при компиляции MEL впервые встречает соответствующий оператор (все подробности компиляции сценариев изложены в разделе 3.3.4). По завершении компиляции все операторы, устанавливающие начальные значения глобальных переменных, игнорируются.

```
proc do()
{
    global int $depth = 3;
    print ("\n" + $depth);
    fcdepth = 1;
    print (" " + $depth);
}

go();
go();
```

На экран будет выведено следующее:

```
3 1
1 1
```

При первом вызове процедуры до ее требовалось скомпилировать, так что глобальная переменная `$depth` получила начальное значение 3. Когда процедура до вызывается еще раз, она уже находится в памяти и не нуждается в компиляции. Происходит обычный вызов. На сей раз используется текущее значение переменной `$depth`, и ее повторная инициализация не производится. Если бы процедура до требовала перекомпиляции, то глобальная переменная снова получила бы свое начальное значение.

Нельзя не отметить еще одно небольшое **различие**, касающееся инициализации глобальных и локальных переменных. Коль скоро все переменные блока создаются при входе в него, тогда же создаются и инициализируются локальные **переменные**. Например:

```
{  
    int $b; // Значение по умолчанию равно 0  
    int $a = 3 * $b;  
}
```

Создание и инициализация глобальных переменных предшествует выполнению любого другого кода. По сути, такие переменные можно инициализировать лишь значениями констант. При инициализации им нельзя, к примеру, присвоить значение результата вызова процедуры.

```
global int $a = 23; // OK  
global int $b = 1.5 * screenSize(); // Ошибка!
```

Конфликты

Все глобальные переменные существуют в едином пространстве (единой области видимости), поэтому не исключено, что два пользователя по незнанию применяют для обозначения своих глобальных переменных одно и то же имя. Первый из них пользуется этим именем в своих **сценариях**, второй - в своих. И ни один из них не знает о том, что теперь эта переменная может изменяться за пределами его сценария. Один пользователь может установить одно значение этой переменной, а затем второй может установить **другое**. Первый пользователь **предполагает**, что переменная содержит именно «его» значение, хотя это уже не так. Легко понять те проблемы, которые влечет за собой активность двух пользователей, изменяющих значение одной и той же переменной.

Пользуясь командой **env**, вы можете получить список всех глобальных переменных, известных системе в настоящий момент. Даже если вашей переменной нет в этом списке, это не означает, что конфликт не возникнет позже. Если другой пользователь загрузит сценарий, содержащий глобальную переменную с тем же именем, конфликт все же случится.

Лучшим **решением**, предотвращающим любые конфликты, является обеспечение уникальности имен переменных. В компании Alias Wavefront программисты присоединяют к началу всех глобальных переменных строчную букву **g**. Например:

```
$gScaleLimits
```

Поэтому, давая имя своей переменной, стоит воздержаться от такого соглашения об именах. Лучше предпослать каждому имени два или три символа, являющиеся уникальными для вас, вашей компании или вашего проекта. К примеру, все имена, которые входят в ваш код, разработанный для проектов по программированию Maya, вы можете начинать с `mp`:

```
global string $mpUserName;
```

Нельзя гарантировать что кто-нибудь другой не воспользуется тем же именем, однако подобный подход должен существенно сократить вероятность такого события.

Как только глобальная переменная описана, ее уже нельзя удалить. Причина этого заключается в том, что от существующей переменной может зависеть какой-то другой сценарий. Поэтому все глобальные переменные, описанные в ходе сеанса работы с Maya, остаются в памяти до его окончания. Если вы хотите удалить некоторые глобальные переменные, то единственный способ сделать это - выйти из Maya и запустить пакет снова.

Ранее было указано, что для получения перечня всех текущих глобальных переменных предназначена команда `env`. Чтобы проверить, обозначает ли данное имя существующую глобальную переменную, воспользуйтесь следующей процедурой.

```
global proc int isGlobal( string $varName )  
{  
    string $vars[] = `env`;  
    for( $var in $vars )  
    {  
        if( $var == $varName )  
            return yes;  
    }  
    return no;  
}
```

Эту процедуру можно использовать так:

```
if( isGlobal( "$myGlobal" ) )  
    ... сделать что-либо
```

Общая практика программирования свидетельствует о необходимости избегать использования глобальных переменных, пока они не станут абсолютно необходимы. Коль скоро эти переменные совместно используются *всеми* сцена-

риями и кодом MEL, они могут приводить к появлению *трудно* находимых программных ошибок. Гораздо лучше описывать переменные как локальные объекты сценария, и тогда возможность возникновения конфликтов будет отсутствовать.

Глобальные процедуры

Подобно переменным, процедуры тоже можно описывать как локальные или глобальные. Процедура является *локальной*, если не указано иное. Локальная процедура доступна лишь из того сценария, в котором она описана. К ней могут обращаться все процедуры в пределах этого сценария, тогда как процедуры из-за его пределов - нет. Такое правило позволяет создавать самодостаточные сценарии, не вызывающие потенциальных конфликтов с другими сценариями, где могут использоваться процедуры с таким же именем. Чтобы описать процедуру как глобальную, начните ее описание с ключевого слова *global*.

```
global proc printName( string $name )  
{  
    print $name;  
}
```

Теперь эту процедуру можно вызывать из любой точки Maya. Как и глобальные переменные, глобальные процедуры находятся в одном и том же *пространстве имен*, поэтому ничто не мешает другому сценарию также иметь в своем составе аналогичную процедуру. Процедура, код которой считан в память последним, перекрывает любые предыдущие описания. Для уменьшения вероятности такого события пытайтесь присваивать своим процедурам уникальные имена. К примеру, назовите свою процедуру не *printName*, а *dgPrintName*. Хотя гарантии того, что другой процедуры с таким же именем не существует, дать нельзя, это на деле снизит вероятность конфликта. Помните также о том, что при считывании исходного кода процедуры она *будет* постоянно находиться в памяти Maya. Сокращая число глобальных процедур, вы снизите *загрузку* памяти системы Maya. В общем случае, попытайтесь ограничить число глобальных процедур в своих сценариях минимально возможным.

Язык MEL обладает особым механизмом поиска глобальных процедур, если они еще не загружены в память. Работая с исходным кодом сценария и обнаружив процедуру, которая пока не известна, MEL пытается ее найти. Если, к примеру, вы вызываете процедуру *createSpline()*, а она еще не описана в текущем сценарии и не является глобальной, Maya пытается найти файл сценария с именем *createSpline.mel*. Для этого выполняется поиск по всем каталогам сценарiev. Если файл найден, он считывается в память.

Редактор сценариев

Немаловажно помнить о том, что все переменные и процедуры, которые вы описываете в редакторе сценариев **Script Editor**, *автоматически* становятся глобальными. Это происходит даже **тогда**, когда вы не указываете ключевое слово *global*. Такое положение дел может не всегда совпадать с вашими намерениями, особенно когда вы экспериментируете с редактором. К счастью, есть способ это обойти. Вспомнив о том, что все переменные, описанные **внутри** блока, автоматически уничтожаются при его **завершении**, вы можете создать фрагмент кода, который удалит все, что в нем было описано. Поэтому при написании кода в редакторе **Script Editor** просто поместите его в скобки, чтобы создать блок.

В отношении процедур это невозможно. Поскольку они описываются в среде **Script Editor**, понятия локального сценария не существует. По существу, все процедуры, описанные в **Script Editor**, автоматически становятся глобальными, и избежать этого никак нельзя.

3.2.8. Дополнительные методы выполнения команд

Команды **MEL** можно выполнять, пользуясь для этого множеством различных механизмов. В следующем примере показано, как несколькими способами выполнить одну и ту же команду.

1. `currentTime -query;`
2. `'currentTime -query';`
3. `eval("currentTime -query");`

Важно понять то, что команда по-прежнему выполняет одно и то же действие, независимо от того, как был инициирован ее запуск. Разные методы выполнения продемонстрированы лишь для того, чтобы предоставить вам **большую** гибкость в управлении этим процессом. В первом случае команда выполняется, но не возвращает результат. Этот метод выполнения наиболее типичен. Во втором случае весь текст команды помещается между символами обратных кавычек `(')`. Команда выполняется, как и прежде, но на этот раз ее результат возвращается. В последнем случае она вызывается при помощи другой команды- **eval**. Команда **eval** позволяет передать ей в виде строки целую цепочку команд языка **MEL**, а затем выполнить их. Она также возвращает результат.

Результаты команд

Необходимо различать способы получения результатов команд. Получение **значений**, возвращаемых процедурами, совершенно очевидно. Если, к примеру, у вас есть процедура `getUserName()`, которая возвращает имя текущего пользователя, вы можете вызвать ее следующим образом:

```
$userName = getUserName();
```

Значение, возвращаемое процедурой, будет присвоено переменной `$userName`. Однако если вы хотите получить выходное значение команды, вызывать ее напрямую нельзя. Команда `currentTime` служит для запроса текущего времени. Используя ее непосредственно, как показано ниже, вы получите ошибку.

```
$time = currentTime -q;
```

Команда должна помещаться в одинарные обратные кавычки (').

```
$time = `currentTime -q`;
```

Тогда Maya произведет вычисления над текстом в кавычках и вернет результат. Затем его можно сохранить в переменной `$time`.

Распространенная оплошность - нечаянно поместить оператор в апострофы, или одинарные прямые (), а не обратные кавычки {}. Применение апострофов ведет к синтаксической ошибке.

Команда eval

Язык MEL содержит очень мощное, но малоизвестное средство - команду eval. Она позволяет «на лету» формировать цепочки команд и выполнять их средствами Maya. Это означает, что ваши сценарии на языке MEL могут генерировать операторы, которые затем выполняются. В других неинтерпретируемых языках подобной возможности не существует. Взамен вам пришлось бы написать код, который затем прошел бы обычные стадии компиляции-сборки, делающие новый код доступным для выполнения. Рассмотрим следующий оператор, где складываются два числа.

```
float $res = 1 + 2;
print $res;
```

Что если вы хотите разрешить своим пользователям производить собственные вычисления? Поскольку заранее вы не знаете, каковы могут быть их желания, непосредственно в виде сценария вам не удастся записать ничего. Однако вы можете воспользоваться командой eval, предназначенней для выполнения пользовательских операторов во время работы. Далее описана процедура, которая принимает на вход оператор, а затем выполняет его.

```
global proc calc ( string $statement )
{
    float $res = eval("float $_dummy = " + $statement);
    print ("\n" + $statement + " = " + $res);
}
```

Теперь пользователи могут вводить в окне Script Editor и выполнять следующие операторы:

```
calc( "1 + 2" );
calc( "10 * 2 / 5" );
```

Процедура выведет такие результаты:

```
1 + 2 = 3
10 * 2 / 5 = 4
```

Операторы были сформированы «на лету», после чего выполнены. По сути дела, вы можете написать свой собственный сценарий, позволяющий «на лету» генерировать программные инструкции и выполнять их во время работы. Несмотря на то что данный пример очень прост, сложность операторов, которые могут быть выполнены, не ограничена. Вы даже можете создать сценарий, который породит целую программу как цепочку операторов языка MEL, а затем по команде eval ее выполнит.

3.2.9. Эффективность

Так как MEL является интерпретируемым языком, он должен транслировать инструкции сценария в реальном масштабе времени. По существу, эти издержки на интерпретацию могут заставить его работать медленнее по сравнению с компилируемыми языками. Имеется, однако ряд практических приемов программирования, которых вы можете придерживаться и которые могут увеличить скорость ваших сценариев на языке MEL.

Явное объявление типов

Прежде всего, указывайте тип переменных при их объявлении. Поэтому описанию переменной вида

```
$objs = 'selectedNodes';
```

следует предпочесть явное объявление типа:

```
string $objs[] = 'selectedNodes';
```

Это увеличит скорость работы MFX, поскольку интерпретатору не придется определять тип переменной на основании присваиваемого ей значения. Кроме того, **MEL** сможет лучше выполнять проверку типов.

Размеры массивов

Если вы используете массивы, то сможете добиться большей эффективности, если укажете количество элементов, которое они будут содержать, во время их описания. Это позволит избежать необходимости изменять их размер в дальнейшем средствами языка **MEL**. Даже если вы не знаете заранее точного числа элементов в массиве, стоит все-таки сделать хотя бы грубое предположение. Вероятно, оно окажется ближе к истине, чем принятый в **MEL** по умолчанию размер массива, равный 16. Размер по умолчанию равен 16 потому, что таково количество чисел с плавающей запятой, образующих матрицу преобразования (4 x 4). Если, к примеру, вы знаете, что максимальное число элементов примерно равно 50, укажите этот размер.

```
int $nums[50];  
int $avgs[]; // Принимается размер по умолчанию, равный 16 элементам
```

Установка размера массива не запрещает вам добавлять в него элементы в будущем. Эффективность, достигаемая за счет задания начального размера, обусловлена тем, что **MEL** не должен выполнять множество операций по изменению размера области памяти при добавлении элементов, что повышает скорость выполнения.

3.2.10. Дополнительные конструкции языка

Следующие программные конструкции и операторы языка **MEL** не являются абсолютно необходимыми для написания полнофункциональных сценариев; однако вы можете встретить их в текстах, написанных другими разработчиками, и потому вам следует с ними ознакомиться. Многие из этих конструкций приводятся лишь потому, что делают некоторые операторы более лаконичными, но при этом они не выполняют ничего такого, чего нельзя было бы достичь при помощи программных конструкций, которые были представлены ранее.

Цепочки присваивания

Одно и то же значение можно присвоить более чем одной переменной, объединив эти переменные в цепочку.

```
int $score, $results;  
$score = $results = 253; // То же, что $score = 253; $results = $score;
```

Подставляемые арифметические операторы

Выполняя основные арифметические операции с участием переменных, в которых будет храниться результат вычислений, можно использовать следующие операторы. Оператор сложения-присваивания предназначен для прибавления к переменной некоторого значения и сохранения в ней результата.

```
$score += 3.2; // То же, что $score = $score + 3.2
```

Прочие арифметические операторы имеют аналогичный вид.

```
$score *= 2;  
$score -= 4;  
$score /= 1.3;  
$score %= 3;
```

Операторы инкремента и декремента

Во время простого счета к переменной добавляется 1 или вычитается из нее. Данную операцию можно записать короче, пользуясь операторами инкремента и декремента.

```
$score++; // То же, что $score = $score + 1  
$score--; // То же, что $score = $score - 1
```

Эти операторы также можно применять в *префиксной* и *постфиксной* форме. В зависимости от формы оператора возникают различные побочные эффекты, связанные с присваиванием значения. Во избежание путаницы на такие подобные эффекты лучше всего не полагаться.

Условный оператор

УСЛОВНЫЙ оператор (`? :`) можно использовать в целях сокращения синтаксически длинных операторов до более коротких. Следующий код:

```
int $a = 3;  
if( $a > 1 )  
    $score = 100;  
else  
    $score = 50;
```

можно при помощи условного оператора сократить до двух простых строк:

```
int $a = 3;  
$score = ($a > 1) ? 100 : 50;
```

Если сравнение дает истинный результат, то переменная в левой части принимает значение, следующее за символом ?. Если результат ложный, переменная принимает значение справа от символа : .

Оператор **switch**

Оператор **switch** используется для преобразования конструкции, которую вы знаете как цепочку операторов **if**, в более сжатую и удобочитаемую форму. Рассмотрим следующую последовательность операторов;

```
int $a = 3;  
if( $a == 1 )  
    print "a is 1";  
else  
{  
    if( $a == 2 )  
        print "a is 2";  
    else  
{  
        if( $a == 3 )  
            print "a is 3";  
        else  
            print "a is another number"; // "a имеет другое значение"  
    }  
}
```

Все эти операторы сравнивают исходную переменную \$a с некоторым известным значением. Оператор **switch** делает этот тип операторов более удобным для чтения и понимания. При помощи оператора **switch** тот же код можно переписать так:

```
int $a = 3;  
switch( $a )  
{  
case 1:  
    print "a is 1";  
    break;  
  
case 2:
```

```
print "a is 2";
break;

case 3:
    print "a is 3";
    break;

default:
    print "a is another number";
    break;
}
```

Значение \$a участвует в сравнении только один раз. В зависимости от значения переменной выполняется один из следующих далее операторов case. Каждый из них определяет, какие значения он обрабатывает. Первый оператор case, а именно case 1, выполняется в том случае, когда \$a равно 1.

В конце каждой секции case расположен оператор break. Он ставится для того, чтобы предотвратить продолжение выполнения следующего оператора case. Фактически break производит безусловный переход за пределы оператора switch.

Последний оператор, обозначенный как default, – это «ловушка» для всех прочих случаев. Он содержит операторы, которые выполняются в той ситуации, когда значение не соответствует ни одному другому оператору case.

Возможны случаи, когда вам захочется опустить break для данного оператора case. Это имеет место, если несколько различных значений следует обработать одинаковым образом. В следующем примере, когда \$a равно либо -1, либо -2, выполняется оператор print "below zero". Для значений 1 и 2 выполняется оператор print "above zero".

```
int $a = 2;
switch( $a )
{
    case -1:
    case -2:
        print "below zero"; // меньше нуля
```

```
break;

case 0:
    print "zero";           // нуль
    break;

case 1:
case 2:
    print "above zero";   // больше нуля
    break;
}
```

Оператор for-in

Оператор `for-in` представляет собой краткую форму записи итерации по элементам массива. В то время как перебор элементов массива выполняется при помощи цикла `for`:

```
int $nums[4] = { 4, 7, 2, 3 };
int $elem;
int $i;
for( $i = 0; $i < size($nums); $i++ )
{
    $elem = $nums[$i];
    выполнить какие-либо действия с $elem
}
```

при помощи цикла `for-in` этот код можно сократить до следующего:

```
int $nums[4] = { 4, 7, 2, 3 };
int $elem;
for( $elem in $nums )
{
    выполнить какие-либо действия с $elem
}
```

3.3. Создание сценариев

Для решения любых задач, за исключением самых простых, вам будет недостаточно написать один оператор MEL. Несмотря на то что строка **Command Line** идеально подходит для запуска коротких команд, а редактор **Script Editor** вполне годится для выполнения нескольких операторов, наступает момент, когда вы должны составлять более сложные тексты на языке MEL. Такие тексты лучше всего сохранять в виде сценариев. Сценарий MEL – это всего лишь текстовый файл, куда входят все операторы. Если они сохранены в виде файла, то их можно многократно вызывать, когда это потребуется, не выполняя ввод каждый раз заново. Кроме того, создание сценариев открывает возможность использования большего диапазона функций MEL, включая описание собственных процедур.

3.3.1. Текстовый редактор

Так как сценарии на языке MEL представляют собой обычные текстовые ASCII-файлы, то для их написания и правки вы можете использовать практически любой текстовый редактор. При этом важно, чтобы создаваемые файлы имели простой текстовый формат. Такие приложения, как Microsoft Word, автоматически помещают в итоговый файл элементы форматирования и прочую «скрытую» информацию⁵. Ваш редактор должен уметь записывать обычный текстовый ASCII-файл, не содержащий никакой форматной информации.

По возможности выберите редактор, отображающий текущую позицию каретки или курсора в строке и в столбце. При обнаружении ошибки в сценарии MEL способен указать номер строки и столбца, где содержится некорректный оператор. Тогда вы сможете быстро перейти к этой строке и этому столбцу, если ваш редактор поддерживает такую возможность.

Кроме того, убедитесь в том, что текстовый редактор использует моноширинный шрифт. Это означает, что ширина всех символов одинакова. Если это не так, вам будет трудно задать схему размещения операторов, поскольку одни символы окажутся уже или шире других.

⁵ Эта проблема решается путем элементарной настройки Microsoft Word. Чтобы установить действующий при сохранении документов формат файла по умолчанию, обратитесь к пункту главного меню **Сервис | Параметры...**, выделите в окне диалога вкладку **Сохранение**, выберите в выпадающем списке **Сохранять файлы Word как:** элемент **Только текст (*.txt)** и нажмите кнопку **OK**, - Примеч. перев.

Если у вас нет текстового редактора, то вы можете использовать **Script Editor**, однако я рекомендую его для составления лишь самых простых сценариев. Команды можно набирать в области **Command Input Panel**, а затем выполнять их. Выполненные команды сразу же перемещаются на панель **History Panel**, поэтому их можно скопировать, после чего вставить в **Command Input Panel** для повторного запуска. Это может быть утомительно, если не сказать больше. Кроме того, на различных платформах многие из «горячих клавиш» редактора **Script Editor** (копирование, вырезание, вставка и т. д.) отличаются. По этой причине, а также из-за того что возможности **Script Editor** в области редактирования текстов существенно ограничены, возможно, вам лучше научиться пользоваться другим редактором.

3.3.2. Сохранение сценариев

Чаще всего сценарии сохраняются в виде текстовых файлов с расширением **.mel**. Притом что расширение имени файла может быть любым, лучше всего пользоваться именно этим. Если имя файла не имеет расширения, Maya предполагает наличие расширения **.mel**.

Каталог сценариев по умолчанию

Допустим, вы создали файл сценария, и его нужно поместить в каталог, где Maya сможет его найти. Maya ищет пользовательские сценарии в нескольких разных местах. Корневой путь, которым пользуется Maya в начале поиска ваших сценариев, основан на регистрационном имени пользователя. В Unix это просто *username*, в Windows – имя, под которым пользователь «ходит в систему». В следующем примере *user* – это каталог с настройками пользователя. **Буква_диска** – буква жесткого диска, где хранятся настройки. Полученный при их объединении результат представляет собой **каталог пользователя**.

Windows 2000:

буква_диска:\Documents and Settings\user\

Windows NT:

буква_диска:\winnt\profiles\user\

Unix\Linux:

Mac OS X:

~/Library/Preferences/

Находясь в каталоге *пользователя*, Maya создаст в нем свой каталог *maya* и соответствующие подкаталоги. Все версии Maya обращаются к общему каталогу *scripts*. В нем находятся используемые вами сценарии для *всех* установленных версий Maya. Если у вас есть сценарии, работающие независимо от того, какая версия Maya запущена, вы можете разместить их в этом каталоге. Помещать сценарии в этот каталог следует лишь в случае уверенности в том, что они смогут выполняться под любой версией Maya, возможно, включая и будущие. На различных платформах общий каталог *scripts* занимает следующее положение.

Windows 2000:

буква_диска:\Documents and Settings\user\My Documents\maya\scripts

Windows NT:

буква_диска:\winnt\profiles\user\maya\scripts

Unix\Linux:

~/maya/scripts/

Mac OS X:

~/Library/Preferences/AliasWavefront/maya/scripts/

Вообще говоря, вы можете сохранить свои сценарии и в каталоге *scripts* текущей версии Maya, с которой работаете. Этот каталог включает явно указанный номер версии, например *maya\4.0\scripts*. Расположив сценарии в каталоге конкретной версии, вы гарантируете отсутствие несовместимости с более ранними или поздними версиями пакета. В следующих примерах обозначим используемую версию Maya как *x.x*.

Windows 2000:

буква_диска:\Documents and Settings\user\My Documents\maya\x.x\scripts

Windows NT:

буква_диска:\winnt\profiles\user\maya\x.x\scripts

Unix\Linux:

~/maya/x.x/scripts/

Mac OS X:

~/Library/Preferences/AliasWavefront/maya/x.x/scripts/

Когда Maya получает запрос на выполнение сценария, она начинает с просмотра независимого от версий каталога `maya\scripts`, а затем переходит к каталогу, зависимому от версий: `\maya\x.x\scripts`.

Чтобы узнать каталог script текущего пользователя, выполните команду `internalVar`.

```
string $dir = `internalVar -userScriptDir`;
```

Собственные каталоги сценариев

Если вы хотите, чтобы ваши сценарии лежали в вашем собственном каталоге, нужно лишь установить переменную окружения с именем `MAYA_SCRIPTS_PATH`. Хотя понятие переменных окружения существует на **всех** платформах, конкретная реализация и методы их использования могут отличаться. Если ваши сценарии, к примеру, находятся в каталоге `c:\myScripts`, путь к ним можно добавить к переменной окружения следующим образом:

```
set MAYA_SCRIPTS_PATH=$MAYA_SCRIPT_PATH;c:\myScripts
```

Если Maya ищет какой-нибудь сценарий, она просматривает все сценарии, перечисленные в переменной `MAYA_SCRIPTS_PATH`, а затем обращается к местам хранения по умолчанию, речь о которых шла выше.

Поместив следующую строку в файл `maya.env`, вы тоже можете установить названную переменную окружения. Заметьте, что в среде Mac OS X имя этого файла пишется как `Maya.env`, с заглавной буквой *M*.

```
MAYA_SCRIPTS_PATH=собственный_каталог_сценариев
```

Если файла `maya.env` не существует, просто создайте его. Файл следует сохранить в главном пользовательском каталоге `maya` с учетом текущей версии пакета. Этот каталог таков:

Windows 2000:

```
буква_диска:\Documents and Settings\user\My Documents\maya\x.x\
```

Windows NT:

```
буква_диска:\winnt\profiles\user\maya\x.x\
```

Unix\Linux:

```
~/maya/x.x/
```

Mac OS X:

```
~/Library/Preferences/AliasWavefront/maya/x.x/
```

Чтобы, находясь в среде Maya, выяснить текущее положение файла `maya.env`, воспользуйтесь командой `about`.

```
string $envFile = `about -environmentFile`;
```

Автоматический запуск

Всякий раз, когда вы запускаете Maya, вам может потребоваться выполнение серии команд или операторов. С этой целью создайте файл `userSetup.mel` и поместите его в собственный каталог `scripts`. Во время старта Maya выполнит все команды, которые в нем перечислены.

Немаловажно отметить, что после выполнения сценария `userSetup.mel` сцена Maya очищается. Поэтому любые изменения, внесенные в сцену посредством этого сценария, будут утрачены.

3.3.3. Подготовительный этап разработки

К сожалению, Maya не обладает комфортной интегрированной средой разработки (IDE) для создания сценариев. Однако ценой небольших усилий можно подготовить собственную среду, которая поможет автоматизировать многие задачи разработки. Этот начальный этап вполне будет стоить затраченных усилий, когда дело дойдет до составления большого числа сценариев.

Обязательно выполните эти шаги, так как полученная в результате среда разработки найдет применение в следующих разделах.

Первый шаг - автоматизация процесса открытия сценария MEL для редактирования.

1. Выберите пункт меню **File | New Scene**.

2. Откройте редактор **Script Editor**.

3. Введите следующий текст:

```
string $scriptsPath = `internalVar -userScriptDir`;
chdir $scriptsPath;
```

4. В зависимости от используемой операционной системы, добавьте следующую строку:

```
Irix/Linux:
system( "editor learning.mel >/dev/null 2>&1 &" );
```

Windows:

```
system( "start editor learning.mel" );
```

Вместо строки *editor* подставьте имя вашего текстового редактора и путь к нему. Например, в среде Windows вы можете заменить строку *editor* строкой *notepad.exe*.

Теперь в окне **Script Editor** должно быть три строки текста.

5. Для проверки команд нажмите клавиши **Ctrl+Enter**.

Если все прошло хорошо, текстовый редактор должен открыться. В зависимости от того, каким редактором вы пользуетесь, на экране может появиться запрос на создание файла *learning.mel*.

6. Сохраните пустой файл, после чего закройте редактор.

7. На панели **History Panel** редактора **Script Editor** выделите текст двух команд, которые только что были выполнены.

8. Левой кнопкой мыши перетащите текст на панель **Shelf**.

На панели появится кнопка, содержащая выделенный текст.

9. Щелкните по вновь созданной кнопке **Shelf**.

Будет запущен текстовый редактор, и в нем откроется файл сценария *learning.mel*. Новая кнопка на панели **Shelf** позволяет быстро открывать сценарий и приступать к его редактированию. Далее будем называть ее кнопкой **Edit Script** (Редактировать сценарий).

10. Поместите в файл *learning.mel* следующий текст:

```
sphere;  
move 1 0 0;
```

11. Сохраните файл сценария.

12. В редакторе **Script Editor** наберите следующую команду:

```
source learning;
```

Сценарий можно считать в память, не указывая расширение *.mel*. Файл с полным именем *learning.mel* будет найден **Maya** автоматически. Чтение файлов описано чуть позднее.

13. Для выполнения команды нажмите клавиши **Ctrl+Enter**.

Будет построена сфера, которая затем сместится на одну единицу вдоль оси *x*.

14. На панели **History Panel** редактора **Script Editor** выделите предыдущую выполненную команду.

15. Левой кнопкой мыши перетащите текст на панель Shelf.

На панели появится новая кнопка с текстом команды. Она позволит загружать сценарий и выполнять его. Далее будем называть ее кнопкой Execute Script (Выполнить сценарий).

16. Выберите пункт меню File | New Scene.

17. Щелкните по кнопке Execute Script.

Сценарий `learning.mel` будет загружен, а затем выполнен. Результатом станет создание и перемещение сферы.

Даже несмотря на то что такая установка не заняла много времени, теперь вы можете быстро загружать, редактировать и выполнять сценарий, всего несколько раз щелкнув кнопкой мыши.

3.3.4. Чтение в память

В предыдущем примере вы прочитали в память сценарий `learning.mel`. Понимание всех деталей процесса *считывания в память* сценария MEL является очень важным. Сценарий MEL существует в виде текстового файла на жестком диске и недоступен для Maya до тех пор, пока вы его не считаете. В ходе чтения файла производятся следующие действия.

- ◆ В порядке перечисления выполняются все команды на языке MEL.
- ◆ В память загружаются все глобальные процедуры. Их выполнение откладывается до момента явного вызова.

Как только файл сценария MEL считан, он уже не используется, и обращение к нему не происходит. Maya сделала все, что предписывает файл сценария, и теперь содержит в своей памяти все глобальные процедуры. Согласно *одному* из самых распространенных заблуждений, Maya якобы автоматически распознает изменения в сценарии, если после их внесения он был сохранен. Этого не произойдет, пока вы не считаете сценарий в память еще раз, что заставит Maya выполнить и занести его в память, заменив любую более старую версию, которая могла быть считана ранее. По сути дела, важно, чтобы вслед за изменением сценария вы сразу же его считали в память. Если вы не произведете считывание только что измененного сценария, Maya продолжит работать со старым сценарием, находящимся в ее памяти.

Обычный цикл разработки протекает так. Сначала нужно создать файл сценария и считать его в память. Проверить результаты работы. Исправить сценарий и сохранить изменения. Снова считать файл сценария, проверить результаты

работы. Исправления и повторное считывание продолжаются до тех пор, пока сценарий не будет полностью отложен и протестирован.

Важно помнить о том, что сценарий не считывается автоматически при загрузке сцены, в которой он используется. Если вы не считаете файл сценария явным образом, команды и процедуры, которые в нем описаны, будут недоступны.

Если сценарий, которые вы хотите считать, расположен в одном из известных каталогов, отведенных для хранения сценариев MEL (см. 3.3.2 «Сохранение сценариев»), просто укажите его название.

```
source learning.mel;
```

Вообще говоря, имя файла сценария лучше всего всегда помещать в двойные кавычки (""). Это гарантирует правильность обработки имен файлов, содержащих полный путь или пробелы.

```
source "learning.mel";
```

Если сценарий находится в неизвестном Maya каталоге, нужно указать полный путь к нему,

```
source "c:\\myScripts\\learning.mel";
```

Если имя файла сценария не имеет расширения, Maya автоматически присоединит к нему расширение .mel.

3.3.5. Отладка и тестирование

К сожалению, в поставку Maya не входит интерактивный отладчик. Однако есть несколько советов, которые помогут вам обнаруживать в своих сценариях синтаксические и логические ошибки.

Сохраняйте сценарий перед его выполнением!

Прежде чем выполнить сценарий или любые команды языка MEL, для которых этот совет имеет значение, проверьте, сохранили ли вы текущую сцену, а также другие нужные файлы Maya. Остановить выполнение сценария или последовательности команд MEL в интерактивном режиме невозможно. Если сценарий вошел в бесконечный цикл или инициировал долгий вычислительный процесс, его нельзя отменить. Единственный способ остановить сценарий - полностью завершить процесс Maya. Тогда у вас не будет возможности сохранить текущую сцену Maya, поэтому все несохраненные изменения будут потеряны.

Если сценарий завершается, но дает неверные результаты, вы можете попытаться аннулировать его работу. Однако в ходе своего выполнения сценарий мог вызывать определенные операции, которые не поддерживают режим отмены,

так что откат к начальной сцене может быть невозможен. Если же сцена была сохранена заранее, ее можно просто открыть заново и снова запустить сценарий,

Еще более надежный подход состоит в том, чтобы сделать копию исходной сцены, а затем пользоваться этой копией в ходе работы над сценарием. Таким образом, вы никак не коснетесь исходной сцены, а значит, ее нельзя будет случайно изменить. Сценарий можно применять к исходной сцене лишь **тогда**, когда он проверен и работает нормально.

Отображение номеров строк

Если вы не самый удачливый программист на свете, то ваши сценарии не всегда будут компилироваться и выполняться без ошибок с первого раза. Наиболее распространенные ошибки - синтаксические. Синтаксическими называют такие ошибки, при возникновении которых набранный вами текст кажется интерпретатору бессмысленным, поэтому он выводит надлежащее сообщение и заканчивает свою работу. Далее рассмотрены шаги, связанные с обнаружением и исправлением синтаксических ошибок.

1. Щелкните по кнопке **Edit Script**.

2. Удалите точку с запятой после команды `sphere`, с тем чтобы файл выглядел следующим образом:

```
sphere  
move 1 0 0;
```

Такое исправление, очевидно, ведет к синтаксической ошибке, поскольку между двумя командами нет точки с запятой, которая их разделяет.

3. Сохраните файл сценария.

4. Щелкните по кнопке **Execute Script**.

В строке **Command Feedback** будет показан следующий текст. Вывод текста на красном фоне указывает на сообщение об ошибке.

```
// Error: No object matches name: move //  
// Ошибка: Нет объекта, соответствующего имени move //
```

Так как вы не включили в текст точку с запятой, Maya предположила, что команда `move` вызвана как один из аргументов команды `sphere`. Такая проблема достаточно легко поддается отладке из-за небольших размеров сценария. В гораздо более длинных и сложных сценариях, которые содержат существенно большее число строк, ошибки находить сложнее. К счастью, Maya позволяет указывать, в какой строке произошла синтаксическая ошибка.

5. В меню редактора **Script Editor** выберите пункт **Script] Show Line Numbers** (Сценарий | Показывать номера строк).
6. Щелкните по кнопке **Execute Script**.

```
// Error: No object matches name: move //
// Error: file: D:/Documents and Settings/davidgould/My Documents/maya/
4.0/scripts/learning.mel line 2: No object matches name: move //
// Ошибка: Нет объекта, соответствующего имени move //
// Ошибка: файл: D:/Documents and Settings/davidgould/My
Documents/maya/
4.0/scripts/learning.mel строка 2: Нет объекта, соответствующего имени
move //
```

На этот раз сгенерировано более полное сообщение об ошибке. В дополнение к номеру строки, где находится оператор, содержащий ошибку, Maya приводит полный путь к файлу сценария.

Функция **Show Line Numbers** также включает номер столбца в строке, где произошла ошибка. Возьмем, к примеру, сообщение об ошибке с такой ссылкой: line 12. 31. Эта ошибка относится к 31-му столбцу 12-й строки. Обладая такой дополнительной информацией, вы сможете более точно указать местоположение ошибки. Maya сохраняет установки признака **Show Line Number** между сессиями работы, так что эта функция будет активна и при следующих запусках.

Трассировка выполнения сценария

В среде Maya имеется окно просмотра переменных, а значит, в процессе выполнения сценария вы можете наблюдать их текущие значения. Контрольные точки устанавливать нельзя, поэтому отслеживайте текущее значение переменной в каждой точке сценария. Вместо контрольных точек должен использоваться старый, испытанный и проверенный метод вывода на экран подробных диагностических сообщений. В зависимости от того, хотите ли вы, чтобы результаты диагностики отображались в главном окне Maya или заносились в стандартный поток сообщений об ошибках [**Output Window** (Окно вывода) в среде Windows], можно использовать, соответственно, команды `print` или `trace`.

Команда `print` способна выводить на экран значения переменных различных типов, а также массивов. При этом нет необходимости в использовании строковых описаний форматов, как это делается в языке C.

```
int $i = 23;  
print $i; // Результат: 23  
vector $b = << 1.2, 3.4, 7.2 >>;  
print $b; // Результат: 1.2 3.4 7.2  
int $nums[3] = { 3, 6, 7 };  
print $nums; // Результат: 3 6 7 (каждое значение на новой строке)
```

Воспользовавшись дополнительными преимуществами простого сцепления строк, вы можете быстро разработать серию диагностических операторов вывода.

```
int $i = 23;  
print ("\\nThe value of $i is " + $i ); // "Значение $i равно "  
string $name[] = `cone`;  
print ("\\nThe name of the cone is " + $name[0] ); // "Имя конуса - "
```

Для вывода строки в стандартный выходной поток служит команда **trace**. Под Windows стандартный поток отображается в окне **Output Window**. На других платформах этот выходной поток можно перенаправить в файл, загрузив Maya из командной строки. В следующем примере стандартный вывод перенаправляется в файл **stdout.txt**:

```
maya > stdout.txt
```

Пользуясь этим методом и внедряя отладочные операторы в различные фрагменты кода, вы можете сохранить полный отчет о выполнении своего сценария. Пользователям Windows достаточно щелкнуть правой кнопкой мыши в окне **Output Window** и выбрать пункт меню **Select All** (Выбрать все). Скопируйте текст и поместите его в файл.

Команда **trace** работает так же, как и команда **print**.

```
trace "\\nAt the start of printName"; // "При запуске printName"  
int $a = 33;  
trace ("The value of $a is " + $a); // "Значение $a равно "
```

Еще одним преимуществом команды **trace** является ее способность показать имя файла и номер строки сценария, где произошел вызов. Это существенно облегчает отыскание мест расположения операторов отладки. **Если** бы сценарий **learning.mel** содержал, к примеру, следующие операторы:

```
int $a = 65;  
trace -where ("The value of $a is " + $a);
```

то команда trace вывела бы такие сведения:

```
file: D:/Documents and Settings/davidgould/My Documents/maya/4.0/scripts/
      learning.mel line 3: The value of $a is 65
файл: D:/Documents and Settings/davidgould/My Documents/maya/4.0/scripts/
      learning.mel строка 3: Значение $a равно 65
```

Это значительно облегчает поиск конкретного вызова команды отладки, который произвел вывод данной информации.

Вывод предупреждений и ошибок

Чтобы сигнализировать о появлении предупреждения или возникновении ошибки, используйте, соответственно, команды warning и error. По общему соглашению, команда **warning** применяется в том случае, когда происходит некритическая ошибка и сценарий может продолжить свою работу. Если же возникает фатальная ошибка и сценарий не может выполняться дальше, а потому должен быть завершен, используется команда **error**. Обе эти команды выводят сообщение в окне команд Maya. Когда Maya выполняется в пакетном режиме, оба типа ошибок выводятся в стандартный поток сообщений об ошибках. Обычно это та же оболочка или то же окно команд, откуда запущен пакет.

Команда **warning** выводит текст «Warning:» («Предупреждение:»), за которым следует заданное сообщение. В строке **Command Feedback** этот текст отображается пурпурным цветом.

```
warning "Incorrect size given, using default.";
// "Указан неверный размер, используется значение по умолчанию."
```

Результат:

```
Warning: Incorrect size given, using default.
```

Команда **error** выводит текст «Error:» («Ошибка:»), за которым следует заданное сообщение. В строке **Command Feedback** этот текст отображается красным цветом.

```
error "Unable to load image."; // "Невозможно загрузить изображение."
```

Результат:

```
Error: Unable to load image.
```

В случае вызова команды **error** выполнение сценария немедленно прекращается и возникает исключительная ситуация. После этого делается полная развертка стека выполнения.

Также вы можете получить полный путь к файлу сценария и номеру строки, соответствующей вызову команды вывода информации. В основном это делается в целях отладки.

```
warning -showLineNumber true "Missing data file";
```

Результат:

```
// Warning: file: D:/Documents and Settings/davidgould/My Documents/maya/4.0/scripts/learning.mel line 3: Missing data file //
// Предупреждение: файл: D:/Documents and Settings/davidgould/My Documents/maya/
4.0/scripts/learning.mel строка 3: Отсутствующий файл данных //
```

Наряду с этим, опция `-showLineNumber` может применяться в окончательных вариантах сценариев, предназначенных для работы конечных пользователей. При возникновении проблем пользователи могут пересыпать вам эту дополнительную информацию, тем самым значительно облегчая вашу работу по отысканию ошибок. Та же опция доступна и для команды `errort`.

Проверки во время работы

Так как функционирование ваших сценариев может зависеть от большого количества иных сценариев и команд, то перед вызовом процедуры или команды может возникнуть потребность в проверке ее существования. Чтобы выяснить, доступна ли та или иная процедура или команда, создана команда `exists`. Получив от нее ложное значение, вы можете предпринять соответствующие действия.

```
source simulation.mel
if( ! `exists startSimulation` )
    error "Please reinstall the simulation software";
    // "Пожалуйста, установите программу имитации заново"
```

Определение типов

Если вы не уверены в том, какой тип имеет данная переменная, процедура или другой оператор языка **MEL**, воспользуйтесь командой `whatIs`. В этом примере команда `whatIs` служит для определения типа переменной `$a`.

```
$a = 23;
whatIs "$a";
// Result: int variable //
// Результат: переменная int //
```

Следующий пример показывает, что идентифицировать можно и массивы переменных.

```
$objs = `ls`;
whatIs "$objs";
// Result: string[] variable //
// Результат: переменная string[] //
```

Команду `whatIs` можно использовать и для определения того, ведет ли вызов функции к выполнению встроенной команды.

```
whatIs sphere; // Result command (Результат: команда)
```

Кроме того, ее можно применять для выяснения того, является ли оператор процедурой, а также где расположен исходный код этой процедуры. Если вы выполните в окне **Script Editor** такие команды:

```
proc printName() { print "Bill" };
whatIs printName;
```

то получите следующий результат:

```
// Result: Mel procedure entered interactively. //
// Результат: процедура Mel, введенная при диалоге с пользователем. //
```

Если в сценарии `learning.mel` вы описали глобальную процедуру `printResults`:

```
whatIs printResults;
```

то получите такой результат:

```
// Result: Mel procedure found in: D:/Documents and Settings/davidgould/
  My Documents/maya/4.0/scripts/learning.mel //
// Результат: процедура Mel найдена в: D:/Documents and
  Settings/davidgould/
```

`My Documents/maya/4.0/scripts/learning.mel //`

`whatIs` можно использовать, чтобы выяснить, указывает ли оператор на файл сценария. Если это так, на экран будет выведен полный путь к сценарию.

```
whatIs learning;
```

```
// Result: Script found in: D:/Documents and Settings/davidgould/
```

`My Documents/maya/4.0/scripts/learning.mel //`

```
// Результат: Сценарий найден в: D:/Documents and Settings/davidgould/
```

`My Documents/maya/4.0/scripts/learning.mel //`

3.3.6. Поддержка версий

Если ваши сценарии должны выполняться в различных версиях Maya, вам будет приятно услышать о том, что большинство команд MEL работают в них без изменений. Если же вы хотите воспользоваться возможностями, которые не поддерживаются в той или иной версии, возможно, вы сможете предложить альтернативное решение.

Чтобы определить, какая версия Maya сейчас работает, используйте команду `about`.

```
about -version;  
// Result: 4.0 //
```

Следующий пример показывает, как сценарий, поддерживающий только Maya 4.0, может прекратить свое выполнение при запуске в более старой версии Maya.

```
float $ver = float(`about -version`);  
if( $ver < 4.0 )  
    error "Only supports Maya 4.0 or later";  
    // "Поддерживает только Maya 4.0 и старше"  
else  
    print "Supports this version of Maya";  
    // "Поддерживает эту версию Maya"
```

Команда `about` может использоваться и для определения текущей операционной системы.

```
about -operatingSystem;  
// Result: nt //
```

Если вам нужно загружать сценарии, зависящие от операционной системы, вы можете сделать это, запросив сначала команду `about`.

```
if( `about -operatingSystem` == "nt" )  
    source "c:\\myScripts\\start.mel";  
else  
    source "~/myScripts/startX.mel";
```

3.3.7. Размещение

Если вы единственный, кто будет пользоваться созданными вами сценариями, оставьте их в своем локальном каталоге `scripts`, и этого будет вполне достаточно. Если же **вам** нужно сделать сценарии доступными большому числу пользователей, вы должны выбрать схему их размещения. Следующие правила описывают один из способов размещения сценариев с целью доступа к ним широкой аудитории. В зависимости от обстоятельств, вам может понадобиться и другой **метод**.

1. Если работа ведется в производственной среде, где каждый пользователь может обращаться к центральному серверу, задача размещения существенно упрощается. Просто создайте на сервере совместно используемый каталог, например:

```
\server\mayaScripts
```

В этом примере используется формат, соответствующий универсальному коду имен **UNC** (Universal Naming Code). Вы же будете пользоваться тем форматом пути, которого требует ваша сетевая система.

Если у вас нет центрального сервера, а вместо этого каждый пользователь имеет свою собственную локальную **машину**, создайте на каждой машине каталог, например:

```
c:\mayaScripts
```

Вне зависимости от того, находится ли этот каталог *на* сервере или локальном жестком диске, будем обозначать его как **каталог_сценариев**.

1. Скопируйте готовые файлы сценариев MEL в **каталог_сценариев**
3. На каждой из машин пользователей установите переменную окружения с именем **MAYA_SCRIPTS_PATH** и включите в нее путь к каталогу, где расположены сценарии.

```
set MAYA_SCRIPTS_PATH=$MAYA_SCRIPTS_PATH;katalog_сценариев
```

Также вы можете обновить пользовательский файл `maya.env`, чтобы он содержал установку переменной окружения.

```
MAYA_SCRIPTS_PATH=$MAYA_SCRIPTS_PATH;katalog_сценариев
```

Можно, хотя и не рекомендуется, сохранять сценарии непосредственно в следующем каталоге:

Mac OS X:

```
/Applications/Maya x.x/Maya.app/Contents/scripts/
```

Все другие:

`maya_install\scripts\`

Лучше всего хранить свои сценарии отдельно от стандартных сценариев Maya. Это предотвратит случайное переопределение одного из стандартных сценариев Maya вашим собственным. К тому же такой подход помогает упростить обновление,

Обновление

Если в тот момент, когда вы размещаете свои сценарии, пользователи уже работают с Maya, им не удастся получить самую последнюю версию автоматически. Для этого они должны покинуть среду и запустить ее вновь. В ином случае сценарии достаточно просто считаны в память. Так или иначе, это гарантирует использование самых последних версий.

3.4. Объекты

Чаще всего MEL служит для создания и редактирования объектов. В этом разделе будет описан широкий спектр команд, предназначенных для работы с ними. Как упоминалось ранее, все объекты Maya - это, на самом деле, узлы графа **Dependency Graph**, поэтому в процесс создания и использования объектов в действительности входит создание и использование узлов. К счастью, чтобы пользоваться объектом или узлом, вы не обязаны знать всех деталей его реализации. Существует огромное количество команд MEL, которые разработаны специально для обращения к объектам и не требуют знания особенностей их функционирования. Некоторые команды к тому же могут работать с множеством различных объектов.

3.4.1. ОСНОВЫ

Рассмотрим основы создания объектов и манипулирования ими при помощи MEL.

1. Откройте сцену **Primitives.ma**.

Сцена состоит из трех NURBS-примитивов: конуса, сферы и цилиндра. Получение полного списка объектов в составе сцены - очень распространенная операция.

2. Откройте редактор Script **Editor**.

3. Выполните следующую команду:

```
ls;
```

На экран будет выведен полный перечень объектов сцены.

```
// Result: time1 renderPartition renderGlobalsList1 defaultLight ...
```

Для получения списка, содержащего только формы-поверхности, выполните следующую команду:

```
ls -type surfaceShape
```

Результат таков:

```
// Result: nurbsConeShape1 nurbsCylinderShape1 nurbsSphereShape1 //
```

Нередко вам будет нужен список текущих выделенных объектов.

4. Выделите объект nurbsCone1.

5. Выполните следующую команду:

```
ls -selection;
```

Результат имеет вид:

```
// Result: nurbsCone1 //
```

Чтобы удалить объект, воспользуйтесь командой delete.

6. Выполните следующую команду:

```
deletenurbsCone1;
```

Другой способ удаления текущего выделенного объекта - просто применить команду delete без аргументов. Для переименования объекта служит команда rename.

7. Выполните следующую команду:

```
rename nurbsSphere1 ball;
```

Объект nurbsSphere1 переименован в ball. Чтобы определить, существует ли объект с данным именем, воспользуйтесь командой objExists.

8. Выполните следующую команду:

```
if( `objExists ball` )
    print( "ball does exist"); // "Объект ball действительно
существует"
```

Каждый объект имеет конкретный тип. Для выяснения типа объекта создана команда objectType.

9. Выполните следующую команду:

```
objectType ball;
```

Объект ball имеет тип transform. Это значит, что он представляет собой transform-узел.

Сценарий *listAll*

Этот сценарий иллюстрирует прием организации цикла по всем объектам сцены и выводит на экран имена, а также типы объектов.

```
proc listAll()
{
print( "\nNodes..." ); // "Узлы..."
string $nodes[] = `ls`;
for( $node in $nodes )
{
    string $nodeType = `objectType $node`;
    print ("\\nNode: " + $node + " (" + $nodeType + ")");
}
}
```

listAHO;

Сначала описана процедура *listAll()*. Она не принимает никаких параметров и не имеет возвращаемого значения.

```
proc listAll()
{
```

Вслед за выводом с новой строки текста *Nodes...* следующим шагом является получение полного списка узлов сцены. Результат команды *ls* сохраняется в массиве *\$nodes*. Обратите внимание на необходимость заключения команды *ls* в одинарные обратные кавычки *(`)* для получения возвращаемого значения.

```
string $nodes[] = `ls`;
```

Далее организуется цикл по всем элементам списка.

```
for( $node in $nodes )
{
```

Тип каждого узла определяется с использованием команды `objectType`. Результат сохраняется в переменной `$nodeType`.

```
string $nodeType = `objectType $node`;
```

Наконец, информация об узле выводится на экран. Она представляет собой результат сцепления ряда строк-литералов, а также ранее сохраненных переменных строкового типа. Перед вызовом команды `print` очень важно поместить все выражение в круглые скобки.

```
print ("\\nNode: " + $node + " C" + $nodeType + ")");
```

Если открыта сцена **Primitives.ma**, то во время своего запуска сценарий выводит следующую информацию:

```
Nodes...
Node; time1 (time)
Node: renderPartition (partition)
Node: renderGlobalsList1 (renderGlobalsList)
Node: defaultLightList1 (defaultLightList)
Node: defaultShaderList1 (defaultShaderList)
Node: postProcessList1 (postProcessList)
Node: defaultRenderUtilityList1 (defaultRenderUtilityList)
Node: lightList1 (lightList)
Node: defaultTextureList1 (defaultTextureList)
Node: lambert1 (lambert)
... продолжение
```

3.4.2. Иерархии

Каждая форма-поверхность имеет родительский узел **transform**. Поэтому принадлежащий ей узел **shape** считается его потомком. Тот может быть предком другого узла **transform**. В действительности вы можете создать произвольную иерархию узлов с любым количеством потомков. Каждый из этих потомков, в свою очередь, может иметь собственные дочерние узлы.

Язык MEL содержит несколько команд создания иерархии объектов и перемещения по ней.

1. Откройте сцену **Primitives.ma.**

2. В редакторе **Script Editor выполните следующую команду:**

```
group -name topTransform nurbsSphere1;
```

Команда **group** служит для создания нового узла **transform**. В данном случае он носит название **topTransform** и имеет потомка-узел **nurbsSphere1**. Чтобы сделать данный узел потомком другого узла, можно использовать команду **parent**.

```
parent nurbsCone1 topTransform;
```

Теперь узел **nurbsCone1** является потомком узла **topTransform**. Иерархия выглядит так, как показано на рис. 3.3.

Перемещение, масштабирование или поворот узла **topTransform** влияет на всех его потомков. Выполните следующую команду:

```
move -relative 0 3 0 topTransform;
```

И конус, и сфера смещаются вверх. Чтобы преобразования родителя не влияли на его потомков, воспользуйтесь командой **inheritTransform** для отмены действия этой зависимости.

```
inheritTransform -off nurbsCone1;
```

Конус возвращается в исходное положение. С этого момента перемещения узла **topTransform** не оказывают воздействия на конус, но влияют на сферу. Чтобы увидеть список потомков узла, обратитесь к следующей команде:

```
listRelatives topTransform;
```

Результат:

```
// Result: nurbsSphere1 nurbsCone1 //
```

Для получения полного списка всех дочерних узлов, включая косвенных потомков, используйте флаг **-allDescendants**. Для получения списка всех форм-потомков данного узла, используйте флаг **-shapes**, как показано ниже:

```
listRelatives -shapes nurbsSphere1;
```

Результат:

```
// Result: nurbsSphereShape1 //
```

Список предков узла можно получить так:

```
listRelatives -parent nurbsSphereShape1;
```

Результат:

```
// Result: nurbsSphere1 //
```

Если объект имеет несколько экземпляров, всех его возможных родителей можно получить, пользуясь флагом **-allParents**. Для изменения порядка *сестринских узлов* служит команда **reorder**. Следующая команда позволяет перенести узел **nurbsCone1** в начало списка потомков:

```
reorder -front nurbsCone1;
```

Переупорядочение сестринских узлов никак не влияет на отношения «родитель - потомок», а просто изменяет порядок их следования в списке дочерних узлов. Разрыв соединения между родителем и потомком эквивалентен объявлению последнего потомком узла **world**. **World** - это предок всех остальных узлов, расположенный на самом верхнем уровне иерархии. Для разрыва соединения с преобразованием **nurbsCone1** используйте команду
`parent -world nurbsCone1;`

Также можно применять команду **ungroup**. Эквивалентный вызов имеет вид:
`ungroup -world nurbsCone1.`

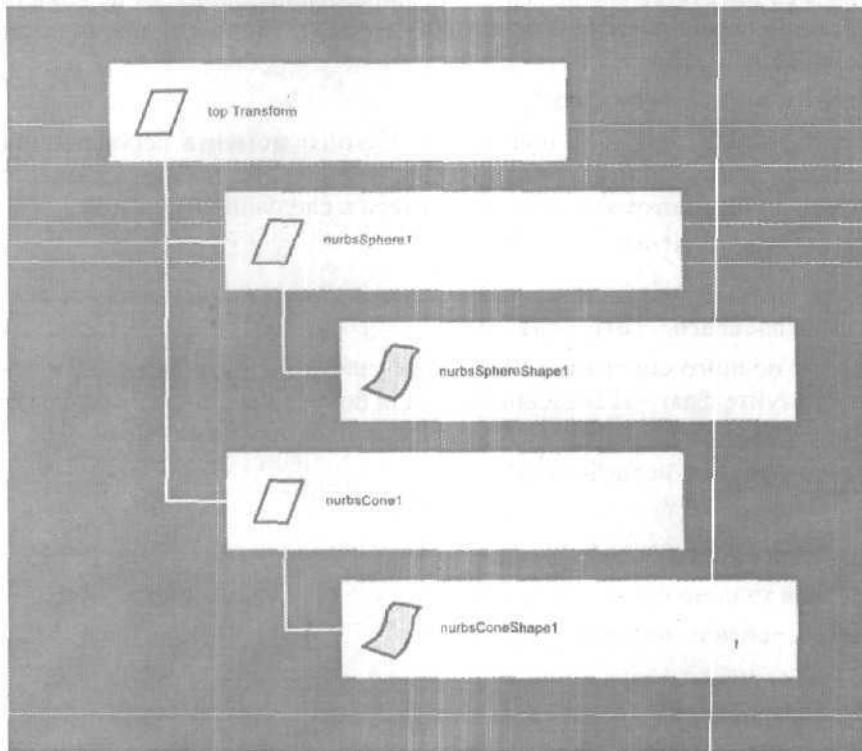


Рис. 3.3. Иерархия узлов

3.4.3. Преобразования

Практически все объекты в составе сцены (камеры, источники света, поверхности и т. д.) хранятся как узлы ОАГ. Большинство этих узлов состоят из узла **shape**, а также узла **transform**, являющегося его родителем. Узел **transform** определяет положение, ориентацию и масштаб своих дочерних узлов.

1. Откройте сцену **Primitives.ma**.

2. В редакторе **Script Editor** выполните следующую команду:

```
move -relative 3 0 0 nurbsSphere1;
```

Хотя может показаться, что это привело к перемещению узла **nurbsSphere1**, на самом деле произошло перемещение **nurbsSphereShape1**. Узел **nurbsSphere1** всего лишь хранит ту информацию, которая необходима для перемещения его потомков. Эта информация содержится в матрице преобразования. Затем матрица служит для переноса потомков на новое место. Стало быть, в данном случае преобразование поверхности узла **shape (nurbsSphereShape1)** выполняется при участии матрицы, которая реализует перенос узла на новое место вдоль оси **x**.

3. Выполните следующую команду:

```
scale -absolute 1 0.5 1 nurbsSphere1;
```

На сей раз сфера прижимается к оси **y**. Вращения объекта можно добиться с использованием команды **rotate**.

```
rotate -relative 45deg 0 0 nurbsSphere1;
```

Трансляцию, вращение и масштабирование объекта можно одновременно осуществить при помощи команды **xform**. Выполните следующий оператор:

```
xform -relative -translation 0 -2 -3 -scale 0.5 1 1  
-rotation C 0 10deg nurbsSphere1;
```

Кроме того, команду **xform** можно использовать для реализации сдвига.

```
xform -shear 1 0 0
```

Матрицы преобразования

Когда вы работаете с узлом преобразования, изменяя отдельные его составляющие (выполняя масштабирование, трансляцию и вращение), окончательным результатом становится матрица преобразования. Она позволяет упаковать все эти операции в единую структуру. Матрицы преобразования имеют четыре

строки и четыре столбца: 4×4 . Чтобы осуществить преобразование точки, ее умножают на матрицу преобразования справа.

точка' = точка X матрица

Текущая матрица преобразования хранится в узле **transform**. По запросу Maya возвращает различные типы матриц в зависимости от того, в каком *пространстве* вы хотите работать.

Пространства

Вводя описание точки, вы автоматически соотносите ее с началом координат (O, O, O). О такой точке говорят, что она расположена в *локальном пространстве*. Локальным называется пространство, в котором точка существует изначально и в котором она не подвергается никаким преобразованиям. Точки, описанные в таком пространстве, имеет любая форма.

Каждая форма обладает узлом преобразования, задача которого - перенести ее в *мировое пространство*. Мировым именуют «итоговое» пространство, где производится рисование форм. Если форма располагает только одним узлом преобразования, он неявным образом определяет преобразование локального пространства в мировое. **Возьмем** для примера форму пальца **fingerShape**, обладающую единственным родительским **transform-узлом** **finger**.

finger | fingerShape

Для преобразования точки формы **fingerShape** в мировые координаты просто умножьте точку на матрицу преобразования узла.

мировая точка = точка x fingerTransform

Если узел преобразования формы обладает другими родительскими узлами категории **transform**, то для расчета окончательного преобразования локального пространства в мировое необходимо произвести *сцепление* всех преобразований родителей. Положим, что узел преобразования **finger** обладает другими родителями: **hand** (кисть) и **arm** (рука).

arm | hand j finger j fingerShape

Теперь для преобразования точки из *локального* пространства в мировое нужно осуществить сцепление всех преобразований родителей.

**finalTransform = fingerTransform X handTransform x
armTransform**

мировая точка = точка x finalTransform

Обратите внимание на то, что матрицы умножаются справа. Это означает, что любая новая матрица приписывается к существующей матрице справа от знака умножения: **существующая матрица X новая матрица.**

1. Откройте сцену **Primitives.ma**.
2. В редакторе **Script Editor** выполните следующую команду:

```
xform -query -matrix nurbsSphere1;  
// Result: 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 //
```

Получена матрица преобразования узла **nurbsSphere1**. Теперь попытайтесь ее сохранить. Выполните следующую команду:

```
matrix $mtx[4][4] = `xform -query -matrix nurbsSphere1`;
```

К сожалению, это приведет к возникновению ошибки.

```
// Error: line 1: Cannot convert data of type float[] to type matrix. //  
// Ошибка: строка 1: Нельзя преобразовать данные типа float[] в тип  
matrix. //
```

Проблема состоит в том, что команда **xform** возвращает не матрицу [4][4], а нераздельный массив из 16 вещественных чисел. По этой причине правильный способ получения матрицы таков:

```
float $mat[] = `xform -query -matrix nurbsSphere1`;
```

Возвращаемый массив является результатом «плоской упаковки» двухмерного массива в одномерный. Матрица из четырех строк и четырех столбцов выглядит так:

```
[ a b c d ]  
[ e f g h ]  
[ C i J K l ]  
[ m n o p ]
```

Преобразование в одномерный массив дает следующий результат:

```
[ a b c d e f g h i j k l m n o p ]
```

Все строки матрицы просто записываются друг за другом. Чтобы обратиться к одномерному массиву через индексы строки и столбца, воспользуйтесь данной формулой преобразования:

индекс в массиве = индекс строки \times число столбцов + индекс столбца;

Несмотря на то что вы можете запросить текущую матрицу преобразования, вам не удастся задать ее, воспользовавшись матрицей, которая вычислена за-

ранее. Матрицу преобразования можно изменить лишь косвенным путем, изменяя значения атрибутов **scale**, **rotate** и **translate** узла **transform**.

По умолчанию команда **xform** возвращает текущее преобразование, хранящееся в узле **transform**. Для получения матрицы преобразования локального пространства в мировое выполните следующую команду.

```
float $mat[] = `xform -query -worldSpace -matrix nurbsSphere1`;
```

Сценарий objToWorld

Приведенный ниже сценарий можно использовать для перевода точки из локального пространства формы в ее окончательное положение в мировом пространстве.

```
proc float[] transformPoint( float $pt[], float $mtx[] )
{
    float $res[] = { 0.0, 0.0, 0.0 };

    if( size($pt) != 3 && size($mtx) != 16 )
        <
        warning "transformPoint proc: pt must have three elements and
                matrix must have 16 elements";
    // "процедура transformPoint: точка должна содержать три элемента,
    // матрица должна содержать 16 элементов";
    return $res;
}

$res[0] = $pt[0] * $mtx[0] + $pt[1] * $mtx[4] +
         $pt[2] * $mtx[8] + $mtx[12];
$res[1] = $pt[0] * $mtx[1] + $pt[1] * $mtx[5] +
         $pt[2] * $mtx[9] + $mtx[13];
$res[2] = $pt[0] * $mtx[2] + $pt[1] * $mtx[6] +
         $pt[2] * $mtx[10] + $mtx[14];
return $res;
}

proc float[] objToWorld( float $pt[], string $transformNode )
{
```

```
float $mtx[16] = `xform -query -worldSpace -matrix $transformNode`;
float $res[] = transformPoint( $pt, $mtx );
return $res;
}
```

К сожалению, язык MEL не содержит средств умножения матриц для работы с векторами и матрицами преобразования. В сочетании с проблемой выдачи последних как нераздельных массивов, это означает, что вы должны проделать эти операции самостоятельно.

Процедура `transformPoint` предназначена для преобразования точки с участием матрицы преобразования. Точка (`$pt`) представлена как массив из 3 вещественных чисел. Матрица (`$mtx`) дана как массив из 16 вещественных чисел. Процедура возвращает итоговое положение преобразованной точки как массив из 3 чисел вещественного типа.

```
proc float[] transformPoint( float $pt[], float $mtx[] )
```

```
{
```

Важно отметить положенное в основу процедуры предположение о том, что определяемое матрицей преобразование является *аффинным*. Аффинное преобразование состоит лишь из операций масштабирования, вращения и трансляции. Масштабирование может быть неравномерным. Коль скоро матрица преобразования является аффинной, ее последний столбец, по предположению, равен [0 0 0 1].

Описывается и инициализируется точка, которая станет результатом преобразования:

```
float $res[] = { 0.0, 0.0, 0.0 };
```

Далее проверяется длина массивов `$pt` и `$mtx`. Если она не соответствует правильному значению, выдается предупреждение, и процедура возвращает результат.

```
if( size($pt) != 3 && size($mtx) != 16 )
{
    warning "transformPoint proc: pt must have three elements and
            matrix must have 16 elements";
    return $res;
}
```

Затем путем умножения матриц точка подвергается преобразованию. Наконец, полученная в результате точка возвращается.

```
$res[0] = $pt[0] * $mtx[0] + $pt[1] * $mtx[4] +
          $pt[2] * $mtx[8] + $mtx[12];
$res[1] = $pt[0] * $mtx[1] + $pt[1] * $mtx[5] +
          $pt[2] * $mtx[9] + $mtx[13];
$res[2] = $pt[0] * $mtx[2] + $pt[1] * $mtx[6] +
          $pt[2] * $mtx[10] + $mtx[14];
return $res;
```

Процедура `objToWorld` принимает точку, заданную в локальном пространстве формы, и преобразует ее в мировое пространство. Она, принимает точку (\$pt) в виде массива из трех вещественных чисел, а также название узла `transform` (\$transformNode). Результат - точка в мировом пространстве - возвращается как массив из 3 элементов вещественного типа.

```
proc float[] objToWorld( float $pt[], string $transformNode )
{
```

Сначала запрашивается матрица преобразования локального пространства в мировое.

```
float $mtx[16] = `xform -query -worldSpace -matrix $transformNode`;
```

Далее при помощи этой матрицы выполняется преобразование точки, затем следует ее возврат.

```
float $res[] = transformPoint( $pt, $mtx );
return $res;
}
```

Приведем пример использования этой процедуры,

```
float $pt[] = { 1.0, 2.0, 3.0 };
$pt = objToWorld( $pt, "nurbsSphere1" );
print $pt;
```

Сценарий spaceToSpace

Далее рассматривается более сложный сценарий, способный преобразовать точку из одного пространства в другое.

```
proc float[] transformPoint( float $pt[], float $mtx[] )
```

```
... как и раньше  
}  
  
proc int getInstanceIndex( string $nodePath )  
{  
    string $paths[] = `ls -allPaths $nodePath`;  
    int $i;  
    for( $i=0; $i < size($paths); $i++ )  
    {  
        if( $paths[$i] == $nodePath )  
            return $i;  
    }  
    return -1;  
}  
  
proc float[] spaceToSpace( float $pt[],  
                           string $fromSpace, string $fromNode,  
                           string $toSpace, string $toNode )  
{  
    float $res[] = $pt;  
    float $mtx[];  
  
    // Преобразовать точку в мировое пространство  
    if( $fromSpace == "local")  
    {  
        $mtx = `xform -query -worldSpace -matrix $fromNode`;  
        $res = transformPoint( $res, $mtx );  
    }  
  
    // Преобразовать точку в целевое пространство  
    if( $toSpace == "local")  
    {  
        int $inst = getInstanceIndex( $toNode );  
        string $attr = $toNode + ".worldInverseMatrix[" + $inst + "]";
```

```

$mtx = `getAttr $attr`;
print "\nInverse: "//"Обратная матрица:";
print $mtx;
$res = transformPoint( $res, $mtx );
}

return $res;
}

```

Процедура `getInstanceIndex` возвращает индекс экземпляра данного узла. Полный путь к объекту по ОАГ задан аргументом `$nodePath`.

```

proc int getInstanceIndex( string $nodePath )
{

```

Определяется полный перечень всех возможных путей к данному узлу.

```
string $paths[] = `ls -allPaths $nodePath`;
```

По каждому из этих путей организуется цикл.

```

int $i;
for( $i=0; $i < size($paths); $i++ )
{

```

Если путь совпадает с заданным, индекс пути возвращается. Он является индексом экземпляра.

```

if( $paths[$i] == $nodePath )
    return $i;
}

```

Если путь к узлу не найден, возвращается -1 как признак неудачного завершения функции.

```

return -1;
}

```

Процедура `spaceToSpace` принимает преобразуемую точку `$pt` как массив из трех вещественных чисел. Возможные пространства – локальное и глобальное. Пространство, в котором определена точка, указывает переменная `$fromSpace`. Узел, в котором определена точка, указывает переменная `$fromNode`. Переменные `$toSpace` и `$toNode` содержат, соответственно, информацию о целевом пространстве и целевом узле.

```
proc float[] spaceToSpace( float $pt[],  
                           string $fromSpace, string $fromNode,  
                           string $toSpace, string $toNode )  
{
```

Результирующая точка `$res` инициализируется значением исходной.

```
float $res[] = $pt;
```

Чтобы перевести точку в единое унифицированное пространство, выполняется ее преобразование в мировые координаты. Если исходное пространство `$fromSpace` уже является мировым, это преобразование выполнять не нужно; в противном случае точка должна быть описана в локальном пространстве и потому она требует преобразования в мировое.

```
if( $fromSpace == "local" )
```

```
{
```

Для узла `$fromNode` запрашивается матрица преобразования локального пространства в мировое.

```
$mtx = `xform -query -worldSpace -matrix $fromNode`;
```

Затем точка преобразуется.

```
$res = transformPoint( $res, $mtx );
```

```
}
```

Поскольку теперь точка имеет мировые координаты, вы можете перенести ее в целевое пространство. Если аргумент `$toSpace` задает мировое пространство, работа завершена. Если пространство является локальным, необходимо преобразовать точку из мировой системы координат в локальное пространство целевого узла.

```
if( $toSpace == "local" )
```

```
{
```

Найдем индекс экземпляра целевого узла. Его важно использовать потому, что пути к тиражируемому узлу могут быть различными. Индекс пути к целевому узлу нужен, чтобы гарантировать получение правильной матрицы преобразования.

```
int $inst = getInstanceIndex( $toNode );
```

Затем для целевого узла запрашивается матрица преобразования мирового пространства в локальное. Эту матрицу представляет атрибут `worldInverseMatrix`.

Он является массивом матриц, каждая из которых соответствует возможному пути к экземпляру. Индекс экземпляра служит для отыскания нужной матрицы.

```
string $attr = $toNode + ".worldInverseMatrix[" + $inst + "];  
$mtx = `getAttr $attr`;
```

Далее с помощью этой матрицы точка преобразуется и погружается в локальное пространство целевого узла.

```
$res = transformPoint( $res, $mtx );  
}
```

После этого точки, полученная в результате преобразования, возвращается.

```
return $res;  
>
```

Процедура spaceToSpace может использоваться следующим образом:

```
float $pt[3] = { 1.0, 2.0, 3.0 };  
$pt = spaceToSpace( $pt, "local", "nurbsSphere1", "local", "nurbsConel" );  
print ("\\n From nurbs to cone: " + $pt[0] + ", " +  
      $pt[1] + ", " + $pt[2]);
```

Чтобы преобразовать точку в локальных координатах в мировые, воспользуйтесь командой

```
$pt = spaceToSpace( $pt, "local", "nurbsSphere1", "world", "" );
```

Чтобы преобразовать точку из мировых координат в локальные, воспользуйтесь командой

```
$pt = spaceToSpace( $pt, "world", "", "local", "nurbsConel" );
```

3.4.4. Атрибуты

Так как вся информация о сцене хранится в атрибутах отдельных узлов, очень часто бывает необходимо обращаться к атрибутам и редактировать их. Для получения и установки различных атрибутов объекта MEL предоставляет, соответственно, команды getAttr и setAttr.

1. Откройте сцену Primitives.ma.
2. Откройте редактор Script Editor.
3. Выполните следующие команды:

```
$rad = `getAttr makeNurbCone1.radius`;  
print ("\\nRadius: " + $rad); // "Радиус: "
```

Результат:

Radius: 1

4. Выполните следующие команды:

```
setAttr makeNurbCone1.radius 2.5  
$rad = `getAttr makeNurbCone1.radius`;  
print ("\\nRadius: " + $rad);
```

Конус становится больше, а результат принимает вид:

Radius: 2.5

Заметьте, что вызов `setAttr` не был заключен в обратные кавычки (`). Коль скоро вас не интересует значение возврата команды `setAttr`, ее можно вызывать напрямую.

5. Выполните следующие команды:

```
float $s[] = 'getAttr nurbsSphere1.scale';  
print $s;
```

Поскольку атрибут `scale` имеет тип `double3`, для его хранения потребовался массив вещественных переменных. Для установки таких сложных значений атрибутов их нужно разбивать на отдельные вызовы `setAttr`.

6. Выполните следующие команды:

```
vector $sc = << 1.5, 2.3, 1.4 >>;  
setAttr nurbsSphere1.scale ($sc.x) ($sc.y) ($sc.z);
```

Масштаб сферы записывается в компоненты переменной `$s`. Интуитивно более понятным может показаться простой вызов `setAttr`.

```
setAttr nurbsSphere1.scale $sc;  
// Результат: ошибка при синтаксическом разборе аргументов
```

К сожалению, эта команда вызывает ошибку. Компоненты масштабного вектора, как и любого другого сложного атрибута, должны задаваться по отдельности.

В предыдущем примере вам было известно название объекта, радиус которого вы хотели определить. А что если вы намерены прочитать или установить значение атрибута текущего выделенного объекта?

1. Выделите все примитивы.

Попытайтесь наполовину уменьшить высоту всех объектов.

2. Выполните следующие команды:

```
$objs = `ls -selection`;
for( $obj in $objs )
    setAttr ($obj + ".scale") 1 0.5 1;
Высота выделенных примитивов уменьшится.
```

Динамические атрибуты

Каждый узел обладает предопределенным набором атрибутов. К примеру, узел **transform** наделен множеством разнообразных атрибутов, предназначенных для определения параметров трансляции, масштабирования и вращения узла преобразования. К узлам можно добавлять дополнительные атрибуты. Они могут служить для присоединения данных пользователя к тому или иному узлу. Различные объекты на игровой сцене могут, например, классифицироваться как оружие (weapon), снадобье (potion) или препятствие (obstacle). С каждым объектом связано некоторое количество очков (points), приписываемых к общему результату игрока, а также информация о возможности обновления объекта (renewable).

1. Откройте сцену **Primitives.ma**.
2. Выделите сферический объект с именем **nurbsSphere1**.
3. Выполните следующие команды:

```
$objs = `ls -selection`;
for( $obj in $objs )
{
    addAttr -longName "points" -attributeType long $obj;
    addAttr -longName "renewable" -attributeType bool $obj;
    addAttr -longName "category" -attributeType enum
        -enumName "Weapon:Potion:Obstacle" $obj;
}
```

Атрибуты будут добавлены к узлу **nurbsSphere1**.

4. Откройте редактор **Attribute Editor** (Редактор атрибутов).
5. Щелкните по вкладке **nurbsSphere1**.
6. Чтобы развернуть ее, щелкните по кнопке **Extra Attributes** (Дополнительные атрибуты).

Вы увидите все три атрибута, которые только что добавили.

Если атрибут существует, а вы попытаетесь добавить его снова, возникнет ошибка. Для ее предотвращения воспользуйтесь командой **attributeExists**, которая позволит определить, действительно ли узел уже содержит такой атрибут.

```
if( !attributeExists( "points", $obj ) )  
    addAttr ...;
```

Этой же командой можно пользоваться при работе с атрибутами узла, которые были созданы заранее.

Чтобы вывести динамические атрибуты в окне **Channel Box**, их надо сделать кадрируемыми. Это значит, что такой атрибут сможет участвовать в установке ключевых кадров. Для разрешения кадрирования атрибута **points** воспользуйтесь следующей командой:

```
setAttr -keyable true nurbsSphere1.points;
```

Заметьте, что порядок отображения атрибутов в окне **Channel Box** совпадает с порядком их добавления. Изменить порядок следования атрибутов позднее станет невозможно, поэтому будьте бдительны в том случае, если упорядочение имеет какое-то значение.

Для удаления динамического атрибута служит команда **deleteAttr**.

```
deleteAttr nurbsSphere1.points;
```

Кроме того, динамический атрибут можно переименовать.

```
renameAttr nurbsSphere1.points boost;
```

Удалять или переименовывать атрибуты, созданные заранее, нельзя. Эти команды работают лишь с динамическими атрибутами.

Информация об атрибутах

Каков исчерпывающий список атрибутов данного узла? Для выдачи полного перечня атрибутов предназначена команда **listAttr**.

```
listAttr nurbsSphere1;
```

Она возвращает массив строк (**string[]**) с именами атрибутов. Следующий пример показывает, как организовать цикл по отдельным именам с целью их вывода на экран.

```
string $attrs[] = `listAttr nurbsSphere1`;  
print ("\\nAttributes..."); // "Атрибуты..."  
for( $attr in $attrs )  
    print ("\\n" + $attr);
```

К тому же вы можете использовать фильтры для ограничения списка атрибутов конкретным типом. Для просмотра только динамических атрибутов воспользуйтесь следующей командой:

```
listAttr -userDefined nurbsSphere1;
```

Все атрибуты, допускающие кадрирование, можно просмотреть с помощью команды

```
listAttr -keyable nurbsSphere1;
```

Имеется полное справочное руководство по всем внутренним узлам Maya, где перечислены сами узлы и все их атрибуты. Для обращения к документации выберите в главном меню Maya пункт **Help | Nodes and Attribute Reference...** (**Помощь | Справочник по узлам и атрибутам...**).

Чтобы получить общую информацию об атрибуте, выполните команду `getAttr`.

```
getAttr -type nurbsSphere1.points; // Результат: long  
getAttr -keyable nurbsSphere1.points; // Результат: 0
```

Чтобы получить прочую информацию об атрибуте, выполните команду `attributeQuery`.

```
attributeQuery -node nurbsSphere1 -hidden points; // Результат: 0  
attributeQuery -node nurbsSphere1 -rangeExists points; // Результат: 0
```

3.5. Анимация

Язык MEL содержит немало команд для создания и редактирования элементов анимации. Многие из этих команд могут оказаться очень полезными и при автоматизации типичных анимационных задач. Хотя создание всех ключевых кадров в процессе рисования объекта обычно входит в задачу аниматора, создавать, редактировать и удалять ключевые кадры можно и при помощи MEL. Генерируя ключевые кадры программным путем, вы можете создавать сложные анимационные решения, которые были бы невозможны или попросту непрактичны при использовании более традиционных методов. Кроме того, вы можете пользоваться сценариями, работа которых состоит в том или ином изменении ряда ключевых кадров, анимированных вручную. MEL способен освободить аниматора от множества рутинных и утомительных задач, тем самым оставляя больше времени для более творческой работы.

3.5.1. Время

Любое обсуждение вопросов анимации должно начинаться с точного определения времени. В Maya это особенно важно потому, что вы можете задавать время в самых разных единицах измерения, включая кадры, полукадры, секунды, минуты и т. д. В среде Maya время хранится в секундах, хотя оно и может выводиться в любом удобном формате. Например, на экране время может быть показано в кадрах, однако в Maya для его хранения служат секунды.

Текущая единица измерения времени указывается в настройках параметров. Выберите пункт меню **Window | Settings/Preferences | Preferences...** (Окно Установка/Параметры Параметры...). Щелкните по категории **Settings** (Установка). Перед вами появится элемент **Working Units** (Рабочие единицы). По умолчанию единицей измерения времени является **Film[24fps]** (Фильм[24 кадр/с]). Выбор других единиц времени влечет за собой изменение масштаба текущей анимации, приводящее ее в соответствие с новыми единицами измерения. Если, к примеру, 12-й кадр является ключевым при измерении времени в единицах **Film[24fps]**, то при переходе к **Film[30fps]** ключевым станет 15-й кадр, что сохранит относительное положение ключевого кадра во времени. Воспроизведение 12-го кадра в единицах времени фильма соответствует времени 0.5 секунды. Согласно стандарту **NTSC**, ключевым должен стать 15-й кадр, гарантирующий сохранение положения ключа относительно момента, равного 0.5 секунды.

Чтобы узнать текущую единицу измерения времени, воспользуйтесь следующей командой:

```
currentUnit -query -time;
```

Для установки текущей единицы времени выполните команду

```
currentUnit -time "min"; // Установить минуту как единицу времени
```

По умолчанию Maya настраивает все ключевые кадры в соответствии с новыми единицами измерения ради сохранения относительного времени показа этих кадров. Для установки текущих единиц времени без автоматического изменения положения ключевых кадров используйте команду

```
currentUnit -time "min" -updateAnimation false;
```

Важно понимать тот факт, что все команды языка MEL учитывают в своей работе текущие единицы измерения времени. Если такой единицей является **Film[24fps]**, время указывается в кадрах. Если единицей измерения времени являются миллисекунды (**milliseconds**), время указывается в миллисекундах. Составляя сценарии на языке MEL, никогда не полагайтесь на то, что вам заранее известны рабочие единицы измерения времени. Установить текущее время позволяет команда **currentTime**. Она дает различные результаты при разных временных единицах.

```
currentTime 10;
```

Если время измеряется в единицах **Film[24fps]**, текущее время будет соответствовать 10-му кадру. Если время измеряется в миллисекундах, текущее время будет равно 10 миллисекундам. При необходимости установить абсолютное время, не зависящее от текущих единиц, вы можете присоединить к числовому значению ^{6*}

чению времени ту единицу, которая вам нужна. Следующая команда, например, всегда устанавливает текущее время, равное 2 секундам:

```
currentTime 2sec;
```

После числового значения времени вы можете записать любую из следующих единиц: hour (часы), min (минуты), sec (секунды), millisec (миллисекунды), game, film, pal, ntsc, show, palf, ntscf. Установить текущее время, равное $\frac{1}{4}$ часа, можно так:

```
currentTime 1.25hour;
```

Единицы, не соответствующие стандартным значениям времени, переведены в соответствующее число кадров в секунду (**fps**, frames per second) и представлены в табл. 3.6.

ТАБЛИЦА 3.6. ЕДИНИЦЫ ИЗМЕРЕНИЯ ВРЕМЕНИ

Единица измерения	Описание	Число кадров в секунду
game	Скорость анимации в играх	15
film	Обычный кинофильм	24
pal	Стандарт телевещания PAL (кадры)	25
ntsc	Стандарт телевещания NTSC (кадры)	30
show	Формат Show (кадры)	48
palf	Стандарт телевещания PAL (полукадры)	50 (2 × частота кадров)
ntscf	Стандарт телевещания NTSC (полукадры)	60 (2 × частота кадров)

Более точно, воспроизведение сигнала NTSC происходит с частотой 29,97 кадр/с, поэтому рендеринг с частотой 30 кадр/с потребует отбрасывания кадров итогового изображения или их редактирования. Осуществляя вывод в стандарте SECAM, пользуйтесь настройками PAL, коль скоро оба стандарта имеют одинаковую скорость воспроизведения.

Запрашивая содержимое конкретного **атрибута**, по умолчанию вы будете получать его значение в текущий момент времени.

```
currentTime 5;
```

```
getAttr sphere.translateX; // Получить атрибут в момент времени 5
```

Команда **getAttr** может служить и для запроса значения в момент времени, отличный от текущего.

```
currentTime 5;
```

```
getAttr -time 10 sphere.translateX; // Получить атрибут в момент времени 10
```

Подобная возможность может быть особенно полезна при получении значений атрибутов различных объектов в разные моменты времени. При этом значение `currentTime` заранее устанавливать не нужно. К сожалению, это позволяют не все команды. Команда `setAttr`, например, дает возможность устанавливать значение атрибута только в текущее время. Так как смена текущего времени ведет к обновлению всей сцены, это может оказаться накладным, если вы хотели лишь изменить время, выполнить ряд команд и вернуться обратно. К счастью, текущее время можно менять и без обновления сцены. В следующем примере выполнена смена текущего времени, произведена установка значения и осуществлен возврат к прежнему моменту. Для предотвращения обновления сцены просто используйте флаг `-update false`.

```
float $cTime = currentTime -query; // Определить текущее время  
currentTime -update false 10; // Перейти к 10-му кадру, но не обновлять сцену  
setAttr sphere.translateX 23.4; // Установить значение атрибута в 10-м кадре  
currentTime -update false $cTime; // Восстановить прежнее время
```

3.5.2. Воспроизведение

В Maya имеется немало команд для непосредственного управления показом текущей анимации. Они описывают процесс воспроизведения в окнах просмотра Maya, но не регулируют скорость итоговой анимации. Поскольку Maya сначала рассчитывает каждый кадр анимации, а лишь затем выводит его на экран, скоростью итоговой анимации считают скорость интерактивного воспроизведения в реальном масштабе времени. Скорость, с которой анимация отображается в диалоговых окнах просмотра, зависит от множества факторов, в том числе скорости вашей машины, сложности сцены и возможностей используемой графической карты.

1. Откройте сцену SimpleAnimatedSphere.ma.
2. Щелкните по кнопке Play.
Сфера станет перемещаться по экрану слева направо.
3. В редакторе Script Editor выполните следующую команду:

```
play;
```

Maya приступит к показу анимации, для чего и служит команда `play`. Вы можете управлять направлением и иными параметрами воспроизведения. Не останавливая показа, выполните следующие действия.

4. В редакторе **Script Editor** выполните следующую команду:

```
play -forward false;
```

Теперь анимация будет воспроизводиться в противоположном направлении.

5. В редакторе **Script Editor** выполните следующую команду:

```
play -query -state;
```

```
// Result: 1 //
```

Состояние команды play равно 1, если в данный момент происходит показ анимации, и 0 в ином случае.

6. В редакторе **Script Editor** выполните следующую команду:

```
play -state off;
```

Показ анимации будет остановлен.

В числе параметров текущего воспроизведения описываются диапазон и скорость показа. Пользуясь командой `playbackOptions`, вы можете их изменять. При этом важно понимать различие между диапазоном анимации и диапазоном воспроизведения. Диапазон анимации совпадает с общей продолжительностью последней. Диапазон воспроизведения может быть равен всему диапазону анимации или составлять малую его часть. Работая над конкретным фрагментом, зачастую лучше всего сужать диапазон воспроизведения до этих пределов, не касаясь диапазона анимации в целом.

1. В редакторе **Script Editor** выполните следующую команду:

```
playbackOptions -minTime 12 -maxTime 20;
```

Заданный диапазон воспроизведения теперь имеет пределы 12, 20. Диапазон анимации остается равным 1, 48. Для установки диапазона анимации воспользуйтесь флагами `-animationStartTime` и `-animationEndTime`.

2. В редакторе **Script Editor** выполните следующие команды:

```
undo;
```

```
playbackOptions -loop "oscillate";
```

```
play;
```

Теперь анимация будет поочередно воспроизводиться в прямом и обратном направлении.

3. В редакторе **Script Editor** выполните следующую команду:

```
playbackOptions -query -playbackSpeed;
```

```
// Result: 1 //
```

Скорость воспроизведения анимации равна 100%. Для ее показа со скоростью вдвое меньше нормальной используйте следующую команду.

```
playbackOptions -playbackSpeed 0.5;  
// Result: 0.5 //
```

Важно понимать то, что скорость показа - это всего лишь ориентир. Чаще итоговый вариант воспроизведения не соответствует текущим настройкам единиц времени. Чтобы увидеть фактическую скорость вывода анимации, вы должны показать на экране частоту кадров, обратившись к элементу **Heads Up Display** (Большой экран). Для этого включите опцию меню **Display | Heads Up Display | Frame Rate** (Отобразить Большой экран | Частота кадров). Вы обнаружите, что частота изменяется. Это происходит оттого, что скорость, с которой Maya способна воспроизводить анимацию, зависит от сложности сцены и анимации. Если сцена оказалась сложной, Maya может быть вынуждена отбрасывать некоторые кадры ради поддержания желаемой скорости. Коль скоро система постоянно оценивает количество отбрасываемых кадров, вы и наблюдаете в результате их «прыгающую» частоту.

4. Уменьшите размер текущего окна, чтобы отобразить все окна просмотра.
5. В редакторе **Script Editor** выполните следующую команду:

```
playbackOptions -view "all";
```

Анимация будет воспроизводиться не только в текущем окне, но и во всех других окнах просмотра. Некоторые параметры воспроизведения фиксируются в числе прочих настроек и сохраняются вплоть до следующего сеанса работы. Среди них параметры, определяющие скорость, зацикливание и работу с окнами просмотра. Чтобы восстановить значения этих параметров по умолчанию, используйте следующие команды:

```
playbackOptions -loop "continuous";  
playbackOptions -playbackSpeed 1;  
playbackOptions -view "active";
```

6. Щелкните по кнопке **Play**, чтобы остановить анимацию.

Пользуясь средствами языка **MEL**, вы можете управлять инструментом **Playblast**. С его помощью Maya воспроизводит текущую анимацию, сохраняя каждый кадр в анимационном файле с целью его дальнейшего отображения. При демонстрации каждого кадра Maya фактически осуществляет захват экрана, поэтому важно, чтобы текущее окно просмотра было раскрыто на протяжении всего периода захвата изображения. Следующая команда активизирует **Playblast**, а затем помещает результат в файл с именем **test.mov**.

```
playblast -filename test.mov;
```

3.5.3. Анимационные кривые

Функции

Анимация в Maya, в основе своей, является не чем иным, как результатом изменений значения атрибута с течением времени. При воспроизведении анимации атрибут в каждый момент времени имеет определенное значение. Если он анимирован, то при переходе от одного временного отсчета к другому значение будет изменяться. Говорят, что изменяющееся значение зависит от времени. Эта связь между текущим временем и тем или иным значением может быть описана при помощи математической функции. В других пакетах анимационные кривые и в самом деле называются функциональными. Функция определяется следующим образом:

$$y = \text{ад}^*$$

Эта запись, по существу, указывает на то, что функция (f) принимает входное значение (x) и возвращает выходное значение-результат (y). Функция сама по себе может быть очень простой или очень сложной. Нам не столь важно, как именно рассчитывается ее значение. Важно понять то, что, принимая одно значение на вход, функция выдает другое на выход. Скажем, у вас есть функция, которая просто увеличивает значение x на 2. Этую функцию можно записать так:

$$y = x + 2.$$

Пусть нам дано произвольное множество значений x . Передадим его на вход этой функции и посмотрим, какие числа получатся в результате. Если данное множество значений x содержит значения (0, 1, 2, 3), то значения y составят в итоге (2, 3, 4, 5). Объединив каждое значение x с соответствующим результатом y , можно получить точку (x, y) . На рис. 3.4 показан результат нанесения этих точек на координатную плоскость.

Теперь, проведя через точки линию, вы получите прямую. Итак, функция $y = x + 2$ описывает прямую линию. В зависимости от того, какое уравнение имеет функция, вы можете получать разные множества точек, а в результате построения линий - разные кривые. К примеру, гладкую синусоидальную кривую можно получить, если взять функцию $y = \sin(x)$.

А сейчас представим себе, что значение x соответствует некоторому моменту времени. Передав своей функции новое временное значение, мы получим новый результат y . Именно так происходит анимация атрибутов. Описав функцию, которая принимает на вход время и выдает результат, можно получить ряд изменяющихся значений.

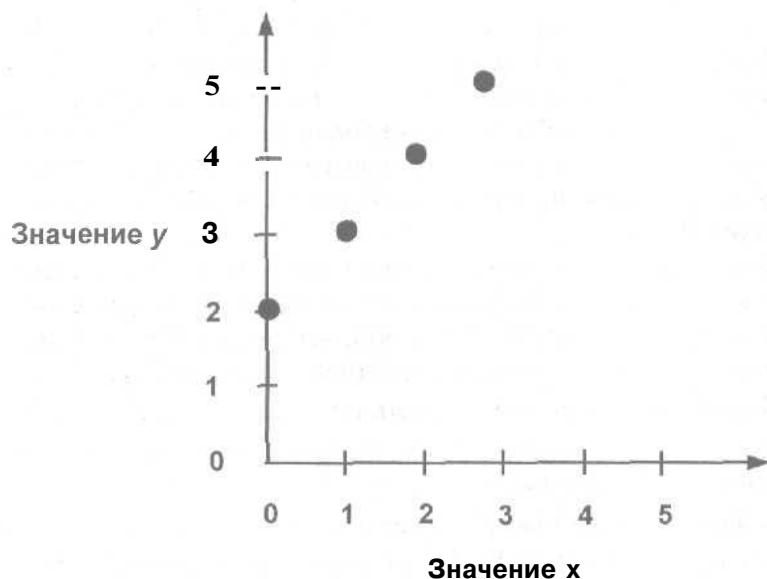


Рис. 3.4. График значений x и y

Зная об этом, вы обнаружите, что все анимационные кривые Maya, по идеи, могут считаться математическими функциями. Для данного значения x они вычисляют результат, т. е. значение y. Устанавливая ряд ключевых кадров, вы, по сути, задаете математическую функцию. Преимущество такого подхода состоит в том, что вы делаете это наглядно, добавляя ключевые кадры и манипулируя ими, а не записывая функцию в виде уравнения. Пользуясь выражениями Expression, речь о которых пойдет в одном из следующих разделов, вы действительно сможете описывать свои собственные математические функции для анимации атрибутов.

Еще один способ выражения этой зависимости состоит в интерпретации математической функции как описания отображения одного множества значений (x) в другое множество (y). Коль скоро анимационные кривые являются функциями, они и реализуют такое отображение. Создавая анимацию во времени, вы описываете отображение времени в значение другого атрибута. Если, к примеру, вы анимируете атрибут `translateX`, то тем самым описываете закон отображения значений времени в значения `translateX`. Ключевые кадры содержат это отображение в явном виде. Все они имеют входное и выходное значение:

ключевой кадр = (входное значение, выходное значение).

К примеру, для того чтобы описать ключевой кадр, который при заданном входном значении 3 отображает его в значение 2, нужно создать ключ (3, 2). Создав ключевые кадры для каждого входного значения, вы явно опишете отображение x в y . Выбрав иную стратегию, вы можете описать ограниченное количество ключевых кадров, позволив компьютеру определить отображение в местах пропуска ключей посредством интерполяции. Именно так работают кривые анимации по ключевым кадрам.

В Maya допустимо отображение друг в друга произвольных значений. Выполняя анимацию во времени, вы отображаете его в некоторое вещественное значение. Например, анимируя трансляцию объекта, вы описываете отображение времени в расстояние. Тогда ключевые кадры имеют следующий вид:

ключевой кадр = (время, расстояние).

Реализуя анимацию вращения, вы описываете отображение времени в величину угла, поэтому ключевые кадры имеют вид

ключевой кадр = (время, угол).

Управляемый ключ представляет собой всего лишь отображение одного значения в другое.

ключевой кадр = (входное значение, выходное значение).

Управляющий атрибут передается в качестве входного значения, а управляемый атрибут устанавливается равным выходному значению-результату. Понимание **того**, что анимация на основе времени не имеет концептуальных отличий от управляемой анимации, открывает перед вами широкое поле возможностей. Любой вход можно отобразить в произвольный выход. К примеру, вы можете отобразить величину угла в значение времени. В итоге по данному входному углу будет получено известное значение времени.

ключевой кадр = (угол, время).

Затем этот результат, т. е. время, можно передать на **вход** другой анимационной кривой, использующей время в качестве своего входа. Ограничений по способу соединения различных типов отображений не существует.

Установка ключевых кадров

Анимационная кривая состоит из множества управляющих точек (ключевых кадров, ключей) и связанных с ними касательных. Первые служат для описания ключей анимации. Вторые определяют способ интерполяции значений между ключами. Существует немало способов описания интерполяции, включая линейную, ступенчатую, сплайновую, плоскую интерполяцию и др.

Процесс создания кривых анимации заметно упрощается при использовании различных инструментов редактирования кривых, входящих в состав Maya. Эти инструменты позволяют в режиме диалога создавать точки кривых, касательные и сегменты, а также манипулировать ими. Наряду с применением языка MEL для интерактивного построения кривых, вы можете использовать его и для создания и редактирования кривых анимации.

В ходе анимации атрибута путем установки ключевых кадров Maya автоматически создает анимационную кривую и связывает ее с **анимируемым** атрибутом. Кривые анимации хранятся как узлы Dependency Graph. Для получения полного перечня всех узлов кривых анимации, которые связаны с данным узлом, воспользуйтесь командой `keyframe` так, как показано ниже.

```
keyframe -query -name ball;  
// Result: nurbsSphere1_translateX ball_scaleX //
```

Этот пример говорит о том, что узел с именем `ball` имеет два узла анимации, а именно `nurbsSphere1_translateX` и `ball_scaleX`. Чтобы получить узел анимационной кривой, который соответствует данному атрибуту, используйте команду

```
keyframe -query -name ball.translateX;  
// Result: nurbsSphere1_translateX //
```

Чтобы определить, представляет ли данный узел кривую анимации, обратитесь к команде `isAnimCurve`. В следующем примере показано получение узла анимационной кривой, связанного с атрибутом узла, а также выполнена проверка того, является ли полученный узел кривой анимации.

```
string $nodes[] = `keyframe -query -name ball.translateX`;  
if( isAnimCurve( $nodes[0] ) )  
    print( "is animation curve" ); // "является анимационной кривой"
```

Узнать о возможности анимации данного узла вам позволит команда `listAnimatable`.

```
listAnimatable -type ball;  
// Result; transform //
```

Определить, какие из атрибутов узла могут быть анимированы, вы можете и по-другому:

```
listAnimatable ball;  
// Result: ball.rotateX ball.rotateY ball.rotateZ ball.scaleX ball.scaleY  
    ball.scaleZ ball.translateX ball.visibility //
```

Команду `listAnimatable` можно выполнять и по отношению к текущим выделенным узлам, не указывая их явным образом. Просто удалите имя `ball` из оператора в предыдущем примере.

Вычисления

Чтобы получить значение анимированного атрибута, используйте команду `getAttr`. Ее результатом является значение атрибута в текущий момент времени. Побочным эффектом выполнения команды может стать обновление всех входных соединений узла анимационной кривой.

Если вы хотите оценить анимационную кривую в ее нынешнем состоянии, выполните следующий оператор:

```
keyframe -time 250 -query -eval ball.translateX;
```

Пользуясь флагом `-eval`, вы можете быстро находить отдельные значения на кривой анимации в различные моменты времени, и это не приведет к типичному обновлению графа Dependency Graph, затрагивающему все входные соединения кривой. Работая с анимационными кривыми на основе управляемых ключевых кадров, замените флаг `-time` на флаг `-float`.

Бесконечность

Допустимый диапазон кривой анимации описывается первым и последним ключевыми кадрами. Если первым ключевым кадром является 10-й, а последним - 25-й, то допустимый диапазон лежит между 10-м и 25-м кадрами. Любые попытки вычисления кривой вне этого диапазона ведут к получению значений, соответствующих бесконечности на кривой анимации. Эта кривая имеет настройки для фрагментов анимации, расположенных между первым ключевым кадром и бесконечностью, а также последним ключевым кадром и бесконечностью. Фрагмент от первого ключа до бесконечности расположен слева от первого ключевого кадра, фрагмент от последнего ключа до бесконечности расположен справа от последнего ключевого кадра. Эти фрагменты могут характеризоваться одной из следующих настроек,

`constant, linear, cycle, cycleRelative, oscillate`

По умолчанию «бесконечные» фрагменты всех кривых имеют настройку `constant`. При ее использовании значением атрибута становится значение ближайшего ключевого кадра. Для фрагмента от первого ключевого кадра до бесконечности это значение первого ключа, для фрагмента от последнего ключевого кадра до бесконечности это значение последнего ключа. Запросить текущие

настройки для бесконечности можно, воспользовавшись командой `setInfinity` с флагами `-preInfinite` или `-postInfinite`.

```
setInfinity -query -preInfinite ball.translateX  
// Result: constant //
```

Чтобы установить настройки для бесконечности, используйте команды `setInfinity -preInfinite "cycle" ball.translateX;` `setInfinity -postInfinite "linear" ball.translateX;`

Ключевые кадры

Здесь мы опишем создание и редактирование ключевых кадров анимационных кривых, а также организацию запросов к ним.

Создание

1. Выберите пункт меню **File \ New Scene...**
2. В редакторе **Script Editor** выполните следующие команды:

```
sphere;  
rename ball;
```

Теперь установите несколько ключевых кадров для трансляции мяча вдоль оси *x*.

3. Выполните следующие команды:

```
setKeyframe -time 1 -value -5 ball.translateX;  
setKeyframe -time 48 -value 5 ball.translateX;
```

По ним будут созданы ключи, соответствующие 1-му и 48-му кадрам. При вызове `setKeyframe` для не анимированного ранее атрибута происходит автоматическое создание узла анимационной кривой и его соединение с атрибутом. Чтобы увидеть, какой узел кривой анимации теперь управляет атрибутом `translateX`, используйте команду

```
keyframe -query -name ball.translateX;  
// Result: ball_translateX //
```

4. Щелкните по кнопке **Play**.

Мяч начнет двигаться по экрану.

5. Выберите пункт меню **Window | Animation Editors j Graph Editor...** (Окно j Редакторы анимации | Графический редактор...).

Перед вами появится кривая анимации для атрибута мяча с именем `translateX`. Сделаем так, чтобы значение трансляции мяча управляло его масштабом по оси *x*. Для этого нужно создать управляемый ключевой кадр.

6. Выполните следующие команды:

```
setDrivenKeyframe -driverValue 0 -value 0 -currentDriver  
    ball.translateX ball.scaleX  
setDrivenKeyframe -driverValue 5 -value 1 ball.scaleX;
```

Первым создается такой управляемый ключ, в котором атрибут мяча с именем **translateX** объявляется управляющим, а атрибут **scaleX** – управляемым атрибутом. При этом ключ строится так, что **scaleX** принимает нулевое значение, когда **translateX** также равен 0. Затем **создается** другой управляемый ключ. Он устанавливается таким образом, что значение **scaleX** становится равным 1, когда **translateX** будет равно 5.

7. Щелкните по кнопке **Play**.

Пока значение трансляции по оси **x** остается отрицательным, мяч выглядит плоским. По мере движения по направлению к точке **x = 5** он постепенно увеличивается в объеме. Почему же мяч остается плоским, пока не достигнет точки 0? Коль скоро диапазон данной анимационной кривой простирается от 0 до 5, попытки вычислить значение на кривой вне этого диапазона приводят к использованию значений, принятых для фрагментов от начального и последнего ключа до бесконечности. Так как по умолчанию эти значения постоянны, **scaleX** использует равное нулю значение первого ключевого кадра. Теперь удалите управляемую анимационную кривую.

8. Выполните следующие команды:

```
string $nodes[] = `keyframe -query -name ball.scaleX`;  
delete $nodes[0];
```

Первый оператор возвращает имя узла кривой анимации. Второй оператор удаляет узел.

Вставьте ключевой кадр в анимационную кривую для **translateX**. Выполните следующую команду:

```
setKeyframe -insert -time 24 ball.translateX;
```

Между двумя существующими ключами будет **добавлен** новый. Флаг **-insert** позволяет добавить новый ключ без изменения общей формы кривой анимации.

Редактирование

1. Откройте сцену **SimpleAnimatedSphere.ma**.
2. Выделите объект **ball**.
3. Выберите пункт меню **Window | Animation Editors | Graph Editor...**

На экране появится анимационная кривая сферы. Сейчас анимированным атрибутом является лишь **translateX**. В Maya все данные об анимации по ключевым кадрам хранятся в узлах кривой анимации. Узлы содержат список ключевых кадров. При **анимации** во времени каждый ключ состоит из временного и вещественного значения. Воспользуемся теперь языком MEL для редактирования ключевых кадров в анимационных узлах и выполнения запросов к ним.

4. Выполните в окне **Script Editor** следующую команду:

```
keyframe -query -keyframeCount ball.translateX;  
// Result: 2 //
```

Используя флаг **-keyframeCount**, вы можете определить количество ключевых кадров, расположенных на анимационной кривой.

5. В редакторе **Graph Editor** выделите последний ключ, расположенный в 48-м кадре.

6. Выполните следующую команду:

```
keyframe -query -selected -keyframeCount ball.translateX;  
// Result: 1 //
```

Она возвращает количество ключевых кадров, выделенных в настоящий момент. Чтобы определить время, соответствующее выделенным ключам, выполните следующее:

```
keyframe -query -selected -timeChange ball.translateX;  
// Result: 48 //
```

Чтобы получить фактическое значение атрибута в выделенных ключевых кадрах, выполните команду

```
keyframe -query -selected -valueChange ball.translateX;  
// Result: 1.64126 //
```

Для выполнения тех же операций над всеми ключевыми кадрами служат те же операторы, но без флага **-selected**. Теперь приступим к перемещению ключей. Выбор ключа для редактирования требует указания времени, которое ему соответствует. Выполните команду

```
keyframe -edit -time 48 -timeChange 20 ball.translateX;
```

Данная команда перемещает ключ из 48-го кадра в 20-й. Приписав в конце нужное обозначение, вы, как всегда, можете указать время в любых единицах измерения. Например, для редактирования кадра, соответствующего **моменту**

времени 1.5 секунды, просто запишите `-time 1.5sec`. По умолчанию изменения времени считаются абсолютными. Кроме того, ключи можно перемещать на относительную величину. Выполните следующую команду:

```
keyframe -edit -time 20 -relative -timeChange 5 ball.translateX;
```

В результате ключ из 20-го кадра сместится на 5 кадров вправо. Теперь он расположен в 25-м кадре. Для перемещения влево используйте отрицательное изменение времени, например `-5`. Чтобы изменить фактическое значение атрибута в ключевом кадре, выполните команду:

```
keyframe -edit -time 25 -valueChange 2 ball.translateX;
```

Теперь в 25-м кадре мяч займет положение, соответствующее 2 единицам по оси x. Как и при изменении времени, значение в ключевом кадре может изменяться на **относительную** величину. Использование в предыдущем примере флага `-relative` привело бы к увеличению существующего значения на 2. Если вам неизвестно точное значение времени в данном ключевом кадре, однако вы знаете его относительное положение в списке ключей, то можете воспользоваться флагом `-index`. Индексы **начинаются** с нуля, поэтому первый ключевой кадр имеет индекс 0, второй - 1 и т. д. Для перемещения первого ключа влево выполните следующую команду:

```
keyframe -edit -index 0 -relative -timeChange -5 ball.translateX;
```

Указывая несколько ключей при помощи флагов `-time` или `-index`, можно одновременно работать более чем с одним ключевым кадром. Чтобы переместить первый и второй ключи на 10 кадров вправо, выполните следующую команду:

```
keyframe -edit -index 0 -index 1 -relative -timeChange 10  
ball.translateX;
```

Кроме того, одну и ту же операцию можно применить к диапазону ключей. Диапазон может быть задан либо в единицах времени, либо при помощи индексов. Для увеличения на 5 единиц значений всех ключей, начиная с момента времени 0 и заканчивая моментом времени 12, выполните следующую команду:

```
keyframe -edit -time "0:12" -relative -valueChange 5 ball.translateX;
```

Единственным ключом в этом временном диапазоне является первый ключ. Его значение увеличится на 5 единиц. Указывая **диапазон**, вы можете привести только одну из его границ. Например, для работы со всеми ключами, предшествующими моменту времени 20, просто запишите `"::20"`.

Так как начальное значение не приведено, будет использован первый ключ. Заметьте, что в диапазон входят его границы, поэтому к ним относятся любые ключи, лежащие на краю диапазона. Чтобы задать диапазон времени, начиная с момента 20 и включая все последующие ключи, запишите `-time "20:"`. Отсюда логически вытекает, что для выбора всех ключей, согласно этой нотации, следует записать `-time ":"`. Выполните следующую команду для перемещения всех ключей на 3 кадра влево:

```
keyframe -edit -time ":" -relative -timeChange 3 ball.translateX;
```

Эта же запись диапазонов может использоваться и при работе с индексами. Для присвоения всем ключам с индексами с 1 по 20 значения 2.5 воспользуйтесь следующей командой:

```
keyframe -edit -index "1:20" -valueChange 2.5 ball.translateX;
```

Наряду с явным заданием времени и значений ключей, можно выполнить масштабирование их значений. Чтобы масштабировать значения всех ключей, выполните команду

```
scaleKey -time ":" -valueScale 0.5 ball.translateX;
```

Время, соответствующее ключевым кадрам, также можно масштабировать, для чего служит флаг `-timeScale`. Следующая команда вдвое уменьшает время, соответствующее каждому ключу.

```
scaleKey -timeScale 0.5 ball.translateX;
```

Масштабирование выполняется относительно опорной точки. Так называют начало отсчета при совершении этой операции. Чтобы задать новую опорную точку на временной шкале, используйте флаг `-timePivot`. Для масштабирования ключей относительно их центрального значения выполните следующие команды:

```
float $times[] = `keyframe -index 0 -index 1  
                      -query -timeChange ball.translateX`;  
scaleKey -timePivot (($times[0] + $times[1])/2)  
          -timeScale 2 ball.translateX;
```

Флаги `-valuePivot` и `-floatPivot` могут применяться для установки другого начала отсчета значений атрибутов и вещественных величин. В результате перемещения ключей может случиться так, что некоторые из них будут не соответствовать целым номерам кадров. Сейчас оба ключа имеют время, равное -2.752 и 26.248. Чтобы вновь связать их с целочисленными номерами кадров, выполните следующую команду:

```
snapKey -timeMultiple 1 ball.translateX;
```

Команда `snapKey` может применяться и для отдельных ключей, если вы зададите их явно или в виде диапазона. Флаг `-timeMultiple` не обязательно должен иметь целочисленное значение. К примеру, вы можете «привязать» ключевые кадры к интервалам длительностью полсекунды. В таком случае просто запишите `-timeMultiple 0.5sec`.

Кадры разбиения

В Maya существуют два типа ключей: обычные ключевые кадры и разбиения. Различие между ними состоит в том, что время, соответствующее кадру разбиения, определяется относительно ключей, которые *его* окружают. В отличие от обычных ключевых кадров, которые остаются на месте до тех пор, пока вы не передвинете их явным образом, разбиения пытаются сохранить свое положение на временной шкале относительно окружающих ключей при их перемещении.

Чтобы определить, является ли ключевой кадр разбиением, используйте флаги `-query` и `-breakdown`.

```
float $isBreakdown[] = `keyframe -time $keyTime  
    -query -breakdown $nodeAttribute`;  
if( $isBreakdown[0] )  
    ... // выполнить какие-либо действия
```

Превратить ключ в кадр разбиения можно так:

```
keyframe -time $keyTime -breakdown true $nodeAttribute;
```

Касательные

Направление кривой, по которому она входит в ключевой кадр и покидает его, определяется *касательной* к этой кривой. На рис. 3.5 изображена анимационная кривая, а также ее касательные.

Каждый кадр имеет *входящую* и *исходящую* касательную. Входящая касательная лежит слева от ключевого кадра, тогда как *исходящая* – справа. Касательные имеют такие свойства, как тип, угол, вес и блокировка. Тип касательной указывает на способ интерполяции значений. Используемый тип касательной определяет, как интерполируются значения между ключевыми кадрами. Ступенчатая касательная, к примеру, имеет постоянное значение на промежутке между ключами. Сплайновая касательная обеспечивает более гладкий переход от одного ключевого кадра к другому.

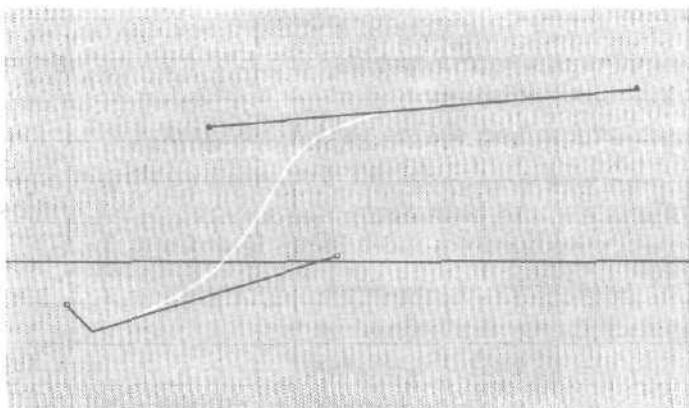


Рис. 3.5. Примеры типов касательных

Угол касательной определяет ее направление. Взвешенные касательные позволяют перемещать свои конечные точки, что дает полный контроль над такими касательными. Касательные поддерживают два типа блокировок. Первый - общая блокировка, при которой угол и длина входящей и исходящей касательной не подлежат изменению. Касательные можно поворачивать, но изменять их углы независимо друг от друга запрещено. Также неизменными остаются их длины. Кроме того, касательные поддерживают блокировку весов. Подобная блокировка происходит в тех случаях, когда веса касательных фиксируются, а их концы нельзя растягивать или сжимать. Хотя вращение касательных при этом все-таки допускается.

1. Откройте сцену **SimpleAnimatedSphere.ma**.
2. Выделите объект **ball**.
3. Выберите пункт меню **Window | Animation Editors | Graph Editor...**
4. В окне **Graph Editor** нажмите клавишу **f** для работы с текущей анимационной кривой.

Текущая кривая - это прямая линия между двумя ключами.

5. Выполните следующую команду:

```
keyTangent -index 0 -outTangentType flat ball.translateX;
```

При воспроизведении анимации мяч будет двигаться сначала медленно, а потом, к концу, все быстрее. Если вы хотите, чтобы мяч совершил резкий прыжок из одного положения в другое, выполните команду

```
keyTangent -index 0 -outTangentType step ball.translateX;
```

Мяч пребывает в исходном положении, а затем, в конце анимации, внезапно перескакивает на новое место. Для восстановления первоначального типа касательной выполните следующую команду:

```
keyTangent -index 0 -outTangentType spline ball.translateX;
```

На месте первого ключа касательные разрываются. Установка некоторых их типов влияет как на входящую, так и на исходящую касательную, и для возврата в начальное состояние может потребоваться смена обеих. Выполните команду

```
keyTangent -index 0 -inTangentType spline ball.translateX;
```

Теперь поверните первую касательную в положение 45° :

```
keyTangent -index 0 -inAngle 45 ball.translateX;
```

Все углы определяются относительно горизонтальной оси абсцисс и отсчитываются в направлении против часовой стрелки. Касательные врачаются постольку, поскольку их угол автоматически блокируется из-за использования касательных «сплайнового» типа. Повороты также могут быть относительными.

```
keyTangent -index 0 -relative -inAngle 15 ball.translateX;
```

Угол входящей касательной увеличен на 15° . Чтобы узнать направление исходящей касательной как единичного вектора (длины 1), используйте флаги `-ox` и `-oy`.

```
float $dir[] = `keyTangent -index 0 -query -ox -oy ball.translateX`;
print $dir;
```

Результат:

0.153655

0.988124

Тот же оператор, но с флагами `-ix` и `-iy`, возвращает направление входящей касательной. Теперь разблокируйте входящую и исходящую касательные, разрешив тем самым их независимое вращение.

```
keyTangent -index 0 -lock false ball.translateX;
```

В редакторе Graph Editor касательные изображены разными цветами. Вращение исходящей касательной больше не должно влиять на входящую.

```
keyTangent -index 0 -outAngle 15 ball.translateX;
```

Чтобы приписать касательным тот или иной вес, нужно для перехода к взвешенным касательным преобразовать кривую анимации в целом.

```
keyTangent -edit -weightedTangents yes ball.translateX;
```

Теперь кривая содержит взвешенные касательные. Их конечные точки показаны в окне **Graph Editor** по-другому. Важно знать о возможности искажения кривой при ее преобразовании с использованием взвешенных касательных и дальнейших попытках вернуться к невзвешенному варианту. Касательные, не обладающие весом, теряют весовую информацию, поэтому почти всегда кривая становится иной. Изменения весов касательных второго ключевого кадра можно добиться следующим образом:

```
keyTangent -index 1 -inWeight 20 ball.translateX;
```

Коль скоро веса обеих касательных по умолчанию заблокированы, вам нужно разблокировать их для того, чтобы работать с весами каждой из них независимо друг от друга. Необходимо разблокировать как сами касательные, так и их веса.

```
keyTangent -index 1 -lock false -weightLock no ball.translateX;
```

Теперь вы можете изменить вес одной касательной, не затрагивая другой.

```
keyTangent -index 1 -inWeight 10 ball.translateX;
```

Вес входящей касательной изменен, и это изменение не затронуло веса исходящей касательной.

Выполняя первоначальную анимацию атрибута, Maya строит анимационную кривую. По мере добавления ключевых кадров типы и веса касательных определяются автоматически в соответствии с установками **Tangents** (Касательные) в разделе **Window j Settings/Preferences | Preferences... | Keys** (Окно | Установка/Параметры | Параметры... | Ключевые кадры). Изменив **их**, вы можете быть уверены в том, что все ключевые кадры, созданные с этого момента, руководствуются новыми установками. Для запроса текущих установок воспользуйтесь следующими **командами**:

```
keyTangent -query -global -inTangentType; // Запрос типа входящей касательной
```

```
keyTangent -query -global -outTangentType; // Запрос типа исходящей касательной
```

```
keyTangent -query -global -weightedTangents; // Запрос автоматического задания веса
```

Для установки входящей касательной по умолчанию, наберите следующую команду:

```
keyTangent -global -inTangentType flat;
```

Для установки такой входящей касательной, при которой все новые анимационные кривые станут автоматически обладать весом, используйте следующую команду:

```
keyTangent -global -weightedTangent yes;
```

Буфер обмена для ключевых кадров

Ключевые кадры можно вырезать, выполнять их копирование и вставку. Maya содержит буфер обмена, куда временно помещаются вырезанные и скопированные ключевые кадры. Оттуда их можно вставлять в иные временные фрагменты той же анимационной кривой или в другую кривую анимации. В буфер обмена можно поместить и копию всей серии ключевых кадров.

1. Откройте сцену **SimpleAnimatedSphere.ma**.
2. Выделите объект **ball**.
3. Выполните следующую команду:

```
copyKey -index 0 ball.translateX;  
// Result: 1 //
```

В буфер обмена скопирован первый ключ. Теперь вставьте его, придав ему новое положение на шкале времени.

```
pasteKey -time 12 -option insert ball.translateX;
```

Вы произвели вставку первого ключа, поместив его на кривую в место расположения 12-го кадра.

Значение ключа уменьшено наполовину. Чтобы удалить ключ и поместить его в буфер обмена, используйте команду **cutKey**.

```
cutKey -index 1 -option keys ball.translateX;
```

Чтобы удалить ключ, не помещая его в буфер обмена, выполните команду **cutKey** с флагом **-clear**.

```
cutKey -index 1 -clear ball.translateX;
```

Примеры

Несколько следующих примеров призваны продемонстрировать приемы создания и редактирования анимационных кривых на основе ключевых кадров.

Сценарий printAnim

Широко распространенной задачей является организация цикла по всем кривым анимации и получение входящих в них ключевых кадров. В данном сценарии описана процедура **printAnim**, которая выводит на экран информацию

об анимации узлов, выделенных в настоящее время. При запросе подробной информации процедура также показывает тип ключа и сведения о касательных.

```
proc printAnim( int $detailed )  
{  
    print "\nAnimation..."; // "Анимация..."  
    string $animNodes[];  
    float $keytimes[3];  
    string $sel[] = `ls -selection`;  
    for( $node in $sel )  
    {  
        print ("\nNode: " + $node); // "Узел: "  
        $animNodes = `keyframe -query -name $node`;  
        for( $ac in $animNodes )  
        {  
            print ("\nAnimCurve: " + $ac); // "Кривая: "  
            $keytimes = `keyframe -query -timeChange $ac`;  
            print ("\n" + size($keytimes) + " keys: " ); // "кадра(-ов)"  
            for( $keyt in $keytimes )  
            {  
                $keyv = `keyframe -time $keyt -query -valueChange $ac`;  
                if( $detailed )  
                {  
                    float $isBd[] = `keyframe -time $keyt -query -breakdown $ac`;  
                    print ("\n" + ($isBd[0] ? "Breakdown" : "Normal") + " Key: ");  
                    // "Разбиение" : "Обычный ключ"  
                }  
                print (" [ " + $keyt + ", " + $keyv[0] + " ]");  
            }  
            if( $detailed )  
            {  
                print ("\nTangent: "); // "Касательная: "  
                $keyinT = `keyTangent -time $keyt -query -inTangentType $ac`;  
            }  
        }  
    }  
}
```

Первым шагом после описания процедуры является получение перечня выделенных узлов. Затем осуществим перебор всех узлов *перечня*.

```
string $sel[] = `ls -selection`;
for( $node in $sel )
```

Используя команду `keyframe`, можно получить список всех анимационных кривых, связанных с данным узлом. Далее организуем цикл по кривым анимации.

```
$animNodes = `keyframe -query -name $node`;  
for( $ac in $animNodes )
```

Получим список всех временных отсчетов, соответствующих ключевым кадрам текущей анимационной кривой.

```
$keytimes = `keyframe -query -timeChange $ac`
```

Количество ключей определяется как размер массива `$keytimes`.

```
print ("\\n" + size($keytimes) + " keys: ");
```

Располагая списком временных отсчетов, вы можете **реализовать** цикл по всем ключевым **кадрам**, поскольку способны **проиндексировать** ключи по времени. Кроме того, ключи можно проиндексировать и по порядку их размещения.

```
for( $keyt in $keytimes )
```

Найдем значение атрибута текущего ключевого кадра. В данном случае важно указать целевой ключ при помощи флага `-time $keyt`, записанного перед фла-

гом -query. Если вы не выдержите такой порядок следования аргументов, команда закончится неудачно.

```
$keyv = `keyframe -time $keyt -query -valueChange $ac`;
```

При выводе подробной информации отображается тип ключевого кадра. Ключ может являться либо кадром разбиения, либо обычным ключевым кадром.

```
if( $detailed )
{
    float $isBd[] = `keyframe -time $keyt -query -breakdown $ac`;
    print ("\n" + C$isBd[0] ? "Breakdown" : "Normal") + " Key:" );
}
```

На экран выводится время и значение атрибута текущего ключа.

```
print (" [ " + $keyt + ", " + $keyv[0] + " ]");
```

Для запроса сведений о касательных используется команда keyTangent с флагом -query. При этом определяются четыре различных свойства касательных: `inTangentType`, `outTangentType`, `inAngle`, `outAngle`. Касательные могут иметь один из перечисленных типов: spline, linear, fast, slow, flat, step, fixed и clamped. Входящий и исходящий углы описывают углы касательных, а те, в свою очередь, задают их направление,

```
if( $detailed )
{
    print ("\nTangent: ");
    $keyinT = `keyTangent -time $keyt -query -inTangentType $ac`;
    $keyoutT = `keyTangent -time $keyt -query -outTangentType $ac`;
    $keyinA = `keyTangent -time $keyt -query -inAngle $ac`;
    $keyoutA = `keyTangent -time $keyt -query -outAngle $ac`;
    print ("(" + $keyinT[0] + " angle=" + $keyinA[0] + ", "
           + $keyoutT[0] + " angle=" + $keyoutA[0] + ")");
}
```

Ниже приведен пример выходной информации, которую выдает процедура `printAnim`.

```
Animation...
Node: ball
AnimCurve: nurbsSphere1_translateX
2 keys:
```

```

Normal Key: [1, -5.3210074]
Tangent: (spline angle=8.426138245, spline angle=8.426138245)
Normal Key: [48, 1.641259667]
Tangent: (spline angle=8.426138245, spline angle=8.426138245)
AnimCurve: ball_scaleX
0 keys:

```

Сценарий printTangentPositions

Выдавая информацию о касательной, Maya представляет ее в виде направления и веса. Для точного определения положения конечной точки касательной эту информацию нужно перевести в декартовы координаты. Следующий сценарий отображает сведения о конечных точках касательных. Данные могут быть выведены относительно соответствующих ключевых кадров или по абсолютному положению относительно начала отсчета.

```

proc printTangentPositions( string $animCurve, int $absolute )
{
    print ("\\nTangent Positions..."); // "Точки касательных..."

    float $ktimes[], $kvalues[];
    if( $absolute )
    {
        $ktimes = `keyframe -query -timeChange $animCurve`;
        $kvalues = `keyframe -query -valueChange $animCurve`;
    }

    float $xcomps[], $ycomps[], $weights[];
    int $i, $j;
    for( $i=0; $i < 2; $i++ )
    {
        string $xreq, $yreq, $wreq;
        if( $i == 0 )
        {
            $xreq = "-ix";
            $yreq = "-iy";
            $wreq = "-inWeight";
        }
    }
}

```

```
else
{
    $xreq = "-ox";
    $yreq = "-oy";
    $wreq = "-outWeight";
}

$xcomps = `keyTangent -query $xreq $animCurve`;
$ycomps = `keyTangent -query $yreq $animCurve`;
$weights = `keyTangent -query $wreq $animCurve`;

print ("\n");
for( $j=0; $j < size($xcomps); $j = $J + 1 )
{
    $xcomps[$j] *= $weights[$j];
    $ycomps[$j] *= $weights[$j];
    if( $absolute )
    {
        $xcomps[$j] += $ktimes[$j];
        $ycomps[$j] += $kvalues[$j];
    }

    print (" [" + $xcomps[$j] + ", " + $ycomps[$j] + "]");
}
}

proc testProc()
{
    string $animCurves[] = `keyframe -query -name ball.translateX`;
    printTangentPositions( $animCurves[0], true );
}

testProc();
```

Процедура printTangentPositions принимает на вход имя узла анимационной кривой (`$animCurve`), а также признак того, планируется ли вывод абсолютного или относительного положения точек касательных (`$absolute`).

```
proc printTangentPositions( string $animCurve, int $E.absolute )
{
```

Если вам нужно получить абсолютное положение точек касательных, то потребуется знать местонахождение связанных с касательными **ключевых** кадров. Запросим значения времени и значения атрибута всех ключевых кадров.

```
float $ktimes[], $kvalues[];
if( $absolute )
{
    $ktimes = `keyframe -query -timeChange $animCurve`;
    $kvalues = `keyframe -query -valueChange $animCurve`;
}
```

Существуют две касательные - входящая и исходящая. Необходимо организовать цикл по каждой из них.

```
for( $i=0; $i < 2; $i++ )
{
```

В зависимости от того, какая **касательная** вам нужна, исправьте флаги команд (`$xreq`, `$yreq`, `$wreq`) так, чтобы получить x- и y-компоненты векторов, а также веса касательных. Также это можно было записать при помощи оператора `switch`.

```
string $xreq, $yreq, $wreq;
if( $i == 0 )
{
    $xreq = "-ix";
    $yreq = "-iy";
    $wreq = "-inWeight";
}
else
{
    $xreq = "-ox";
    $yreq = "-oy";
    $wreq = "-outWeight";
}
```

Теперь запросите *x*-, *y*-компоненты и веса всех касательных.

```
$xcomps = "keyTangent -query $xreq $animCurve";
$ycomps = `keyTangent -query $yreq $animCurve`;
$weights = `keyTangent -query $wreq $animCurve`;
```

Затем организуйте цикл по каждой касательной.

```
for( $j=0; $j < size($xcomps); $J = $J + 1 )
{
```

x- и *y*-компоненты образуют единичный вектор, указывающий направление касательной. Вес определяет длину вектора касательной. Умножив единичный вектор на длину касательной, вы получите вектор, имеющий направление и длину, которые совпадают с направлением и длиной касательной.

```
$xcomps[$j] *= $weights[$j];
$ycomps[$j] *= $weights[$j];
```

В данный момент векторы соотнесены с ключевым кадром, т. е. представляют собой смещения относительно позиции ключевых кадров. Если вы хотите, чтобы положение точек касательных соотносилось с тем же началом отсчета, что и положение ключей, то положение ключей и векторы касательных необходимо сложить.

```
if( $absolute )
```

```
{
    $xcomps[$j] += $ktimes[$j];
    $ycomps[$j] += $kvalues[$j];
}
```

Выведите окончательное положение конечной точки касательной.

```
print (" [" + $xcomps[$j] + ", " + $ycomps[$j] + "]");
```

Для проверки сценария printTangentPositions создана небольшая тестовая процедура testProc. Она выводит на экран абсолютное положение конечных точек анимационной кривой, построенной для атрибута **translateX** узла ball.

```
proc testProc()
{
    string $animCurves[] = `keyframe -query -name ball.translateX`;
    printTangentPositions( $animCurves[0], true );
}
```

```
testProc0;
```

Далее приведен пример выходной информации, которую выдает этот сценарий.

Tangent Positions...

[1.009253906, -5.311753494] [60.4534152, 14.09467487]

[32.76284582, 26.44183842] [8113845077, 34.77971043]

3.5.4. Скелеты

Скелет - это всего лишь иерархия суставов, которые, в свою очередь, имитируют внутренние кости персонажей. Именно иерархические связи между суставами («родитель - потомок») лежат в основе образования скелета. Поворачивая отдельные суставы, персонажи могут принимать нужные позы. К суставам обычно «привязываются» модели персонажей. Этого достигают благодаря процессу *образования покрова*, также известному как *прорисовка оболочек*. Затем, когда суставы приводят в *движение*, это оказывает влияние на модель. Каждый сустав влияет на часть модели и вызывает ее *деформацию*, позволяющую *вообразить*, будто модель обладает мускулами и кожными покровами, скрывающими ее кости.

Скелет обычно создается вручную. Связано это с трудностями автоматической генерации скелета конкретной модели. В зависимости от тех или иных нужд анимации скелет может потребовать применения различных конфигураций суставов. К примеру, модель *велосипедиста* может потребовать дополнительных шейных суставов, а модель человека - нет.

Если вам нужно создать скелет при помощи языка MEL, используйте команду **joint**.

1. Выберите пункт меню **File | New Scene...**
2. Активизируйте окно просмотра **top** и увеличьте его размер до максимума.
3. В редакторе **Script Editor** выполните следующую команду:

joint;

В начале координат появится единственный сустав **joint1**. Выполните следующую команду:

joint -position 0 0 10;

Создан другой сустав **joint2**, ставший потомком первого сустава **joint1**. Поскольку первый сустав был все еще выделен, вызов команды **joint** привел к автоматическому добавлению нового сустава в качестве его потомка. Для *вставки* еще одного сустава между двумя существующими воспользуйтесь командой

insertJoint joint1;

Будет добавлен новый сустав **joint3**. Теперь он является потомком **joint1** и родителем **joint2**. Иерархия суставов сейчас выглядит так:

```
joint1
  joint3
    joint2
```

Обратите внимание на то, что новый сустав находится в начале координат. К сожалению, команда **insertJoint** не позволяет указать начальное положение сустава. Чтобы изменить положение сустава после его создания, используйте команду

```
joint -edit -position 0 0 5 joint3;
```

Сустав **joint3** сдвигается вниз. Его потомок **joint2** также перемещается на равное расстояние. Суставы участвуют в отношении «родитель - ПОТОМOK», поэтому при любом движении предка потомок последует за ним. Поменяйте положение узла **joint3**, и узел **joint2** будет следовать этому перемещению. Если вы хотите передвинуть сустав, не затрагивая его потомков, можете использовать флаг **-component**. Выполните следующую команду:

```
undo;
```

```
joint -edit -position 0 0 5 -component joint3;
```

Предыдущий сдвиг **отменяется**, после чего узел **joint3** сдвигается, однако его дочерние суставы остаются на месте. По умолчанию положение любых узлов задается в мировом пространстве. Более удобным может оказаться **указание положения сустава относительно его родителя**. Это становится возможным с использованием флага **-relative**.

```
joint -edit -relative -position 5 0 0 joint2;
```

Новое положение **joint2** — на 5 единиц правее его родителя **joint3**. Теперь рассмотрим некоторые возможности поворота суставов. По умолчанию создаваемый сустав обладает тремя степенями свободы вращательных движений, т. е. объект может свободно поворачиваться вдоль осей **x**, **y** и **z**. Для определения текущих степеней свободы воспользуйтесь следующей командой:

```
joint -query -degreeOfFreedom joint1;
```

Результат имеет вид:

```
// Result: xyz //
```

Он указывает на то, что сустав совершенно свободен. Чтобы устраниТЬ возможность вращения по всем осям, кроме оси **y**, выполните команду

```
joint -edit -degreeOfFreedom "y" joint1;
```

Новое значение флага `-degreeOfFreedom` полностью заменяет все ранее сделанные установки. Любые попытки поворота сустава вдоль осей `x` и `z` будут заканчиваться неудачей. Кроме того, вы можете ограничить возможный диапазон вращения по любой конкретной оси. По умолчанию таких ограничений нет.

```
joint -edit -limitSwitchY yes joint1;
```

Теперь ограничения поворота по оси `y` вступили в силу. Запрос текущих ограничений выполняется так:

```
joint -query -limitY Joint1;
```

Результат:

```
// Result: -360 360 //
```

В данный момент сустав свободно поворачивается на угол от -360° до $+360^{\circ}$. Ограничить повороты сустава диапазоном $0 - 90^{\circ}$ можно следующим образом:

```
joint -edit -limitY 90deg Joint1;
```

Обратите внимание на то, что для указания углов была использована опция `deg`. Коль скоро вы можете не знать заранее текущую единицу измерения углов, лучше всего явно задать угол в требуемых единицах. Теперь поверните сустав на `угол`, выходящий за пределы диапазона.

```
rotate 0 200deg 0 Joint1;
```

Сустав поворачивается на максимальный угол, равный 90° . Для поворота сустава относительно его родителя воспользуйтесь `следующей` командой:

```
joint -edit -angleY 10deg joint3;
```

Сустав `joint3` повернется на 10° по оси `y`. Важно заметить то, что все повороты суставов по команде `joint` являются относительными. В абсолютных координатах может быть задано только положение суставов. Чтобы запросить текущие углы поворота сустава, используйте команду

```
xform -query -rotation joint3;
```

Результат:

```
// Result: 0 10 0 //
```

Для удаления сустава служит команда `removeJoint`.

```
removeJoint joint3;
```

Она удаляет сустав, а затем автоматически придает всем «висячим» дочерним суставам статус потомков родителя удаленного узла. Если бы вместо нее была использована команда `delete`, вы удалили бы узел `joint3`, а также все суставы, являющиеся его потомками.

Сценарий outputJoints

Этот сценарий принимает на вход иерархию суставов, а затем выводит положение, углы поворота и масштаб суставов в заданном интервале времени. Выходная информация сохраняется в файл и отображается в окне Maya. Сценарий работает со скелетами, которые используют либо не используют инверсно-кинематические (ИК) описатели. Этот сценарий особенно полезен при экспорте информации о суставах в приложения, не имеющие средств поддержки инверсной кинематики. Сведения о преобразованиях суставов могут быть представлены в локальном или мировом пространстве.

```

proc writeArray( int $fileHnd, float $array[] )
{
    float $v;
    for( $v in $array )
        fwrite $fileHnd $v;
}

proc outputJoints( string $rootNode, string $filename,
                   float $startFrame, float $endFrame,
                   int $outputWS )
{
    int $fileHnd = `fopen $filename w`;
    if( $fileHnd == 0 )
    {
        error ("Unable to open output file " + $filename + " for writing");
        // "Невозможно открыть выходной файл для записи"
        return;
    }

    string $childNodes[] = `listRelatives -fullPath
                           -type joint -allDescendents $rootNode`;
    string $rn[] = { $rootNode };
    string $nodes[] = stringArrayCatenate( $rn, $childNodes );

    float $cTime = `currentTime -query`;

```

```
string $spaceFlag = ($outputWS) ? "-worldSpace" : "-objectSpace";

print "\nOutputting joints..."; // "Вывод суставов..."
float $t;
for( $t = $startFrame; $t <= $endFrame; $t++ )
{
    currentTime -update false $t;
    fwrite $fileHnd $t;
    print ("\nFrame: " + $t); // "Кадр: "

    for( $node in $nodes )
    {
        fwrite $fileHnd $node;
        print ("\n Joint: " + $node); // "Сустав: "

        float $pos[] = `xform $spaceFlag -query -translation $node`;
        float $rot[] = `xform $spaceFlag -query -rotation $node`;
        float $scl[] = `xform $spaceFlag -query -relative -scale $node`;

        writeArray( $fileHnd, $pos );
        writeArray( $fileHnd, $rot );
        writeArray( $fileHnd, $scl );

        print ("\n pos=[ " + $pos[0] + " "
               + $pos[1] + " " + $pos[2] + " ]");
        print ("\n rot=[ " + $rot[0] + " "
               + $rot[1] + " " + $rot[2] + " ]");
        print ("\n scl=[ " + $scl[0] + " "
               + $scl[1] + " " + $scl[2] + " ]");
    }
}

currentTime -update false $cTime;
```

```
fclose $fileHnd;
}
```

Прежде чем перейти к разбору этого сценария, посмотрите на него в действии.

1. Откройте сцену **IKChain.ma**.

Сцена -состоит из скелета- иерархии, управляемой описателем инверсной кинематики.

2. Щелкните по кнопке **Play**.

Цепочка суставов начнет свое движение слева направо.

3. В редакторе **Script Editor** введите предшествующий текст, а затем выполните его.

4. Подставьте другое имя файла, если на вашей машине или в вашей операционной системе команда с данным именем работать не будет.

```
outputJoints( "joint1", "c:\\temp\\joints.txt", 1, 10, false );
```

Информация о суставах будет выведена для всех кадров с 1-го по 10-й. Данные о преобразованиях выводятся в пространстве объекта. Ниже приведен результат.

```
Outputting joints...
```

```
Frame: 1
```

```
Joint: joint1
```

```
pos=[ -12.03069362 4.785929251 2.314326079e-007 ]
```

```
rot=[ -2.770645083e-006 -3.756997063e-007 50.410438 ]
```

```
scl=[ 1 1 1 ]
```

```
Joint: |joint1|joint2|joint3|joint4
```

```
pos=[ 7.326328636 8.071497782e-017 7.585667703 ]
```

```
rot=[ 0 0 0 ]
```

```
scl=[ 1 1 1 ]
```

Эта информация о суставах также выводится в указанный файл.

Теперь рассмотрим этот сценарий подробно. В нем описана простая служебная процедура **writeArray**, которая выводит массив (**\$array**) в файл (**\$fileHnd**).

```
proc writeArray( int $fileHnd, float $array[] )
```

```
{
```

```
7*
```

```

float $v;
for( $v in $array )
    fwrite $fileHnd $v;

```

1

Следом описана процедура `outputJoints`. Ее вход - имя корневого сустава скелета (`$rootNode`), а также имя выходного файла (`$filename`). Диапазон анимации задан переменными `$startFrame` и `$endFrame`. Наконец, переменная `$outputWS` служит признаком того, выводятся ли преобразования в мировом или объектном пространстве.

```

proc outputJoints( string $rootNode, string $filename,
                    float $startFrame, float $endFrame, int $outputWS )

```

\\

Файл открывается по команде `fopen`. Она возвращает описатель файла. Все последующие обращения к файлу используют этот описатель.

```
int $fileHnd = `fopen $filename w`;
```

Если полученный описатель файла равен нулю, то `открытие` файла закончилось неудачно. Уведомим об этом пользователя и завершим работу.

```
if( $fileHnd == 0 )
```

```

{
    error ("Unable to open output file " + $filename + " for writing");
    return;
}
```

Получим перечень всех дочерних суставов.

```
string $childNodes[] = `listRelatives -fullPath -type joint
                        -allDescendents $rootNode`;
```

Организуем список всех суставов скелета, внеся в этот список корневой узел в качестве первого узла, за которым следуют все его потомки. Так как процедура `stringArrayCatenate` требует наличия двух входных массивов, поместим узел `$rootNode` во временный массив `$rn`.

```
string $rn[] = { $rootNode };
string $nodes[] = stringArrayCatenate( $rn, $childNodes );
```

Зафиксируем текущее время. Позднее вам потребуется к нему вернуться.

```
float $cTime = `currentTime -query`;
```

Переменная \$spaceFlag определяет, в каком пространстве будет получена информация о преобразованиях.

```
string $spaceFlag = ($outputWS) ? "-worldSpace" : "-objectSpace";
```

Далее выполним цикл по всем кадрам заданного диапазона.

```
float $t;  
for( $t = $startFrame; $t <= $endFrame; $t++ )  
{
```

Текущее время принимает значение \$t. Заметьте: сцена не обновляется, устанавливается только время.

```
currentTime -update false $t;
```

Время выводится в файл, а также на экран.

```
fwrite $fileHnd $t;  
print ("\nFrame: " + $t);
```

Организуется цикл по всем суставным узлам.

```
for( $node in $nodes )  
{
```

Имя узла выводится в файл и на экран.

```
fwrite $fileHnd $node;  
print ("\n Joint: " + $node );
```

Запрашивается информация о трансляции, поворотах и масштабе текущего сустава. Вам не удастся получить абсолютное значение коэффициента масштабирования, поэтому запрос масштаба должен содержать флаг -relative.

```
float $pos[] = `xform $spaceFlag -query -translation $node`;  
float $rot[] = `xform $spaceFlag -query -rotation $node`;  
float $scl[] = `xform $spaceFlag -query -relative  
-scale $node`;
```

Информация о преобразованиях записывается в файл и выводится на экран.

```
writeArray( $fileHnd, $pos );  
writeArray( $fileHnd, $rot );  
writeArray( $fileHnd, $scl );  
  
print ("\n pos=[ " + $pos[0] + " "  
+ $pos[1] + " " + $pos[2] + " ]");
```

```

print ("\\n    rot=[ " + $rot[0] + " "
      + $rot[1] + " " + $rot[2] + " ]");
print ("\\n    scl=[ " + $scl[0] + " "
      + $scl[1] + " " + $scl[2] + " ]");
}
}

```

В качестве текущего времени восстанавливается предыдущее значение.

```
currentTime -update false $cTime;
```

Наконец, выходной файл закрывается. Заканчивая работу с файлом, очень важно закрыть его явным образом.

```
fclose $fileHnd;
```

```
}
```

Команда fopen применяется для открытия файла как **двоичного**. Открыть файл в текстовом режиме при помощи этой команды нельзя. Об этом важно помнить, пытаясь прочитать файл в другом приложении.

Сценарий scaleSkeleton

Весьма распространенной задачей является **адаптация** готового скелета для представления более мелких или более крупных персонажей. Простое масштабирование скелета часто не дает желаемого **результата**. Проблема состоит в том, что эта операция **увеличивает** или уменьшает скелет относительно начала отсчета корневого **сустава**. В действительности же все суставы должны увеличиваться или уменьшаться в направлении соответствующих костей. На рис. 3.6 показан результат применения к исходному скелету простой операции масштабирования (`scale 0.5 0.5 0.5`). Пропорции, очевидно, не сохранились. Вместе с тем применение сценария `scaleSkeleton` позволяет получить ожидаемый результат масштабирования.

Сценарий `scaleSkeleton` на удивление **невелик**.

```

proc scaleSkeleton( string $rootNode, float $scale )
{
    string $childs[] = `listRelatives -fullPath
                        -type joint -allDescendents $rootNode`;
    for( $child in $childs )
    {
        float $pos[] = "joint -query -relative -position $child";

```

```

$pos[0] *= $scale;
$pos[1] *= $scale;
$pos[2] *= $scale;
joint -edit -relative -position $pos[0] $pos[1] $pos[2] $child;
}
]

```

Сначала следует описание процедуры scaleSkeleton, которая принимает на вход корневой узел скелета (`$rootNode`) и масштабный коэффициент (`$scale`).
proc scaleSkeleton(string \$rootNode, float \$scale)

{

Затем по команде `listRelatives` создается полный список всех узлов-суставов, лежащих ниже корневого узла. Список включает не только прямых потомков, но и всех остальных.

```
string $childs[] = `listRelatives -fullPath
                    -type joint -allDescendents $rootNode`;
```

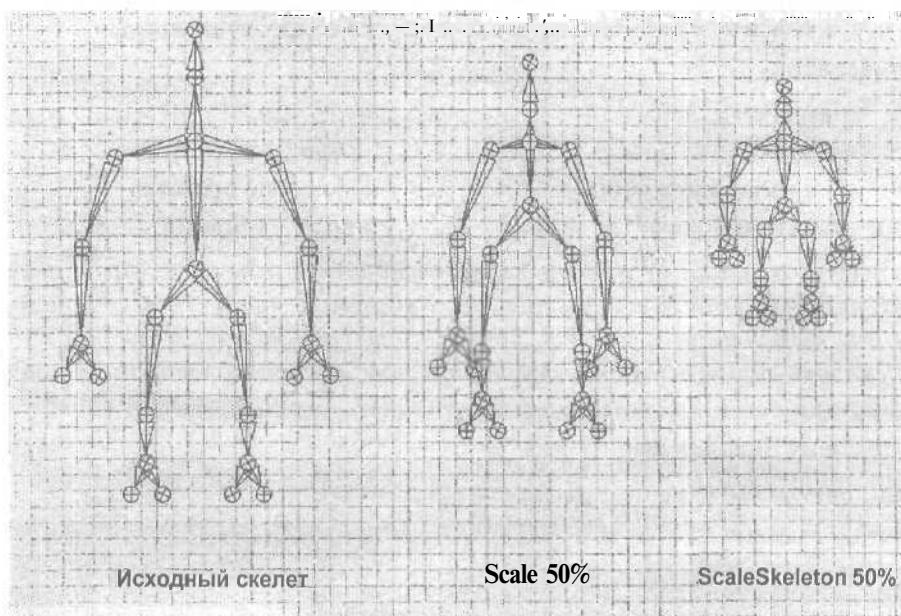


Рис. 3.6. Неправильное и правильное масштабирование

Очень важно использовать полный путь к дочерним узлам. По причине того что некоторые потомки имеют одно и то же имя, а значит являются неуникальными, применение подобных имен в операторах стало бы причиной проблем. На самом деле Maya выдаст сообщение о том, что имя узла неуникально. Однако полные пути к узлам (куда входят имена предков) не повторяются, поэтому используются именно они.

Далее организуется цикл по всем потомкам.

```
for( $child in $childs )  
{
```

Запрашивается текущее положение сустава, рассчитанное относительно его родителя.

```
float $pos[] = `joint -query -relative -position $child`;
```

Затем позиция \$pos масштабируется с коэффициентом \$scale. Позиция – это вектор, отложенный из центра родителя в центр данного сустава. Таким образом, он указывает в том же направлении, что и кость, соединяющая оба сустава. Посредством масштабирования этого вектора вы фактически перемещаете центр дочернего сустава ближе к родительскому суставу либо дальше от него.

```
$pos[0] *= $scale;  
$pos[1] *= $scale;  
$pos[2] *= $scale;
```

Наконец, устанавливается новое относительное положение сустава.

```
joint -edit -relative -position $pos[0] $pos[1] $pos[2] $child;
```

При работе с суставом его центр является простым объектом, поскольку положение узла относительно собственного родителя является всего лишь значением его трансляции.

Чтобы увидеть пример работы процедуры над скелетом, выполните следующие действия.

- 1 Откройте сцену **Skeleton.ma**.
- 2 Опишите приведенную выше процедуру scaleSkeleton.
- 3 Выполните команду

```
scaleSkeleton( "rootJoint", 0.5 );
```

Масштаб скелета пропорционально изменится на 50%.

Сценарий *copySkeletonMotion*

После того как вы **анимировали** одного из персонажей, часто возникает необходимость перенести эту анимацию на другого. Коль скоро оба персонажа имеют одну и ту же скелетную структуру, процесс передачи анимации от одного персонажа к другому сравнительно прост.

```
proc copySkeletonMotion( string $srcRootNode, string $destRootNode )
{
    float $srcPos[] = `xform -query -worldSpace -translation $srcRootNode`;
    float $destPos[] = `xform -query -worldSpace -translation $destRootNode`;

    string $srcNodes[] = `listRelatives -fullPath
                           -allDescendents $srcRootNode`;
    $srcNodes[ size($srcNodes) ] = $srcRootNode;

    string $destNodes[] = `listRelatives -fullPath
                           -allDescendents $destRootNode`;
    $destNodes[ size($destNodes) ] = $destRootNode;

    if ( size($srcNodes) != size($destNodes) )
    {
        error "Source skeleton and destination skeleton are
               structurally different";
        // "Исходный и целевой скелеты имеют различную структуру"
        return;
    }

    string $attrs[] = { "translateX", "translateY", "translateZ",
                        "scaleX", "scaleY", "scaleZ",
                        "rotateX", "rotateY", "rotateZ" };

    int $i;
    for( $i=0; $i < size($srcNodes); $i++ )
    {
        for( $attr in $attrs )
```

```
<
string $inPlugs[] = `listConnections -plugs yes
                      -destination yes
                      ($srcNodes[$i] + "." + $attr)`;

if( size($inPlugs) )
{
    string $tokens[];
    tokenize $inPlugs[0] ":" $tokens;
    string $lnNode = $tokens[0];
    string $inAttr = $tokens[1];

    string $dupInNodes[] = `duplicate -upstreamNodes $inNode`;
    connectAttr -force
                  ($dupInNodes[0] + ":" + $inAttr)
                  ($destNodes[$i] + ":" + $attr);
}
else
{
    $res = `getAttr C$srcNodes[$i] + "." + $attr`;
    setAttr ($destNodes[$i] + ":" + $attr) $res;
}
}

string $moveRoot;
string $parentNodes[] = `listRelatives -parent $destRootNode`;
string $found = "match \"_moveSkeleton\" $parentNodes[0]`;
if( size($found) )
    $moveRoot = $parentNodes[0];
else
    $moveRoot = `group -name "_moveSkeleton" -world $destRootNode`;

move -worldSpace
      ($destPos[0] - $srcPos[0])
```

```
    ($destPos[1] - $srcPos[1])
    ($destPos[2] - $srcPos[2]) $moveRoot;
}
```

Прежде чем перейти к пояснению отдельных фрагментов сценария, посмотрим на результат его работы.

1. Откройте сцену **SkeletonMotion.ma**.

Сцена содержит два скелета. Корневые узлы скелетов именуются, соответственно, **leadDancer** и **copyDancer**.

2. Щелкните по кнопке **Play**.

Скелет **leadDancer** обладает некоторыми базовыми элементами анимации.

3. В редакторе **Script Editor** введите приведенный выше сценарий **copySkeletonMotion**, а затем выполните его.

4. Выполните следующую команду:

```
copySkeletonMotion( "leadDancer", "copyDancer" );
```

5. Щелкните по кнопке **Play**.

Теперь скелет **copyDancer** анимирован так же, как и **leadDancer**. Обратите внимание на то, что **copyDancer** имеет нового родителя **_moveSkeleton**. Параметры трансляции **leadDancer** в точности скопированы в **copyDancer**, из чего следует, что скелеты будут занимать одинаковое положение. Однако хотелось бы, чтобы скелет **copyDancer** двигался так же, как и **leadDancer**, но на некотором расстоянии от исходного положения последнего. Для этого создан новый узел преобразования **_moveSkeleton**, отнесенный на величину разности начального положения обоих скелетов. Скелет **copyDancer** обладает точной копией параметров трансляции **leadDancer**, однако новый родительский узел **_moveSkeleton** удерживает его относительно исходной позиции оригинала.

Описание процедуры **copySkeletonMotion** предполагает, что она принимает на вход корневые узлы исходного (**\$srcRootNode**) и целевого скелета (**\$destRootNode**). Анимация, заложенная в иерархию, берущую начало в исходном узле, копируется в иерархию, берущую начало в целевом узле.

```
proc copySkeletonMotion( string $srcRootNode, string $destRootNode )
{
```

Далее определяется положение обоих корневых узлов в мировом пространстве. Затем эта информация будет использоваться для переноса нового родительского **узла** преобразования. **Заметьте**, что вычисление начального положения

относится к текущему моменту времени. Если вы хотите, чтобы начальное положение соответствовало абсолютному времени, воспользуйтесь сначала командой `currentTime` для перехода к этому моменту.

```
float $srcPos[] = `xform -query -worldSpace -translation $srcRootNode`;
float $destPos[] = `xform -query -worldSpace -translation $destRootNode`;
```

Запрашивается полный список всех дочерних суставных узлов исходного скелета.

```
string $srcNodes[] = `listRelatives -fullPath
    -children -type joint
    -allDescendents $srcRootNode`;
```

К списку также прибавляется корневой узел. Теперь список содержит все исходные суставные узлы исходного скелета.

```
$srcNodes[ size($srcNodes) ] = $srcRootNode;
```

Запрашивается полный список всех дочерних суставных узлов целевого скелета.

```
string $destNodes[] = `listRelatives -fullPath
    -children -type joint
    -allDescendents $destRootNode`;
```

К списку добавляется корневой узел.

```
$destNodes[ size($destNodes) ] = $destRootNode;
```

Если размеры обоих массивов различны, возникает проблема. Это признак того, что иерархии скелетов отличаются друг от друга. В таком случае после вывода сообщения об ошибке процедура завершает свою работу. Обратите внимание на то, что эта проверка не является исчерпывающим тестом структурной идентичности обоих скелетов. При проектировании своей собственной процедуры выполните более тщательную проверку.

```
if( size($srcNodes) != size($destNodes) )
{
    error "Source skeleton and destination skeleton are structurally
different";
    return;
}
```

Далее создайте список всех анимированных параметров, которые вы намерены скопировать. При желании скопировать и другие атрибуты просто пополните ими этот список.

```
string $attrs[] = { "translateX", "translateY", "translateZ",
                    "scaleX", "scaleY", "scaleZ",
                    "rotateX", "rotateY", "rotateZ" };
```

Организуйте цикл по всем исходным суставным узлам.

```
for< $i=0; $i < size($srcNodes); $i++ )  
    i
```

Для каждого суставного узла реализуйте цикл по его атрибутам.

```
for( $attr in $attrs )  
{
```

Получите входящее соединение текущего атрибута.

```
string $inPlugs[] = `listConnections -plugs yes  
                      -destination yes  
                      ($srcNodes[$i] + "." + $attr)`;
```

Если входящее соединение существует, данный атрибут управляет другим узлом. В большинстве случаев, хотя и не всегда, им является кривая анимации, выход которой передается этому атрибуту.

```
If( size($inPlugs) )  
{
```

Так как соединение существует, возьмите это подключение и разбейте его на части. Подключение имеет следующий вид:

<имя_узла>. <имя_атрибута>

Например, `sphere.translateX`. Вы хотите разделить подключение на имя узла (`$inNode`) и имя атрибута (`$inAttr`). Для этого служит команда `tokenize`.

```
string $tokens[];  
tokenize $inPlugs[0] ":" $tokens;  
string $inNode = $tokens[0];  
string $inAttr = $tokens[1];
```

Определив входящий узел, создайте копию его и всех его восходящих соединений. Это позволит полностью продублировать все узлы, прямо или косвенно передающие информацию на вход атрибута.

```
string $dupInNodes[] = `duplicate -upstreamNodes $inNode`;
```

Соедините выходной атрибут нового узла-копии с атрибутом целевого узла.

```
connectAttr -force
    ($dupInNodes[0] + ".." + $inAttr)
    ($destNodes[$i] + ".." + $attr);
```

Если вы хотите, чтобы оба скелета *всегда* были анимированы одинаковым образом, то не делайте копий, а присоедините подключение непосредственно к атрибуту целевого узла. Этого можно достичь следующим образом:

```
connectAttr -force
    ($srcNodes[$i] + ".." + $inAttr)
    ($destNodes[$i] + ".." + $attr);
```

Если атрибут не имеет входных соединений, он не должен анимироваться или управляться извне. В таком случае просто скопируйте значения атрибутов из исходного узла в целевой.

```
else
{
    $res = "getAttr ($srcNodes[$i] + ".." + $attr)";
    setAttr ($destNodes[$i] + ".." + $attr) $res;
}
}
```

Теперь целевой скелет обладает точной копией анимации исходного, а положение первого в точности соответствует положению *второго*. Хотелось бы переместить целевой скелет в начальное положение, добавив для этого предка *_moveSkeleton*. Имя этого родительского узла *transform* будет храниться в переменной *\$moveRoot*.

```
string $moveRoot;
```

Определим список родителей целевого корневого узла.

```
string $parentNodes[] = `listRelatives -parent $destRootNode`;
```

Выясним, является ли его родитель одним из узлов *_moveSkeleton*, который мог быть создан этой процедурой.

```
string $found = "match \"_moveSkeleton\" $parentNodes[0]`;
```

Если это узел *_moveSkeleton*, присвоим строке *\$moveRoot* имя текущего родителя.

```
if( size($found) )
    $moveRoot = $parentNodes[0];
```

```
Если узла _moveSkeleton не существует, создадим его.  
else  
    $moveRoot = `group -name "_moveSkeleton" -world $destRootNode`;  
  
Наконец, узел _moveSkeleton занимает начальное положение целевого кор-  
невого узла.  
move -worldSpace  
    ($destPos[0] - $srcPos[0])  
    C$destPos[1] - $srcPos[1])  
    C$destPos[2] - $srcPos[2]) $moveRoot;  
}
```

3.5.5. Траектории движения

Создать кривую, по которой двигается объект, зачастую проще, чем задать ряд ключевых кадров для каждой позиции или поворота. Кривая описывает путь следования объекта. Построив на поверхности ту или иную кривую, вы можете с легкостью задать анимацию, управляющую движением объекта по этой поверхности. Более того, при движении вдоль кривой объект может выполнять виражи.

1. Откройте сцену **MotionPathCone.ma**.

Сцена состоит из конуса и кривой. Хотелось бы преобразовать сцену так, чтобы конус двигался именно по этой кривой.

2. Выполните следующую команду:

```
string $mpNode = `pathAnimation -curve myCurve myCone`;  
// Result: motionPath1 //
```

Конус переместится к началу кривой. Для этого Maya создаст множество разнообразных узлов зависимости.

Знать все подробности организации этих узлов нет никакой необходимости, однако важно понимать общий характер происходящего. Коль скоро анимация по траектории движения управляет положением объекта **myCone**, атрибуты **translateX**, **translateY** и **translateZ** контролируются узлом **motionPath**. Диаграмма **Dependency Graph** представлена на рис. 3.7.

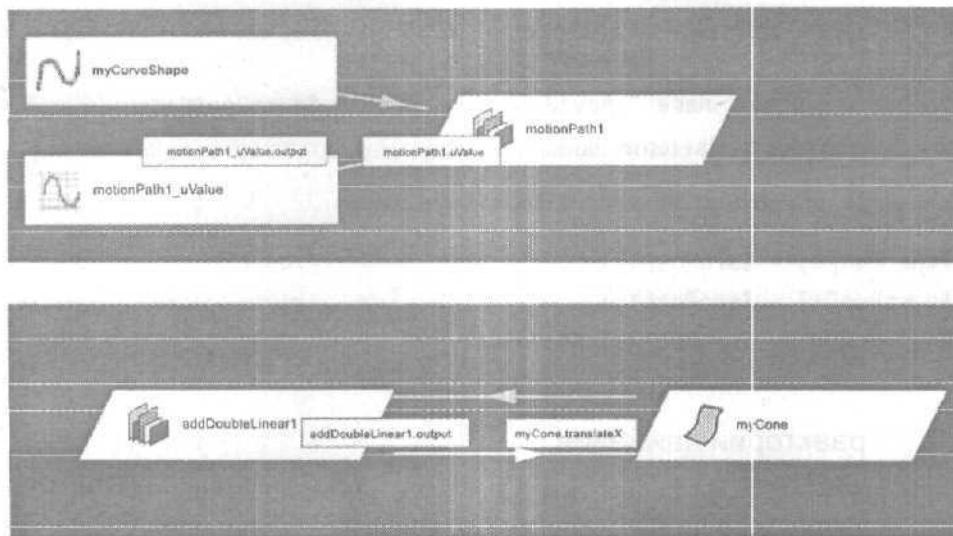


Рис. 3.7. Граф зависимости

Первый слева узел **myCurveShape** представляет собой *my* самую кривую, по которой двигается объект. **motionPath1_uValue** – это узел анимационной кривой, предназначенный для анимации параметра *u*. Каждый из этих узлов поступает на вход **motionPath1**. Данный узел отвечает за вычисление положения и поворота объекта в заданной параметром (*u*) точке кривой. На диаграмме видно, что выход узла **motionPath1**, окончательное положение по оси *x*, передается другому узлу **addDoubleLinear1**, который затем, наконец, пересыпает его атрибуту **translateX** узла **myCone**. Аналогичный поток данных возникает и для атрибутов **translateY** и **translateZ** того же узла **myCone**. Узел **addDoubleLinear** – это простой узел, который принимает на вход два числа и складывает их друг с другом. Результат сложения помещается в атрибут **output**. В данном примере узел **addDoubleLinear** принимает только одно значение, поэтому оно непосредственно отправляется на его выход.

По истечении диапазона анимации хотелось бы установить объект в конце анимационной кривой. С этой целью нужно создать еще один **ключ** для параметра *u*. Для этого нам, во-первых, потребуется имя **узла** траектории движения. К счастью, имя созданного узла возвращается командой **pathAnimation**. Оно хранится в переменной **\$mpNode**.

Во-вторых, необходимо знать параметрический **диапазон** кривой, т. е. начальное и конечное значение параметра кривой в первой и последней управ-

ляющей точке. Для определения максимальной величины *u* на кривой выполните команду

```
float $maxUValue = `getAttr myCurveShape.mixMaxValue maxValue`;
```

Наконец, необходимо знать конечное время диапазона анимации.

```
float $endTime = `playbackOptions -query -animationEndTime`;
```

Теперь, располагая этими тремя значениями, вы можете создать ключевой кадр.

```
setKeyframe -time $endTime -value $maxUValue -attribute uValue  
$mpNode;
```

По этой команде будет организован ключ для атрибута *uValue* узла траектории движения (*\$mpNode*). Ключ будет создан в конце диапазона анимации (*\$endTime*), а его значение приравнено к максимально возможному значению (*\$maxUValue*) параметра кривой.

3. Щелкните по кнопке Play.

Конус начнет свое движение вдоль кривой. Помимо прочего, вы можете задать начальное и конечное время траектории, для чего вам пригодится следующий оператор:

```
pathAnimation -startTimeU 0 -endTimeU 48 -curve myCurve myCone;
```

Тем не менее, предыдущий метод демонстрирует создание ключевых кадров для любого момента времени. А теперь мы хотим заставить конус отслеживать направление траектории.

4. Выполните следующую команду:

```
pathAnimation -edit -follow yes myCone;
```

5. Щелкните по кнопке Play.

Теперь при перемещении конуса его вершина указывает в направлении траектории. Помимо этого, хотелось бы сделать так, чтобы при движении по кривой конус выполнял виражи. Когда объект закладывает виражи, он делает повороты, отклоняясь от своего курса.

6. Выполните следующую команду:

```
pathAnimation -edit -bank yes myCone;
```

Воспроизведя эту анимацию, вы не поймете, глядя на конус, насколько велики его отклонения. Чтобы лучше рассмотреть его виражи, выберите другую ось, которая указывала бы вперед. Выполните следующую команду:

```
pathAnimation -edit -followAxis x myCone;
```

Ось *x* теперь указывает вперед, тем самым заставляя вершину конуса «глядеть» наружу. Влияние виражей просматривается на анимации лучше. Чтобы уменьшить амплитуду виражей, выполните следующую команду:

```
pathAnimation -edit -bankScale 0.5 myCone;
```

Теперь на виражах конус «ныряет» не так глубоко. Переключитесь в прежний режим, когда вершина конуса была обращена вперед, выполнив команду

```
pathAnimation -edit -followAxis y myCone;
```

Чтобы изменить форму конуса ради его плавного скольжения по траектории, выполните следующую команду;

```
flow myCone;
```

7. Щелкните по кнопке **Play**.

Теперь конус полностью покрыт сеткой, которая деформирует его во время движения вдоль кривой.

3.6. Графический интерфейс пользователя

Язык **MEL** можно использовать для создания широкого набора графических интерфейсов пользователя. Фактически весь интерфейс Maya построен и управляется при помощи команд этого языка. Те же команды доступны и вам при проектировании и построении ваших **собственных**, нестандартных пользовательских интерфейсов. Данный раздел посвящен рассмотрению различных элементов интерфейса **пользователя**, имеющихся в вашем распоряжении, а также вопросам управления такими элементами.

3.6.1. Введение

Вам, вероятно, понадобится создать пользовательский интерфейс к тем функциям, которые реализованы в сценариях на языке **MEL**. В обычных **условиях** разработки построение пользовательских интерфейсов может стать долгой и утомительной задачей. Несмотря на то что пакет Maya не содержит визуальных средств создания и размещения элементов **интерфейса**, в нем действительно предусмотрен обширный набор команд создания и **управления** интерфейсами. Привлекая возможности **MEL**, связанные с непосредственным выполнением команд, вы сможете быстро разрабатывать прототипы интерфейсов пользователя. Как только сценарий интерфейса станет готов, его можно будет выполнять и тут же демонстрировать новый продукт. Затем вы сможете вернуться к своему сцена-

рию и по мере надобности отредактировать размещение интерфейсных элементов управления, а также сами элементы.

Немалый выигрыш от использования MEL для всех нужд создания интерфейсов состоит в том, что вам не придется заботиться о поддержке конкретных оконных и интерфейсных систем в разных операционных системах и на различных платформах. Каждая операционная система обычно имеет собственную систему управления окнами со своим собственным API-интерфейсом и характерными особенностями. Применение MEL для создания элементов интерфейса освобождает вас от написания интерфейсного кода, специфичного для той или иной платформы. К тому же существует гарантия того, что, как только вы написали свой интерфейс на языке MEL, он будет аналогично функционировать и вести себя на различных платформах.

Команды **MEL**, предназначенные для организации пользовательского интерфейса, являются частью *системы расширенных слоев (ELF, Extended Layer Framework)*. MEL содержит большой набор элементов пользовательских интерфейсов, включая окна, кнопки, бегунки и т. д. Следующий сценарий служит для генерации окна, показанного на рис. 3.8.

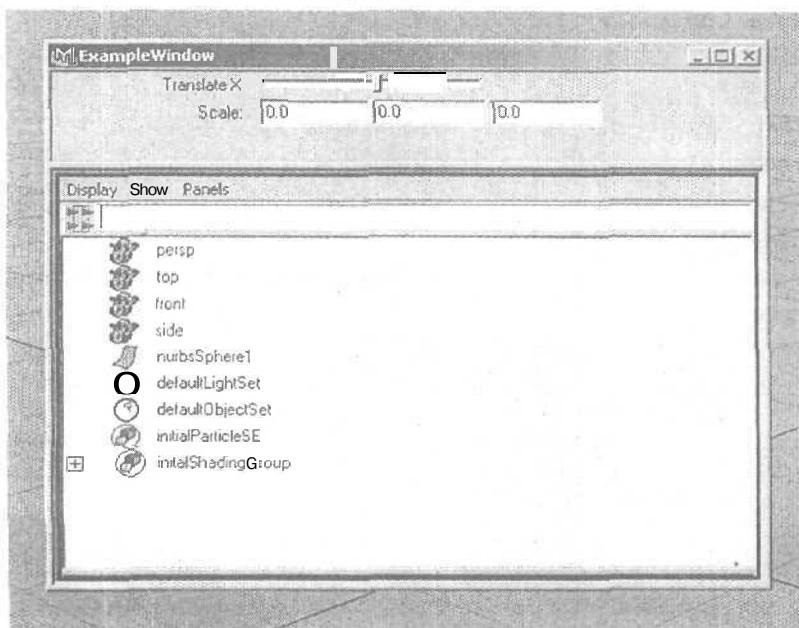


Рис. 3.8. Примерокна

```
window -title "Example Window";
    panelLayout -configuration "horizontal2";
    columnLayout;
        floatSliderGrp -label "Translate X" -min 100 -max 100;
        floatFieldGrp -label "Scale:" -numberOfFields 3 // "Масштаб:"
            setParent ...;
    frameLayout -labelVisible false;
        outlinePanel;
showWindow;
```

Обратите внимание, каким образом столь сложные элементы интерфейса пользователя, как панель схемы сцены, могут легко встраиваться в окна вашей разработки. Ниже мы опишем шаги по созданию простого пользовательского интерфейса.

1. Выберите пункт меню File → New Scene.

2. Щелкните по кнопке Edit Script.

3. Введите следующий сценарий:

```
sphere -name "mySphere";
window myWindow;
columnLayout;
attrFieldSliderGrp -min 0 -max 10 -at "mySphere.sx";
showWindow myWindow;
```

4. Сохраните файл сценария.

5. Щелкните по кнопке Execute Script.

На экране будет построена сфера и выведено окно с заголовком **myWindow**, как показано на рис. 3.9.

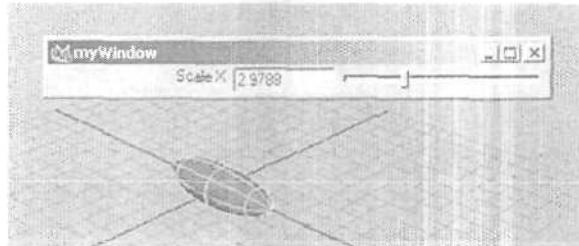


Рис. 3.9. Окно myWindow

6. Передвигните бегунок **Scale X** сначала вперед, потом назад.

Смена положения бегунка вызывает изменение масштаба сферы по оси **x**.

3.6.2. Основные понятия

Для того чтобы понять, как происходит построение интерфейсов, вы предстоит создать несложный интерфейс, а кроме того, узнать некоторые основные понятия из этой области.

1. В редакторе Script Editor наберите следующую команду:

```
window;
```

Результат этого будет таков:

```
// Result; window1
```

Если, как и в случае с большинством команд, вы не укажете явные значения параметров, будут использоваться значения по умолчанию. В данной ситуации Maya построила окно с именем **window1**. По умолчанию оно является невидимым, поэтому вам нужно показать его явно.

2. В редакторе Script Editor выполните следующую команду:

```
window -edit -visible true window1;
```

Теперь окно отображается на экране. Вы можете редактировать свойства интерфейсных элементов, пользуясь флагом **-edit**. В данном случае был установлен параметр **visible**, что позволило сделать окно видимым. Окна наделены множеством различных параметров, в числе которых положение, размер, признак автоматического изменения размера и т. д.

Действия иначе, вы могли бы воспользоваться для отображения окна следующей командой:

```
showWindow window1;
```

Заметьте, что при вызове команды **window** было указано имя окна **window1**. Все элементы пользовательского интерфейса получают имена в момент своего создания. Если вы не присвоите эти имена явным образом, вместо них будут взяты имена по умолчанию. Изменяя элемент интерфейса или ссылаясь на него, вы должны указать его имя, чтобы Maya точно знала о том, к какому элементу вы обращаетесь. Если вы попытаетесь создать новый интерфейсный элемент с тем же именем, которое имеется у существующего элемента, Maya выдаст сообщение об ошибке.

3. Выполните следующую команду:

```
window -edit -widthHeight 200 100 window1;
```

Теперь окно имеет ширину 200 и высоту 100. Каждый из параметров элемента интерфейса может стать объектом запроса.

4. Выполните следующую команду:

```
window -query -numberOfMenus window1;
```

Результат:

```
// Result: 0 //
```

5. Закройте окно **window1**.

6. В редакторе **Script Editor** выполните следующий сценарий:

```
window -widthHeight 200 300 myWindow;  
columnLayout myLayout;  
button -label "First" myButton;  
showWindow myWindow;
```

На экран будет выведено окно **myWindow**. Оно содержит единственную кнопку с текстом **First** (Первая). Заметьте: каждому элементу интерфейса при его создании было присвоено определенное **имя**. Окно (**window**) имеет имя **myWindow**, элемент **columnLayout** носит название **myLayout**, кнопка (**button**) называется **myButton**. Располагая их конкретными именами, вы сможете обращаться **именно** к этим элементам.

Прежде чем добавить в окно элемент **интерфейса**, **Maya** должна **узнать**, каким образом вы хотите поместить этот элемент в пределах окна. С этой целью в окно внедряется **элемент размещения**. Схема размещения управляет способом расположения всех последующих элементов. Существует большое количество разнообразных вариантов размещения. В данном случае в окно был добавлен элемент **columnLayout**. Эта схема предусматривает размещение всех последующих элементов в одном столбце. Чтобы это увидеть, добавьте к **columnLayout** еще одну кнопку (**button**).

7. Выполните следующую команду:

```
button -label "Second" -parent myWindow|myLayout;
```

В окне появляется кнопка с надписью **Second** (Вторая). Будучи добавлена к элементу **columnLayout**, она автоматически занимает место под кнопкой **First**. По мере добавления в окно элементов интерфейса создается их внутренняя иерархия. Вершина иерархии - само окно **myWindow**. **Ниже** находится элемент размещения **myLayout**. Еще ниже расположены обе кнопки. Если один из элементов лежит в иерархии ниже другого, он считается **потомком** первого. Обе кнопки - потомки элемента **myLayout**. При этом элемент, имеющий потомков, именуют **родителем**. Родителем **myLayout** является окно **myWindow**. Какую угодно иерархию элементов можно создать путем при-

соединения к ней потомков. Те, в свою очередь, могут стать родителями собственных дочерних элементов.

Если вы располагаете иерархией интерфейсных элементов, то для обращения к конкретному элементу должны указывать полный путь, который к нему ведет. Полный путь к элементу размещения имеет вид `myWindow|myLayout`. Заметьте: это та же самая форма записи, что использовалась для указания полного пути к объекту сцены (пути по ОАГ). Вместе с тем обратите внимание на отсутствие пробела между именами элементов.

Окна всегда расположены на вершине иерархии. Как и в случае с объектами сцены, два элемента могут носить одинаковые имена, поскольку полные пути к ним будут различны. Допускается, скажем, наличие двух элементов размещения с одним и тем же локальным именем `myLayout` при условии, что они будут находиться в разных иерархиях. Например, `win1|myLayout` и `win2|myLayout` - два корректных различных элемента размещения.

Добавляя новый элемент, вы должны всякий раз указывать его родителя. Однако со стороны программиста это требует больше работы, связанной с поддержкой записей имен элементов. Ради упрощения задачи добавления элементов в Maya введено понятие *родителя по умолчанию*. Если явный родитель не указан, то взамен будет использован родитель по умолчанию. На предыдущем шаге вы создали окно `myWindow`, а затем элемент `columnLayout` с именем `myLayout`. Родитель `columnLayout` вами указан не был, однако таковым автоматически назначен оконный элемент.

При создании интерфейсного элемента он сразу же становится родителем по умолчанию среди элементов своего собственного типа. Существуют отдельные родители по умолчанию среди окон, элементов размещения, меню и т. д. Родители по умолчанию имеются лишь среди тех элементов, которым на правах потомков могут принадлежать другие элементы. Так как окно может содержать в качестве потомков элементы размещения, меню и т. д., то при создании оно автоматически становится родителем по умолчанию среди окон. Когда вы создадите элемент размещения, он станет потомком родителя по умолчанию, т. е. окна. В дальнейшем новый элемент размещения будет родителем по умолчанию среди элементов размещения. Родителем по умолчанию среди окон по-прежнему останется ранее построенное окно. Добавление элемента меню вслед за элементом размещения ведет к вставке меню не в элемент размещения, а в последнее созданное окно. По сути, существует большое разнообразие родителей по умолчанию, используемых в зависимости от типа создаваемого элемента.

Явно установить родителя по умолчанию среди интерфейсных элементов данного типа позволяет команда `setParent`.

8. Закройте окно `myWindow`.

9. В редакторе Script Editor выполните следующий сценарий. Отметим, что пробельный отступ не имеет никакого значения, однако он сделан для упрощения понимания того, какой элемент является **родителем**, а какие - **потомками**.

```
window -widthHeight 200 300 myWindow;  
columnLayout;  
button -label "First";  
button -label "Second";  
rowLayout -number Of Columns 2;  
button -label "Third"; // "Третья"  
button -label "Fourth"; // "Четвертая"  
setParent ...;  
button -label "Fifth"; // "Пятая"  
showWindow myWindow;
```

На экране будет показано окно с пятью кнопками. Первая, вторая и пятая кнопки объединены в **столбец**, а третья и четвертая образуют строку. Вначале строится окно. Оно является самым последним из ранее созданных, а потому автоматически становится родителем по умолчанию среди окон. Следом добавляется элемент `columnLayout`. В отсутствие явно указанных предков элемент располагается на один уровень ниже **текущего** родителя по умолчанию среди **окон**, т. е. `myWindow`. В свою очередь, `columnLayout` становится родителем по умолчанию среди элементов размещения. В него автоматически добавляются следующие две кнопки. Далее **добавляется** элемент `rowLayout`. Поскольку одни элементы размещения могут содержать другие такие же элементы как дочерние, `rowLayout` как потомок **автоматически** размещается на уровень ниже `columnLayout`. Теперь новым родителем по умолчанию среди **элементов** размещения становится `rowLayout`. В него добавляются следующие две кнопки.

Вставку последней кнопки предваряет вызов команды `setParent` с аргументом `...`. Вместо текущего родителя по умолчанию в этом качестве теперь будет выступать его непосредственный предок. Фактически происходит перемещение по иерархии на один элемент вверх. Так как родителем `rowLayout` является `columnLayout`, он и становится новым родителем по умолчанию **сре-**

ди элементов размещения. При добавлении пятой кнопки она добавляется именно в `columnLayout`, но никак не в `rowLayout`.

Подчас бывает нелегко отследить то, какой элемент служит родителем по умолчанию. Добавление элемента к «неправильному» родителю - очень распространенная ошибка. Лучший способ ее избежать - делать отступ в командах, что позволит вам быстро увидеть, какой элемент является нужным родителем. Вместе с тем даже при таком оформлении текста вы не застрахованы от использования неверного родительского элемента.

Если ваш интерфейс имеет иной вид, нежели вы предполагали, убедитесь в том, что вы добавляете элементы к тому родителю, который вам нужен. Чтобы в любой момент времени определить, какой элемент является текущим родителем, используйте следующую команду:

```
setParent -query;
```

Она возвращает полный путь к текущему родительскому элементу.

Когда элементы интерфейса будут размещены, вам захочется их перехватить для выполнения тех или иных полезных действий при взаимодействии с пользователем. Большинство элементов дают возможность указать ряд операторов `MEL`, которые будут выполнены при активизации или изменении пользователем интерфейсного элемента.

10. В редакторе Script Editor выполните следующее:

```
window;  
columnLayout;  
button -label "Click me" -command "sphere";  
showWindow;
```

При этом будет построено окно с единственной кнопкой **Click me** (Щелкните здесь). Как только вы щелкнете по этой кнопке, на экране появится сфера. Хотя данный пример содержит только одну команду `sphere`, вы вправе записать любое количество команд и операторов языка `MEL`.

Некоторые элементы интерфейса выполняют операторы `MEL` при срабатывании того или иного события. Элемент `textField` создан для ввода произвольного пользовательского текста. Он может выполнять операторы `MEL` при получении фокуса, изменении текста пользователем и при нажатии клавиши **Enter**.

11. В редакторе **Script Editor** выполните следующее:

```
global proc myText( string $txt )
{
    print ( "\nThe text entered was " + $txt); // "Введен текст"
}

window;
columnLayout -adjustableColumn true;
textField -text "Change this text" -enterCommand "myText( \"#1\" )";
// "Измените этот текст"
showWindow;
```

На экране будет создано окно с текстовым полем.

12. Измените текст в текстовом поле на любой другой, после чего нажмите клавишу **Enter**.

Script Editor отобразит тот текст, который вы только что ввели. По нажатию **Enter** выполняются операторы **MEL**, перечисленные в параметре **-enterCommand**. Обратите внимание на использование в операторе конструкции **#1**. Текст, содержащийся в элементе **textField**, можно запросить при помощи оператора **textField -query -text <имя элемента textField>**

#1 - это краткая форма записи текущего значения элемента. В данном случае она позволяет получить нужный текст. Будь элемент **флажком**, вы узнали бы, установлен он или сброшен. Чтобы определить значение сложных, составных элементов, просто используйте в командных операторах конструкции **#2, tt3** и т. д.

3.6.3. Окна

Окно представляет собой основной элемент интерфейса и чаще всего служит для объединения широкого спектра других элементов. Оно может содержать строку заголовка, а также кнопки разворачивания и свертывания. Кроме того, окно может размещаться в некоторой экранной области и способно менять свой размер. По умолчанию оно **невидимо**. Причина этого состоит в том, что если вы создадите пустое окно, выведите его на экран, а затем начнете добавлять в него элементы, то ему придется вычислять положение каждого элемента и решать другие задачи, после чего прорисовывать элементы в **режиме** диалога с пользователем. Если же вы создадите невидимое окно и будете добавлять элементы в него,

такое окно отобразится гораздо быстрее, поскольку ему не потребуется выводить все элементы в интерактивном режиме по мере их добавления.

Теперь рассмотрим некоторые из свойств окна более подробно.

1. Выберите пункт меню **File | New Scene**.

2. Щелкните по кнопке **Edit Script**.

3. Введите следующий сценарий:

```
global proc showMyWindow()
{
    window myWindow;
    showWindow myWindow;
}
```



```
showMyWindow();
```

4. Сохраните файл сценария.

5. Щелкните по кнопке **Execute Script**.

Окно **myWindow** будет создано и выведено на экран.

6. Щелкните по кнопке **Execute Script** еще раз.

Появится следующая ошибка:

```
// Error: file: .../scripts/learning.mel line 3:
    Object's name is not unique: myWindow //
// Ошибка: файл: .../scripts/learning.mel строка 3:
    Имя объекта не является уникальным: myWindow //
```

Поскольку окно **myWindow** уже существует, вам не удастся создать еще одно окно с тем же именем. Этого сообщения об ошибке хотелось бы избежать.

7. Введите следующий сценарий, а затем выполните его, как и прежде.

```
global proc showMyWindow()
{
    if( ! `window -exists myWindow* )
        window myWindow;
    showWindow myWindow;
}

showMyWindow();
```

Сообщение об ошибке больше не возникает. Созданию окна предшествует проверка его наличия. Для этого служит команда window с флагом -exists. Если окна нет, оно строится.

Теперь хотелось бы приступить к размещению в окне иных элементов интерфейса. Лучше всего это делать сразу же после создания окна, но прежде чем вы сделаете его видимым. Однако ваше окно уже на экране, так что подобное невозможно. До тех пор пока разработка оконного интерфейса не завершена, всякий раз хотелось бы начинать заново.

8. Введите следующий сценарий, а затем выполните его.

```
global proc showMyWindow()  
{  
    // Присвойте $developing значение false, когда сценарий будет готов  
    int $developing = true;  
    if( $developing && `window -exists myWindow` )  
        deleteUI myWindow;  
    if( ! `window -exists myWindow` )  
        window myWindow;  
    showWindow myWindow;  
}  
  
showMyWindow();
```

Если переменная \$developing имеет истинное значение и при этом окно существует, оно будет удалено. Это позволяет создавать новое окно при каждом запуске сценария.

9. Введите следующий сценарий, а затем выполните его.

```
global proc showMyWindow()  
{  
    // Присвойте $developing значение false, когда сценарий будет готов  
    int $developing = true;  
    if( $developing && "window -exists myWindow" )  
        deleteUI myWindow;  
    if( ! `window -exists myWindow` )  
    {
```

```
window -title "My Window" -resizeToFitChildren true myWindow;
columnLayout -adjustableColumn true;
button -label "Change Name";
}
showWindow myWindow;
}

showMyWindow();
```

На экране появится окно с кнопкой, которая содержит метку **Change Name** (Смените имя). При создании окна параметр **-resizeToFitChildren** принимает истинное значение. Следовательно, окно будет автоматически изменять собственные размеры и «подгонять» их под размеры своих потомков. Параметр **-adjustableColumn** элемента **columnLayout** также установлен в значение **true**, чтобы размер кнопки по горизонтали соответствовал ширине элемента размещения.

К сожалению, окну не удалось нормально изменить свой размер. Причиной этого стало то, что при закрытии окна Maya по умолчанию запоминает его имя, позицию и размер. Если затем вы откроете это окно еще раз, его предыдущее положение и размер восстановятся. Если же вы хотите, чтобы окно имело иной размер и занимало иное положение, их надо задать непосредственно, при помощи одной или нескольких следующих команд. Замените 999 на соответствующее значение.

```
window -width 999 windowName;
window -height 999 windowName;
window -widthHeight 999 999 windowName;
window -topEdge 999 windowName;
window -leftEdge 999 windowName;
window -topLeftCorner 999 999 windowName;
```

Кроме того, вы можете удалить информацию о размере и положении данного окна, хранящуюся в системе Maya. Для этого служит команда **windowPref**. Если же вы хотите, чтобы Maya перестала запоминать размер и положение каждого окна, снимите флажок **Remember Size and Position** (Запоминать размер и положение) в разделе **Preferences | Interface | Windows** (Параметры { Интерфейс | Окна}).

10. Введите следующий сценарий, а затем выполните его.

```
global proc showMyWindow()
{
    // Присвойте $developing значение false, когда сценарий будет готов
    int $developing = true;
    if( $developing && `window -exists myWindow` )
        deleteUI myWindow;
    if( ! `window -exists myWindow` )
    {
        if( $developing )
            windowPref -remove myWindow;
        window -title "My Window" -resizeToFitChildren true myWindow;
        columnLayout -adjustableColumn true;
        button -label "Change Name";
    }
    showWindow myWindow;
}

showMyWindow();
```

Теперь размер окна изменяется в соответствии с размером columnLayout. Как только переменная \$developing примет ложное значение, пользователь сможет задать размер и положение окна, которые останутся в памяти Maya, поскольку вызова команды windowPref -remove myWindow больше не произойдет никогда.

Далее, хотелось бы показать диалоговое окно в момент щелчка по кнопке **Change Name**.

11. Введите следующий сценарий, а затем выполните его.

```
global proc myChangeName()
{
    string $nam[] = `ls -sl`;
    if( size($nam) > 0 )
    {
        string $res = `promptDialog -message "Enter new name:"`
```

```
-button "OK" -button "Cancel" -defaultButton "OK"
-cancelButton "Cancel" -dismissString "Cancel`";
// "Введите новое имя:"
if( $res == "OK" )
{
    string $newname = `promptDialog -query`;
    eval( "rename " + $nam[0] + " " + $newname );
}
}

global proc showMyWindow()
{
// Присвойте $developing значение false, когда сценарий будет готов
int $developing = true;
if( $developing && 'window -exists myWindow' )
    deleteUI myWindow;
if( ! 'window -exists myWindow' )
{
    if( $developing )
        windowPref -remove myWindow;
    window -title "My Window" -resizeToFitChildren true myWindow;
    columnLayout -adjustableColumn true;
    button -label "Change Name" -command "myChangeName();";
}
showWindow myWindow;
}

showMyWindow();
```

Теперь при нажатии кнопки вызывается процедура `myChangeName`. Эта связь установлена путем задания флага `-command`, значением которого является оператор `MEL`, выполняемый при нажатии этой кнопки. Процедура `myChangeName` на-

ходит выделенные объекты. Если есть хотя бы один подобный объект, она отображает диалоговое окно, требующее от пользователя ввода нового имени. Если пользователь щелкнет по находящейся в диалоговом окне кнопке **OK**, объект будет переименован.

Команда `promptDialog` служит примером модального окна диалога. Такое окно похоже на все прочие окна за исключением того, что оно не позволит работать с любыми другими окнами приложения до тех пор, пока не будет закрыто. Другая очень полезная команда вызова диалогового окна - `confirmDialog`. Она выводит на экран сообщение, которое пользователь должен подтвердить или отклонить. К примеру, следующая команда отображает окно диалога и дает пользователю возможность выбрать вариант Yes (Да) или No (Нет).

```
confirmDialog -message "Do you want to delete the object" -button
"Yes"
-button "No"
// "Вы хотите удалить объект?"
```

3.6.4. Схемы размещения

Существует большое разнообразие схем размещения. Каждая из них служит одной общей цели - задать положение или размеры включаемых в нее элементов интерфейса. Вкладывая элементы разных типов друг в друга, можно создавать весьма сложные схемы размещения.

Каждое окно должно иметь, как минимум, одну схему размещения. Первая расположенная в окне схема полностью занимает его клиентскую область. Так именуют часть окна, куда не входят заголовок, строка меню, границы и т. д. Исключение здесь составляет элемент `menuLayout`, высота которого равна высоте меню. Этот элемент не занимает клиентской части окна в целом.

Если вы работаете с элементом размещения и не знаете, какого он типа, можете воспользоваться командой `layout`. Она позволяет оперировать всеми схемами размещения.

```
string $layoutName = <здесь вставьте имя элемента размещения>
layout -edit -width 200 $layoutName;
layout -query -numberOfChildren $layoutName;
layout -exists $layoutName;
```

Ниже описаны некоторые наиболее широко используемые схемы размещения.

ColumnLayout

Все потомки схемы `columnLayout` размещены в одном столбце, один под другим. Вы можете задать ширину столбца и высоту строки, а также способ при соединения каждого дочернего элемента. В следующем примере столбец имеет фиксированную ширину 150. Расстояние между строками равно 8. При добавлении элемента он растягивается на ширину столбца, но со смещением на 12 единиц с каждой стороны. Окончательный результат показан на рис. 3.10.

window;

```
columnLayout -columnAttach "both" 12 -rowSpacing 8 -columnWidth 150;  
button;  
button;  
showWindow;
```

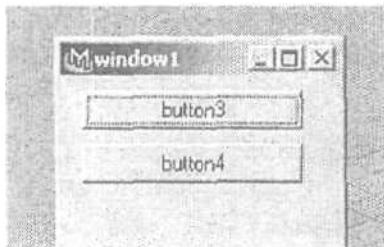


Рис. 3.10. Пример схемы `columnLayout`

Если вы хотите, чтобы дочерние элементы располагались по обеим сторонам и изменяли свой размер при изменении размера элемента `columnLayout`, используйте команду

```
columnLayout -adjustableColumn true;
```

RowLayout

Схема `rowLayout` предполагает размещение всех потомков в виде единой строки. Общее число столбцов в строке должно быть указано. По умолчанию оно равно нулю, поэтому при попытке добавить дочерний элемент вы получите ошибку, аналогичную следующей:

```
// Error: file: .../learning.mel line 4:  
    Too many children in layout: rowLayout1 //  
// Ошибка: файл: .../learning.mel строка 4:  
    Слишком много потомков в элементе размещения: rowLayout1 //
```

Для исправления ситуации просто увеличьте число столбцов при описании `rowLayout`.

Вы можете установить выравнивание, смещение и признак регулировки каждого отдельного столбца. В следующем примере элемент `rowLayout` содержит три столбца. Ширина каждого из них равна, соответственно, 100, 60 и 80. Дочерние элементы первого столбца прикрепляются к обеим сторонам без всякого смещения. Результат показан на рис. 3.11.

`window;`

```
rowLayout -numberOfColumns 3 -columnWidths 100 60 80
          -columnAttach 1 "both" 0;
button;
button;
button;
showWindow;
```

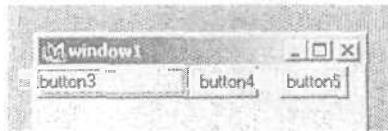


Рис. 3.11. Пример схемы `rowLayout`

GridLayout

Схема `gridLayout` предполагает размещение дочерних элементов в ряд строк и столбцов. По мере добавления потомков `gridLayout` может автоматически выделять для них место, при необходимости увеличивая число строк или число столбцов. Такого поведения схемы можно не допустить, поддерживая постоянное количество тех и других. Кроме того, можно указывать ширину и высоту ячеек сетки.

В следующем примере использован элемент `gridLayout` с двумя строками и столбцами. Каждая отдельная ячейка имеет ширину 30 и высоту 60. На рис. 3.12 показан конечный результат.

`window;`

```
gridLayout -numberOfRowsColumns 2 2 -cellWidthHeight 60 50;
button;
button;
button lastButton;
showWindow;
```

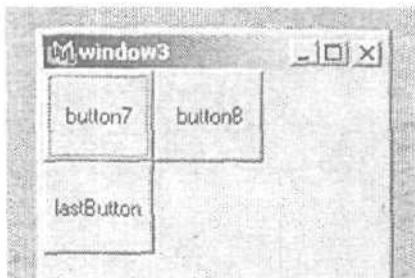


Рис. 3.12. Пример схемы `gridLayout`

Чтобы в дальнейшем изменить положение отдельных элементов сетки, используйте флаг `-position`. Параметрами этого флага являются название дочернего элемента и его новое положение. Новое положение - это индекс, который отсчитывается от 1, начиная с позиции первой строки и первого столбца (1) с переходом по столбцам вниз к следующей строке и т. д. Так, для размещения последней кнопки из предыдущего примера в первой строке второго столбца вы должны использовать команды

```
$buttonName = <здесь поместите имя последней кнопки>;  
gridLayout -position $buttonName 2;
```

FormLayout

Схема `formLayout` - самая гибкая схема размещения, предлагающая великое множество различных вариантов выравнивания и возможностей привязки элементов. Наряду с этим, она предусматривает возможность абсолютного и относительного позиционирования потомков. При добавлении элемента к схеме `formLayout` тот не имеет никакого начального положения по умолчанию. Его позиция должна быть указана позднее. В общем случае процедура состоит, по сути, в добавлении к `formLayout` всех дочерних элементов последней и дальнейшем использовании возможностей редактирования для их окончательной расстановки.

Отдельные края элементов (левый, правый, верхний, нижний) могут выравниваться в соответствии с размерами формы или по краю других элементов, а также «прикрепляться» к заданному положению. В следующем примере на форму помещаются три кнопки, которые выравниваются и привязываются к форме различными способами,

```
window;
string $form = `formLayout -numberOfDivisions 100`;
string $but1 = 'button';
string $but2 = 'button';
string $but3 = 'button';
formLayout -edit
//Кнопка1
-attachForm $but1 "top" 0
-attachForm $but1 "left" 0
-attachForm $but1 "bottom" 0
-attachPosition $but1 "right" 0 50
//Кнопка2
-attachForm $but2 "right" 0
//Кнопка3
-attachPosition $but3 "top" 0 5
-attachControl $but3 "left" 5 $but1
$form;
showWindow;
```

Флаг `-numberOfDivisions` определяет при создании формы число делений, Тем самым задается виртуальная сетка, к которой привязывается положение элемента. Несмотря на то что число делений можно установить произвольно, чаще всего оно приравнивается к 100, что позволяет указывать положение элементов в процентах. Так, выбор значения 50 в качестве позиции по горизонтали аналогичен установке 50% ширины формы.

```
string $form = `formLayout -numberOfDivisions 100`;
```

Далее создаются три кнопки, автоматически добавляемые к текущей схеме размещения. Заметьте, что их имена сохраняются, так как позднее к ним потребуется обратиться.

```
string $but1 = 'button';
string $but2 = 'button';
string $but3 = 'button';
```

Теперь, когда кнопки добавлены к форме, переходите к их расстановке и выравниванию. Для этого служит команда `formLayout` с флагом `-edit`.

```
formLayout -edit
```

Верхняя, левая и нижняя границы первой кнопки выравниваются в соответствии с размерами формы. Правая граница соответствует 50% ее ширины.

```
// Кнопка 1  
-attachForm $but1 "top" 0  
-attachForm $but1 "left" 0  
-attachForm $but1 "bottom" 0  
-attachPosition $but1 "right" 0 50
```

Вторая кнопка просто выравнивается по правому краю формы. Поскольку другие границы кнопки не установлены, она по умолчанию располагается у верхнего края формы и ее ширина равна исходной ширине любой кнопки.

```
// Кнопка 2  
-attachForm $but2 "right" 0
```

Третья кнопка располагается по вертикали на уровне 5% высоты формы. Ее левая граница располагается по правой границе первой кнопки с небольшим отступом, равным 5.

```
// Кнопка 3  
-attachPosition $but3 "top" 0 5  
-attachControl $but3 "left" 5 $but1  
$form;
```

Полученный в результате интерфейс пользователя показан на рис. 3.13. К `formLayout` можно добавлять и иные схемы размещения, в том числе другие элементы `formLayout`. Сочетание одних схем размещения внутри других дает широкие возможности управления точным положением и размером каждого отдельного элемента.

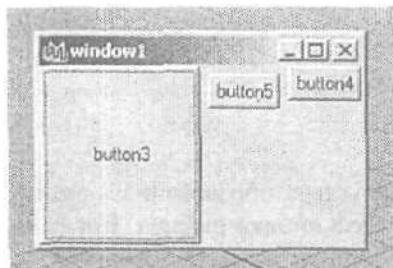


Рис. 3.13. Пример схемы `formLayout`

FrameLayout

Если вам нужно получить интерфейсную область, которая может растягиваться и сжиматься, воспользуйтесь схемой размещения `frameLayout`. Обычно она содержит метку и кнопку разворачивания/свертывания, которая уменьшает размер формы или увеличивает его для отображения каждого дочернего элемента. `frameLayout` обладает возможностями настройки метки, границы, отступов и т. д. Важно заметить, что данная схема может иметь только одного потомка, из чего следует необходимость добавления к `frameLayout` другого элемента размещения для расстановки в нем всех дочерних элементов.

В следующем примере создается схема `frameLayout`. Ее единственным потомком является `columnLayout`. Схема описана как сжимаемая, что позволяет пользоваться кнопкой разворачивания/свертывания. Полученная в результате расстановка `frameLayout` изображена на рис. 3.14.

```
window;
```

```
    frameLayout -label "Settings" -borderStyle "etchedIn"  
        -font "obliqueLabelFont" -collapsible true;  
        columnLayout;  
            button;  
            button;  
            button;  
        showWindow;
```

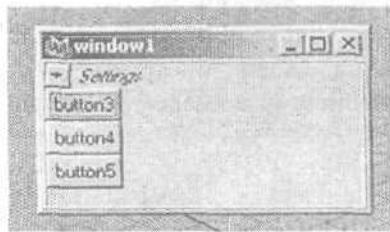


Рис. 3.14. Пример схемы `frameLayout`

TabLayout

Схема `tabLayout` позволяет строить из других схем ряд, образованный вкладками. Верхнюю часть каждой вкладки занимает особая кнопка выбора. Когда та или иная вкладка будет выделена, связанный с ней элемент размещения появится на экране. Подобно `frameLayout`, схема `tabLayout` дает возможность эффективно использовать одну и ту же оконную область, многократно размещая на ней разнообразные элементы.

Все потомки `tabLayout` должны быть элементами размещения. При попытке пополнить схему другим элементом, который отличен от элемента размещения, произойдет ошибка.

В следующем примере будет создан элемент `tabLayout`, а затем к нему добавятся два дополнительных элемента `columnLayout`. Каждый из них содержит простую кнопку. Наконец, будут изменены метки вкладок. Обратите внимание на вызов `setParent ...`, который следует за добавлением кнопки на панель `columnLayout` и необходим для установки `tabLayout` в качестве родителя по умолчанию среди элементов размещения. На рис. 3.15 показан конечный результат.

```
window;  
string $tabs = `tabLayout`;  
string $tab1 = `columnLayout`;  
button;  
setParent ...;  
string $tab2 = `columnLayout`;  
button;  
setParent ...;  
tabLayout -edit  
-tabLabel $tab1 = "Colors"  
-tabLabel $tab2 = "Flavors"  
$tabs;  
showWindow;
```

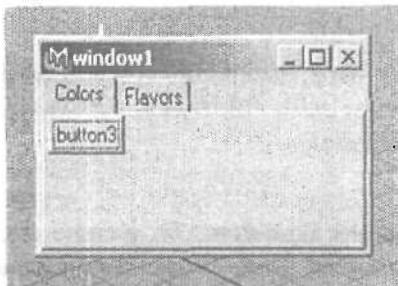


Рис. 3.15. Пример схемы `tabLayout`

ScrollLayout

Если элемент интерфейса слишком велик для его вывода в заданной области, можно воспользоваться схемой размещения **scrollLayout**. Сама по себе она не вносит никакой расстановки элементов, однако отображает полосы прокрутки, предоставляемые дочерней схеме. И действительно, **scrollLayout** всегда содержит еще одну схему с дочерними элементами.

В данном примере в окне размещается схема **scrollLayout**. Далее к ней добавляется **columnLayout** с четырьмя кнопками. Результат показан на рис. 3.16.

```
window;
scrollLayout;
columnLayout;
button;
button;
button;
button;
showWindow;
```

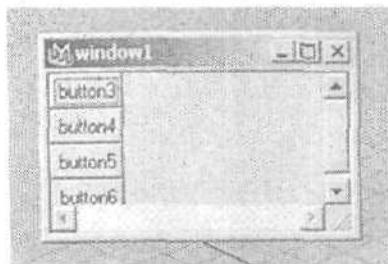


Рис. 3.16. Пример схемы **scrollLayout**

После запуска этого сценария измените размер окна, уменьшив его по вертикали; в таком случае вертикальная полоса прокрутки станет активной и позволит выполнять скроллинг для отображения скрытых кнопок.

MenuBarLayout

Схема **menuBarLayout** предназначена для хранения подменю. Те, в свою очередь, содержат свои собственные отдельные элементы меню (**menuItem**). Схема **menuBarLayout** может, как в следующем примере, служить для создания главного меню окна или независимых меню элементов размещения. Итоговая схема **menuBarLayout** показана на рис. 3.17.

```
window;
menuBarLayout;
    menu -label "File";
        menuItem -label "Exit";
    menu -label "Help" -helpMenu true;
        menuItem -label "About...";
   .setParent ...;
string $tabs = `tabLayout`;
string $tab1 = `menuBarLayout`;
    menu -label "Colors";
        menuItem -label "Red";
        menuItem -label "Green";
    menu -label "Flavors";
        menuItem -label "Vanilla";
        menuItem -label "Chocolate";
tabLayout -edit -tabLabel $tab1 "Confectionary" $tabs;
showWindow;
```

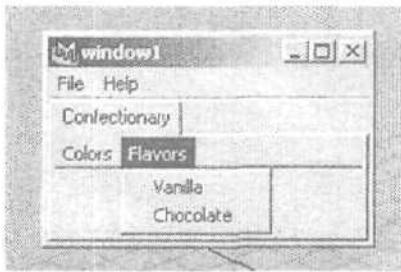


Рис. 3.17. Пример схемы menuBarLayout

Если вы создали окно без меню, после чего добавили другие элементы размещения, то для добавления строки меню просто выполните следующие команды. При этом окно приобретает статус родителя по умолчанию и к нему добавляется элемент menuBarLayout.

```
setParent -topLevel;
menuBarLayout;
// ... добавьте пункты меню
```

3.6.5. Элементы управления

Элементы управления- это отдельные элементы интерфейса, с которыми взаимодействует пользователь. К числу элементов управления относятся кнопки, флажки, бегунки, поля и т. д. Если вы не знаете наверняка тип элемента, с которым работаете, то можете воспользоваться командой control, предназначеннной для редактирования элементов управления и организации запросов к ним.

```
string $controlName = <здесь вставьте имя элемента управления>;
control -edit -width 23 $controlName;
control -query -parent $controlName;
control -exists $controlName;
```

Меню

Вы можете добавить меню к строке главного меню или к элементу размещения menuBarLayout. Следующий пример иллюстрирует создание строки главного меню. Параметру -menuBar во время создания окна присваивается истинное значение. Затем производится вставка меню. Заметим, что это меню может быть разорвано, поскольку его флаг -tearOff имеет значение истины. Меню содержит три дополнительных пункта, в том числе разделитель. Его окончательный вид представлен на рис. 3.18.

```
window -menuBar true;
menu -label "File" -tearOff true;
menuItem -label "New";
menuItem -divider true;
menuItem -label "Exit";
menu -label "Help" -helpMenu true;
menuItem -label "About";
showWindow;
```

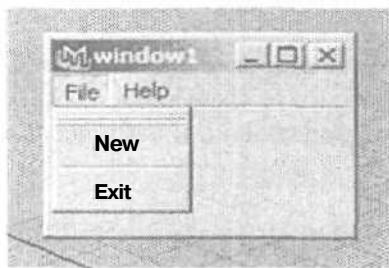


Рис. 3.18. Пример меню

Меню может быть внедрено практически в каждую схему размещения, если элемент `menuBarLayout` был создан первым. В следующем примере показано многообразие разных видов меню. Конечный результат приведен на рис. 3.19.

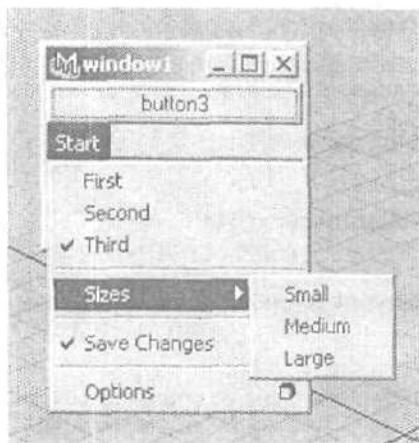


Рис. 3.19. Подробное меню

```
window;  
    columnLayout -adjustableColumn true;  
    button;  
    menuBarLayout;  
        menu -label "Start" -allowOptionBoxes true;  
            // Пункты-переключатели  
            radioMenuItemCollection;  
            menuItem -label "First" -radioButton off;  
            menuItem -label "Second" -radioButton off;  
            menuItem -label "Third" -radioButton on;  
            // Разделитель  
            menuItem -divider true;  
            // Подменю  
            menuItem -subMenu true -label "Sizes";  
            menuItem -label "Small";  
            menuItem -label "Medium";  
            menuItem -label "Large";  
        setParent -menu .;
```

```
// Разделитель
menuItem -divider true;
// Пункт-флажок
menuItem -label "Save Changes" -checkBox on;
// Разделитель
menuItem -divider true;
// Пункт меню со значком выбора
menuItem -label "Options";
menuItem -optionBox true -command "doOptionsBox()";
showWindow;
```

На экране создается окно с элементом `columnLayout` и единственной кнопкой.

```
window;
columnLayout -adjustableColumn true;
button;
```

Так как вы хотите, чтобы меню отображалось внутри элемента размещения и не становилось главным, добавьте `menuBarLayout`.

```
menuBarLayout;
```

Теперь можно добавить само меню. При желании внедрить в него значки выбора убедитесь в том, что параметр `-allowOptionBoxes` установлен в `true`. Иначе внедрить значки выбора вам не удастся.

```
menu -label "Start" -allowOptionBoxes true;
```

Пункты меню, если их добавлять в разделе операторов `radioMenuItemCollection`, могут вести себя подобно кнопочным переключателям. Кроме того, каждый из нижеследующих пунктов должен задать значение флага `-radioButton`. Как и в случае с обычными кнопками-переключателями, в установленном состоянии может находиться только один пункт меню.

```
radioMenuItemCollection;
menuItem -label "First" -radioButton off;
menuItem -label "Second" -radioButton off;
menuItem -label "Third" -radioButton on;
```

Пункт-разделитель рисует в меню строку и служит для визуального отделения элементов друг от друга.

```
menuItem -divider true;
```

Для организации подменю создайте обычный пункт меню (`menuItem`), однако установите его параметр `-subMenu` в значение `true`. Все последующие элементы будут добавлены в это подменю.

```
menuItem -subMenu true -label "Sizes";
menuItem -label "Small";
menuItem -label "Medium";
menuItem -label "Large";
```

Чтобы выйти из подменю, надо перейти к его родителю. Это делается по команде `setParent`.

```
setParent -menu . .;
```

Добавим теперь пункт-флажок.

```
menuItem -label "Save Changes" -checkBox on;
```

В процессе создания элемента меню, одновременно содержащего метку и значок выбора, сначала добавляется пункт меню с меткой. Затем нужно добавить еще один пункт, на сей раз с установленным в истинное значение флагом `-optionBox`. Кроме того, параметр `-command` задан так, что при выделении этого значка вызывается процедура `doOptionsBox`.

```
menuItem -label "Options";
menuItem -optionBox true -command "doOptionsBox()";
```

В ходе добавления пунктов меню вы можете в любое время определить, что представляет собой текущий предок меню, воспользовавшись для этого следующей командой:

```
string $currentMenu = `setParent -query -menu`;
```

При желании задать текущее меню используйте команду

```
setParent -menu $newMenu;
```

Кнопки

Существует большое разнообразие видов кнопок, но все же их **основное** предназначение - обеспечить пользователя элементом управления, по которому тот может щелкнуть, чтобы инициировать некоторое действие. Это действие описывается последовательностью из одного или нескольких операторов языка MEL.

Простейшая кнопка содержит лишь текстовую метку и создается по команде `button`. В следующем примере создается кнопка с меткой (`label`) **Press here!** (Нажмите **здесь!**). По нажатию на эту кнопку выполняется команда (`command`).

Параметр `command` содержит инструкцию вывода текста «Button pressed» («Кнопка нажата»). Получившаяся в результате кнопка показана на рис. 3.20.

```
window;  
    columnLayout;  
        button -label "Press here!"  
            -command "print \"Button pressed.\n\"";  
    showWindow;
```

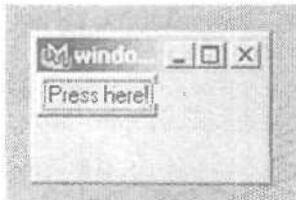


Рис. 3.20. Пример кнопки

Чтобы создать кнопку, на которую нанесена картинка либо пиктограмма, воспользуйтесь командой `symbolButton`. Следующий пример иллюстрирует создание кнопки (`symbolButton`), содержащей пиктограмму `sphere.xpm`. По нажатию кнопки происходит создание сферы. На рис. 3.21 показан конечный результат.

```
window;  
    columnLayout;  
        symbolButton -image "sphere.xpm" -command "sphere;";  
    showWindow;
```



Рис. 3.21. Пример кнопки `symbolButton`

Если вам нужна кнопка, которая содержит и метку, и изображение, используйте команду `iconTextButton`. Фактически вы можете указать, что именно отображается на экране: текст, изображение либо сочетание того и другого, - а также выбрать вариант их взаимного размещения. В следующем примере создается кнопка, содержащая и метку, и пиктограмму. Установка параметра `-style` обеспечивает

отображение метки справа от изображения. Параметр кнопки с именем command содержит оператор MEL "cone", а значит, нажатие кнопки ведет к созданию конуса. Полученная в результате кнопка iconTextButton показана на рис. 3.22.

window;

```
columnLayout -adjustableColumn true;
iconTextButton -style "iconAndTextHorizontal"
    -image1 "cone.xpm" -label "cone"
    -command "cone";
showWindow;
```

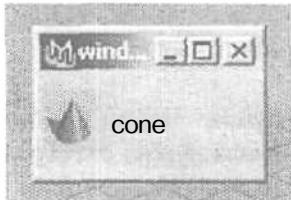


Рис. 3.22. Пример кнопки iconTextButton

Флажки

Флажок напоминает кнопку и отличается от нее только тем, что служит для обозначения включенного или выключеного состояния чего-либо. Щелчок по флашку влечет за собой смену его текущего состояния. Если флашок установлен, он будет сброшен, и наоборот.

В дополнение к команде `checkBox`, которая создает флашок стандартного вида, существует команда `symbolCheckBox`, создающая флашок с пиктограммой, а также команда `iconTextCheckBox`, позволяющая получить комбинацию пиктограммы и метки. В отличие от стандартных флашков, включенное состояние которых отмечается «галочкой», прочие флашки, если они установлены, имеют вдавленное изображение. Следующий пример демонстрирует флашки трех различных типов. Они изображены на рис. 3.23.

window;

```
columnLayout -adjustableColumn true;
checkBox -label "Visibility";
symbolCheckBox -image "circle.xpm";
iconTextCheckBox -style "iconAndTextVertical"
    -image "cube.xpm" -label "cube";
showWindow;
```

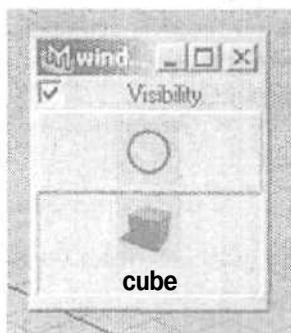


Рис. 3.23. Примеры флагжков

Переключатели

Переключатели позволяют выбирать один элемент из группы. Чтобы знать о том, какие переключатели принадлежат данной группе, сначала нужно создать набор `radioCollection`. Его содержимым являются кнопочные переключатели. Следующий пример демонстрирует создание `radioCollection` и добавление в набор трех стандартных переключателей. Окончательный вид интерфейса представлен на рис. 3.24.

```
window;
```

```
    rowLayout -columnWidth3 60 60 60 -numberOfColumns 3;
```

```
    radioCollection;
```

```
        radioButton -label "Cold";
```

```
        radioButton -label "Warm";
```

```
        radioButton -label "Hot";
```

```
    showWindow;
```

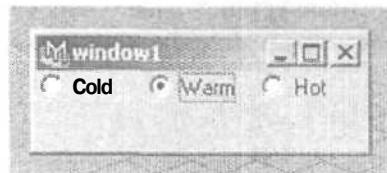


Рис. 3.24. Пример переключателей `radioButton`

Чтобы определить, какой из переключателей выбран в данный момент, используйте команду

```
radioCollection -query -select $radioCollectionName;
```

Каждый переключатель может также содержать пиктограмму. Это делается при помощи команды `iconTextRadioButton` в сочетании с командой `iconTextRadioCollection`. В следующем примере показано, как создать три переключателя с пиктограммами. На рис. 3.25 изображен окончательный результат,

```
window;
    rowLayout -columnWidth3 60 60 60 -numberOfColumns 3;
        iconTextRadioCollection;
            iconTextRadioButton -image1 "sphere.xpm" -label "Cold";
            iconTextRadioButton -image1 "cone.xpm" -label "Warm";
            iconTextRadioButton -image1 "torus.xpm" -label "Hot";
    showWindow;
```



Рис. 3.25. Пример переключателей `iconTextRadioButton`

Текст

Чтобы отобразить одну строку редактируемого или доступного только для чтения текста, используйте команду `textField`. В следующем примере проиллюстрировано создание изменяемого текстового поля. Завершив редактирование текста, важно всегда нажимать клавишу `Enter`. Если вы не нажмете `Enter`, то изменения, внесенные в текст, будут проигнорированы. Текстовое поле `textField` показано на рис. 3.26.

```
window;
    columnLayout -adjustableColumn true;
        textField -editable true;
    showWindow;
```

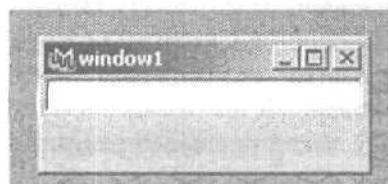


Рис. 3.26. Пример поля `textField`

Если вам нужно отобразить несколько строк текста, воспользуйтесь командой `scrollField`. Элемент `scrollField` может отобразить либо текст, доступный для редактирования, либо текст, доступный только для чтения. Чтобы разрешить изменение текста, просто установите флаг `-editable`. Следующий пример показывает создание поля `scrollField` с пословным переносом. Элемент также использует нестандартный шрифт и является редактируемым. Когда пользователь, работая с этим элементом, нажимает клавишу `Enter`, выполняется параметр `command`. В данном случае происходит вызов вашей собственной процедуры `myGetText`. На рис. 3.27 показан конечный результат.

```
window;
```

```
    columnLayout -adjustableColumn true;
    scrollField -wordwrap true
        -text "This is a section of text that is editable"
        -font boldLabelFont
        -editable true
        -command "myGetText()";
showWindow;
```

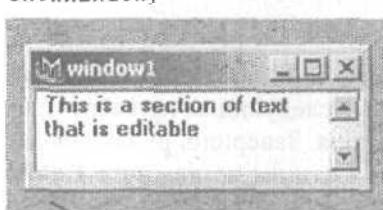


Рис. 3.27. Пример поля `scrollField`

Если вы не используете возможности параметра `-command`, то всегда можете получить пользовательский текст при помощи такой команды:

```
scrollField -query -text $scrollFieldName;
```

Для отображения одной строки статического (неизменяемого) текста вызовите команду `text`. Созданный текст может быть выровнен по левому, правому краю или по центру. Кроме того, вы можете пользоваться специальным шрифтом. В следующем примере окно содержит центрированный текстовый элемент, выведенный наклонным шрифтом `obliqueLabelFont`. Окончательный результат представлен на рис. 3.28.

```
window;  
    columnLayout -adjustableColumn true;  
        text -label "Middle" -align "center"  
            -font "obliqueLabelFont";  
showWindow;
```

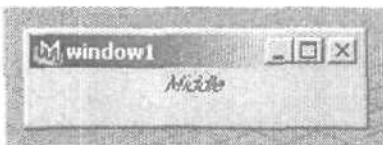


Рис. 3.28. Пример текста

Списки

Для вывода на экран списка текстовых элементов, выбор которых доступен пользователю, используйте команду `textScrollList`. Вы можете настроить конфигурацию элемента `textScrollList` так, чтобы обеспечить выбор одного или нескольких элементов. Последнее достигается установкой истинного значения параметра `-allowMultipleSelection`. Выбрав первый элемент, пользователь может выделить дополнительные элементы, нажав клавишу `Ctrl`.

Следующий пример служит иллюстрацией создания списка `textScrollList` с четырьмя видимыми строками. Пользователь может выбрать несколько элементов. В список заносятся пять элементов, четвертый элемент выделяется. Далее выполняется прокрутка списка, что позволяет сделать элемент с индексом 4 видимым. Окончательный вид `textScrollList` показан на рис. 3.29.

```
window;  
    columnLayout;  
    textScrollList -numberofRows 4  
        -allowMultiSelection true  
        -append "alpha"  
        -append "beta"  
        -append "gamma"  
        -append "delta"  
        -append "epsilon"  
        -selectItem "delta"  
        -showIndexedItem 4;  
showWindow;
```

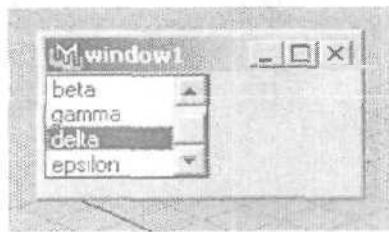


Рис. 3.29. Пример списка `textScrollList`

Перечень выделенных элементов можно получить, используя следующую команду:

```
textScrollList -query -selectedItem $textScrollListName;
```

Для получения списка индексов выделенных элементов обратитесь к команде

```
textScrollList -query -selectIndexedItem $textScrollListName;
```

Заметьте, что в отличие от индексов прочих массивов MEL, здесь индексы начинаются не с 0, а с 1.

Группы

Нередко возникает желание создать группу элементов для отображения некоторых данных. К примеру, вы хотите вывести на экран **число** с плавающей запятой, а затем предложить пользователю для **редактирования** этого числа **бегунок** и поле **ввода**, снабдив их поясняющей меткой. К счастью, Maya предоставляет несколько удобных команд, позволяющих создавать целые группы элементов и управлять ими. Следующий пример демонстрирует создание группы с **бегунком** для управления вещественным значением. С помощью единственной команды **floatSliderGrp** создаются метка, поле редактирования и бегунок. Эта группа имеет метку **Temperature** (**Температура**) и содержит поле ввода. Метку и поле ввода можно отключить. Начальное значение равно 76. Полученная в результате группа **floatSliderGrp** показана на рис. 3.30.

```
window;
columnLayout;
floatSliderGrp -label "Temperature"
    -value 76
    -field true
    -minValue -10.0 -maxValue 100.0
    -fieldMinValue -100.0 -fieldMaxValue 100.0;
showWindow;
```

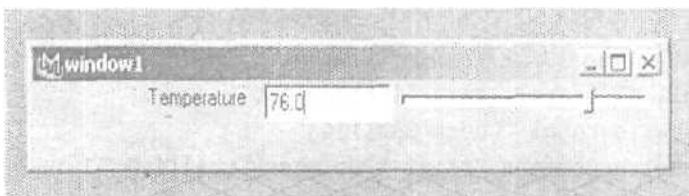


Рис. 3.30. Пример группы **floatSliderGrp**

Обратите внимание на возможность задания различных диапазонов значений для бегунка и поля ввода. Если в поле ввода вводится число, превышающее максимальное значение позиции бегунка, то диапазон бегунка автоматически расширяется. Вместе с тем он никогда не выйдет за пределы минимального и максимального значений, приписанных полю ввода.

Все команды языка **MEL** для организации групп и управления ими заканчиваются на **Grp**. В табл. 3.7 перечислены имеющиеся группы и свойства, которыми те обладают.

ТАБЛИЦА 3.7. СУЩЕСТВУЮЩИЕ ГРУППОВЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Группа	Описание
colorIndexSliderGrp	Прокрутка последовательности цветов
colorSliderGrp	Правка цвета с использованием образцов или бегунка
floatFieldGrp	Правка вещественных чисел при помощи полей ввода
floatSliderGrp	Правка вещественных чисел при помощи поля ввода или бегунка
intFieldGrp	Правка целых чисел при помощи полей ввода
intSliderGrp	Правка целых чисел при помощи поля ввода или бегунка
radioButtonGrp	Набор кнопок-переключателей
textFieldGrp	Правка текста с использованием метки и поля ввода

Наряду с ними, имеются и расширения некоторых групп, включающие в свой состав кнопку. Команды **colorSliderButtonGrp**, **floatSliderButtonGrp** и **textFieldButtonGrp** служат расширением эквивалентных команд **<элемент>SliderGrp**. В следующем примере для создания группы с текстовым полем и кнопкой служит команда **textFieldButtonGrp**. Команда, указанная как параметр **-buttonCommand**, выполняется при нажатии кнопки. На рис. 3.31 приведен конечный результат.

```

window;
columnLayout;
textFieldButtonGrp -label "Word" -text "incredulous"
                    -buttonLabel "Check Spelling"
                    -buttonCommand "print \"do check spelling\"";
showWindow;

```

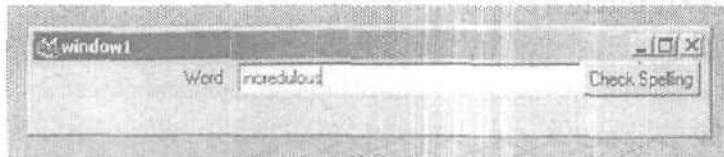


Рис. 3.31. Пример группы textFieldButtonGrp

Изображения

Чтобы вывести на экран статическое растровое изображение, можно воспользоваться либо командой `picture`, либо командой `image`. В среде Windows команда `picture` способна отображать `.xpm` и `.bmp`-файлы, тогда как в `Linux` - только `.xpm`-файлы. Команда `image` может выводить не только эти форматы, но и многие другие.

Сначала Maya ищет растровые картинки в каталоге изображений пользователя, а затем переходит в каталог `maya_install\icons`. Чтобы узнать имя текущего каталога пользовательских изображений, выполните следующую команду;

```
string $imgsDir = `internalVar -userBitmapsDir`;
```

Если файла изображения нет ни в одном из двух каталогов, вы можете использовать полный путь к этому файлу.

В следующем примере команда `picture` служит для вывода изображения `sphere.xpm`. Полученный в результате интерфейс показан на рис. 3.32.

```

window;
columnLayout;
picture -image "sphere.xpm";
showWindow;

```



Рис. 3.32. Пример картинки

Растровые изображения по умолчанию (*sphere.xpm*, *cone.xpm* и т. д.) встроены в среде Windows непосредственно в исполняемый файл Maya (*maya.exe*). На других платформах растровые изображения по умолчанию хранятся в подкаталогах Maya.

Команда *image* служит для отображения всех растровых форматов, которые поддерживает Maya. Сами поддерживаемые форматы на разных платформах различны. Для получения полного перечня форматов изображений, которые поддерживаются на текущей платформе, воспользуйтесь следующими командами:

```
global string $imgExt[];
```

```
print $imgExt;
```

Под Windows результаты будут такими:

```
als
```

```
avi
```

```
cin
```

```
eps
```

```
gif
```

```
jpeg
```

```
iff
```

```
iff
```

... продолжение

Заметим, что это очень нестандартный способ получения форматов изображений и он может оказаться некорректным в некоторых версиях Maya.

Следующий пример иллюстрирует вывод изображения в формате *.gif* на экран по команде *image*. Обратите внимание на указание полного пути к файлу, поскольку этого файла нет ни в одном из каталогов изображений, принятых в Maya по умолчанию. Окончательный результат приведен на рис. 3.33.

```
window;  
columnLayout -height 950 -adjustableColumn true;  
image -image "D:\\Temp\\Vince.gif";  
showWindow;
```



Рис. 3.33. Пример изображения

Панели

Поместить панель в пределах окна можно при помощи одной из команд работы с панелями: `outlinerPanel`, `hardwareRenderPanel`, `modelPanel`, `nodeOutliner`, `spreadSheetEditor` и `hyperPanel`. В следующем примере создано окно с панелью `modelPanel`, отображающей текущее окно просмотра. Панель `modelPanel` показана на рис. 3.34.

```
window;
```

```
panelLayout;  
modelPanel;  
showWindow;
```

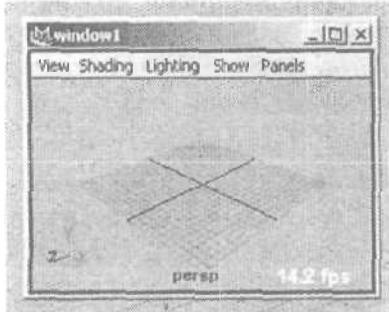


Рис. 3.34. Пример панели `modelPanel`

Для создания окна, одновременно содержащего схему сцены и текущее окно просмотра, используйте следующие команды. Итоговый интерфейс представлен на рис. 3.35.

```
window -width 700 -height 500;  
paneLayout -configuration "vertical2";  
outlinerPanel;  
setParent ...;  
modelPanel;  
showWindow;
```

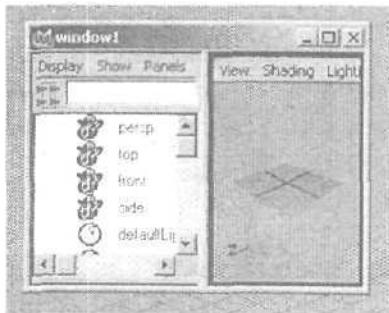


Рис. 3.35. Примеры панелей

Наряду со специальными командами для работы с панелями, вы можете использовать универсальные команды `getPanel` и `panel`, предназначенные для получения и установки параметров данной панели.

```
getPanel -all; // Получить перечень всех панелей  
getPanel -underPointer; // Вернуть имя панели под курсором
```

```
panel -edit -menuBarVisible true $panelName; // Отобразить строку меню  
panel -copy $panelName; // Создать копию панели
```

Инструменты

Не исключено, что вам потребуется добавить в окно и свои собственные инструментальные кнопки. От обычных кнопок они отличаются тем, что активизируют соответствующий инструмент, а не выполняют команду языка MEL. Следующий пример служит иллюстрацией использования инструментальных кнопок, а также их группировки при помощи элемента `toolCollection`. Результаты изображены на рис. 3.36.

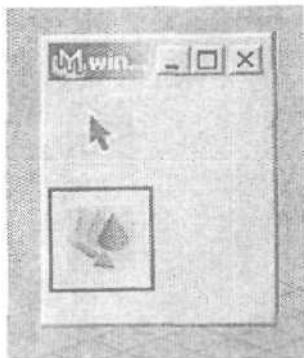


Рис. 3.36. Пример элемента `toolCollection`

```
window;
    columnLayout;
        toolCollection;
            toolButton -tool selectSuperContext
                -toolImage1 selectSuperContext "aselect.xpm";
            toolButton -tool moveSuperContext
                -toolImage1 moveSuperContext "move_M.xpm";
    showWindow;
```

Приступая к построению кнопок (`toolButton`), необходимо создать элемент `toolCollection`. Все последующие экземпляры `toolButton` будут содержаться именно в нем. Команда `toolButton` инициирует создание инструментальной кнопки с тем или иным инструментом, указанным при помощи флага `-tool`. Инструмент представляет собой имя инструментального контекста, активизируемого по нажатию кнопки.

3.6.6. Связывание элементов

Очень часто возникает желание сделать так, чтобы `один` или группа элементов пользовательского интерфейса отражала текущее значение атрибута узла. Кроме того, хотелось бы надлежащим образом модифицировать атрибут узла при изменении его значения через интерфейс пользователя. Maya предоставляет набор команд, которые автоматизируют решение этих задач.

Атрибутная группа

Атрибутные группы являются расширением обычных групп и позволяют связывать с атрибутом узла значение, представленное данной группой. Для этого принадлежащему группе параметру `-attribute` присваивается имя узла и связанного с ним атрибута. В следующем примере строится новый шейдер, а его цвет выводится на экран и управляется группой `attrColorSliderGrp`. Эта полученная в результате группа показана на рис. 3.37.

```
string $objName = `shadingNode -asShader blinn`;
window;
columnLayout;
attrColorSliderGrp -attribute ($objName+.color")
-label ($objName + "'s color");
showWindow;
```

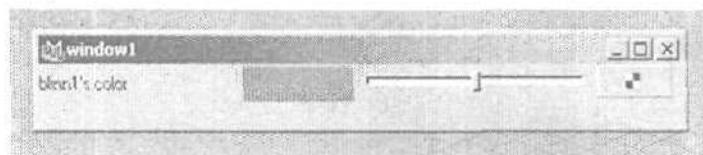


Рис. 3.37. Пример группы attrColorSliderGrp

Первым создается новый узел шейдера `blinn`. Имя нового узла важно запомнить, так как позднее оно потребуется атрибутной группе.

```
string $objName = `shadingNode -asShader blinn`;
```

После создания окна организуется атрибутная группа. Параметр `-attribute` принимает значение принадлежащего узлу атрибута `color`. Также устанавливается метка группы.

```
attrColorSliderGrp -attribute ($objName+.color")
-label ($objName + "'s color");
```

При изменении цвета атрибутная группа отобразит его самое последнее значение. Если, к примеру, вы изменили цвет шейдера в редакторе `Hypershade`, группа автоматически обновится. С изменением атрибутной группы происходит автоматическое обновление атрибута `color` в узле шейдера. Вы не должны беспокоиться и тогда, когда иным станет имя узла. Соединение между атрибутной группой и атрибутом узла по-прежнему сохранится.

В следующем примере мы создадим сферу и организуем атрибутную группу для управления ее масштабом по оси *x*. На рис. 3.38 представлен итоговый вид интерфейса.

```
string $objName[] = `sphere`;
window;
columnLayout;
attrFieldSliderGrp -attribute ($objName[0]+".scaleX")
-min -10 -max 10
-label ($objName[0] + "'s x scale");
showWindow;
```



Рис. 3.38. Группа элементов для управления сферой

Результат выполнения команды `sphere` сохраняется в переменной-массиве `$objName`. Команда `sphere` возвращает имя принадлежащего сфере узла `transform`, а также узел, создавший сферический объект.

```
string $objName[] = `sphere`;
```

Для атрибута сферы с именем `scaleX` создается группа `attrFieldSliderGrp`. Поле ввода и диапазон бегунка ограничиваются промежутком от -10 до 10.

```
attrFieldSliderGrp -attribute ($objName[0]+".scaleX")
-min -10 -max 10
-label ($objName[0] + "'s x scale");
```

Этот последний пример показывает, как атрибутная группа управляет единственным значением. Можно организовать свою группу `attrFieldSliderGrp` для каждой из трех осей. Можно создать и одну группу, которая управляет множеством атрибутов. Для управления значениями всех масштабных коэффициентов (*x*, *y*, *z*) можно выполнить следующие команды. Результат показан на рис. 3.39.

```

string $objName[] = `sphere`;
window;
columnLayout;
attrFieldGrp -attribute ($objName[0]+".scale")
-label ($objName[0] + "'s scale");
showWindow;

```

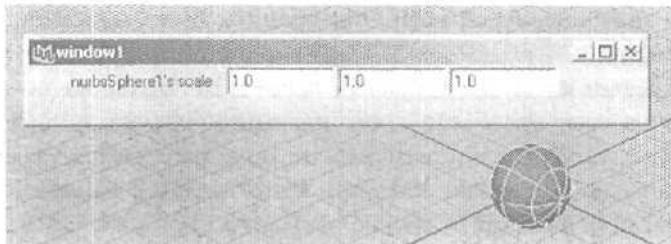


Рис. 3.39. Пример группы attrFieldGrp

Команда attrFieldGrp генерирует группу, способную вместить все дочерние атрибуты **scale** (**scaleX**, **scaleY**, **scaleZ**).

Подчас атрибут не имеет собственного значения, а взамен представляет собой соединение с другим атрибутом. Текстуре можно, например, придать значение карты неровностей шейдера. Следующий код организует атрибутную группу, которая существенно упрощает выбор карты пользователем или переход к существующей карте. Атрибутная группа связана с атрибутом карты неровностей узла шейдера (**normalCamera**). Результат показан на рис. 3.40.

```

string $obj = 'shadingNode -asShader blinn';
window;
columnLayout;
attrNavigationControlGrp -attribute ($obj + ".normalCamera")
-label "Bump Map";
showWindow;

```



Рис. 3.40. Пример группы attrNavigationControlGrp

Заметьте, что если вы не укажете `-showButton false`, то при использовании команды `attrColorSlider1Grp` кнопка навигации будет показана автоматически.

Поле имени

Зачастую возникает необходимость отобразить имя **объекта**, а затем дать пользователю возможность его исправить. Элемент `nameField` позволяет автоматически отображать и редактировать имя объекта. Он постоянно обновляется и потому содержит текущее значение, даже если имя **объекта** было изменено с использованием других средств.

Следующий пример показывает создание элемента `nameField`. Вслед за построением объекта создается элемент `nameField`, и его параметр `-object` принимает значение имени этого объекта. Когда оно изменяется, поле изменяется также. Изменение имени в поле ввода и нажатие клавиши `Enter` ведет к обновлению имени объекта. При этом обновляются и другие элементы типа `nameField`, ссылающиеся на тот же самый объект. Получившийся в результате интерфейс пользователя показан на рис. 3.41.

```
string $torusName[] = 'torus';
window;
columnLayout -adjustableColumn true;
nameField -object $torusName[0];
showWindow;
```

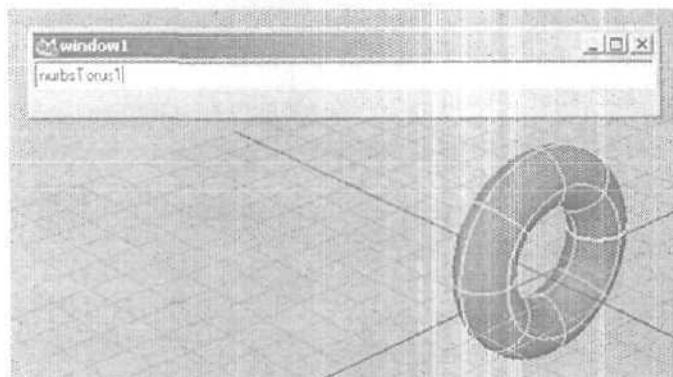


Рис. 3.41. Пример элемента `nameField`

Прочие элементы

Даже если вы не обращаетесь ни к одной из атрибутных групп и не используете элемент `nameField`, вы по-прежнему можете связать атрибут узла и элемент управления. Это делается при помощи команды `connectControl`. В следующем примере организовано множество разнообразных элементов интерфейса. Их значения затем привязываются к атрибутам узла `sphere`. Окончательный вид интерфейса приведен на рис. 3.42.

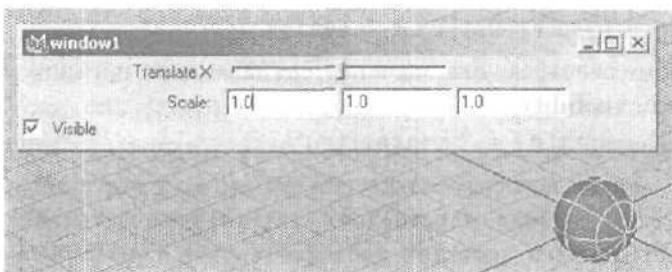


Рис. 3.42. Пример работы с командой `connectControl`

```

string $sphereObj[] = `sphere`;
window;
columnLayout;
string $fieldGrp = 'floatSliderGrp -label "Translate X"
    -min -100 max 100';
connectControl $fieldGrp ($sphereObj[0] + ".translateX");

string $grp = 'floatFieldGrp -label "Scale:" -numberOfFields 3';
connectControl -index 2 $grp ($sphereObj[0] + ".scaleX");
connectControl -index 3 $grp ($sphereObj[0] + ".scaleY");
connectControl -index 4 $grp ($sphereObj[0] + ".scaleZ");

string $check = 'checkBox -label "Visible"';
connectControl $check ($sphereObj[0] + ".visibility");
showWindow;

```

Создав сферу, вы организуете группу `floatSliderGrp`. По команде `connectControl` описывающий сферу атрибут `translateX` соединяется с элементом управления.

```
connectControl $fieldGrp ($sphereObj[0] + ".translateX");
```

С этого момента значение, отображаемое группой, всегда соответствует текущему значению атрибута узла. Далее создается группа `floatFieldGrp` с тремя полями. Каждое отдельное поле связывается с независимым масштабным атрибутом сферы. Для соединения атрибута с конкретным элементом группы необходимо указать его индекс. В данном случае поле для отображения масштаба по оси `x` - это второй элемент группы (индекс 2); первым элементом является метка (индекс 1).

```
connectControl -index 2 $grp ($sphereObj[0] + ".scaleX");
```

Наконец, создается элемент `checkBox`. Его значение связывается с признаком видимости сферы - атрибутом `visibility`.

```
connectControl $check ($sphereObj[0] + ".visibility");
```

Команда `connectControl` может использоваться при работе со следующими элементами: `floatField`, `floatScrollBar`, `floatSlider`, `intField`, `intScrollBar`, `intSlider`, `floatFieldGrp`, `intFieldGrp`, `checkBox`, `radioCollection` и `optionMenu`.

3.6.7. Обратная связь с пользователем

Вместе с предоставлением пользователю интерактивного интерфейса нередко возникает необходимость в обеспечении обратной связи. Такая обратная связь может принимать форму `указаний`, справочных сообщений, а может просто показать пользователю, что во время выполнения сложной операции ввод информации невозможен.

Помощь

Почти все элементы интерфейса позволяют задать некоторый справочный текст. Для этого используется флаг `-annotation`. В следующем примере создается окно с кнопкой. Кнопка снабжена примечанием «`This; is pop-up help`» («Это - всплывающая подсказка»). Когда вы переведете курсор на кнопку и задержите его в таком положении на короткое время, во всплывающей подсказке будет выведен текст примечания. Результат такого действия показан на рис. 3.43.

```
window;  
    columnLayout;  
        button -label "Hover over me" -annotation "This is pop-up help";  
    showWindow;
```



Рис. 3.43. Пример всплывающей подсказки

Всплывающая подсказка - это быстрый и простой способ вывода справки или других инструкций, относящихся к данному элементу интерфейса. Ею можно пользоваться в сочетании с интерфейсным элементом `helpLine`, предназначенным для непрерывного отображения примечания. Следующий пример демонстрирует создание в окне элемента `helpLine`. Примечания добавляются к пункту меню и кнопке. Когда вы наведете курсор на любой из этих элементов, `helpLine` сразу же выведет связанный с ним текст аннотации. Полученный в результате элемент `helpLine` показан на рис. 3.44.

```
window -height 600 -menuBar true;  
    menu -label "Start"  
        menuItem -label "Processing" -annotation "Starts the processing";  
  
    string $form = `formLayout`;  
    button -label "Initialize" -annotation "Initialize the data";  
    string $frame = `frameLayout -labelVisible false`;  
    helpLine;  
  
    formLayout -edit  
        -attachNone $frame "top"  
        -attachForm $frame "left" 0  
        -attachForm $frame "bottom" 0  
        -attachForm $frame "right" 0  
        $form;  
  
    showWindow;
```

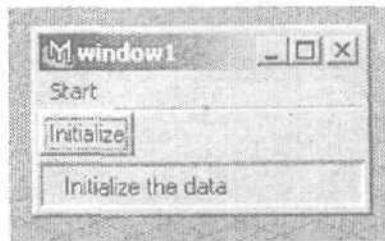


Рис. 3.44. Пример элемента `helpLine`

Отображение индикатора

Чтобы проинформировать пользователя о том, что компьютер занят и не может принять никакой входной информации, используйте команду `waitForCursor`. Курсор мыши сменит свой вид на «песочные часы» или аналогичное ему изображение.

```
waitForCursor -state on;
```

Чтобы вернуть курсор в прежнее состояние, просто выполните команду

```
waitForCursor -state off;
```

Существует немало различных способов сообщить пользователю о том, что в данное время компьютер выполняет ту или иную операцию. Самый простой способ - воспользоваться командой `progressWindow`. Она создаст окно с индикатором проделанной работы. По мере выполнения операции просто **инкрементируйте** параметр этого окна с именем `-progress`. Когда окно индикатора показано на экране, курсор мыши автоматически переводится в состояние ожидания. В следующем примере создается окно `progressWindow`, а затем по ходу выполнения операции его состояние обновляется. Пользователь вправе отменить операцию в любое время. Окно `progressWindow` изображено на рис. 3.45.

```
int $amount = 0;
```

```
progressWindow
    -title "Working"
    -progress $amount
    -status "Completed: 0X"
    -isInterruptable true;

while (true)
{
```

```

II Здесь выполните еще один шаг длинной операции
pause -seconds 1; // Холостая операция

if( `progressWindow -query -isCancelled` ) break;

$amount += 1;

progressWindow -edit
    -progress $amount
    -status ("Completed: "+$amount+"%");
}

progressWindow -endProgress;

```

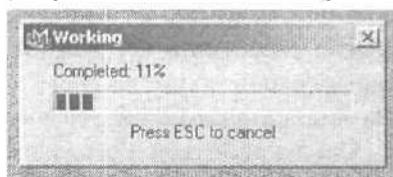


Рис. 3.45. Пример окна progressWindow

Сначала будет создано окно **progressWindow**. Всякий раз можно вывести лишь одно такое окно. Минимальное и максимальное значения диапазона можно задать явно, однако в данном случае используются значения по умолчанию - 0 и 100. Текст, отражающий состояние индикатора, содержит информацию об относительной завершенности операции. Установив флаг **-isInterruptable**, вы разрешили пользователю отменить эту операцию. Если вы не хотите давать пользователю такой возможности, сбросьте этот флаг, придав ему ложное значение.

progressWindow

```

    -title "Working"
    -progress $amount
    -status "Completed: 0X"
    -isInterruptable true;

```

Далее начинается выполнение операции. По мере продвижения процесса будем проверять, не пытался ли пользователь отменить данную операцию. Если это так, завершим работу немедленно.

```
if( `progressWindow -query -isCancelled` ) break;
```

Наконец, по завершении очередного шага обновим окно **progressWindow**. Так как приращение объема работы всякий раз составляет 1%, просто отобразим его текущее значение `$amount`. Также выведем текст сообщения о состоянии процесса.

```
$amount += 1;
```

```
progressWindow -edit
    -progress $amount
    -status ("Completed: "+$amount+"%");
```

Как только операция завершена, важно сообщить об этом окну **progressWindow**. Для этого используйте следующую команду:

```
progressWindow -endProgress;
```

Если окно **progressWindow** не узнает о завершении операции, оно не закроется и не восстановит курсор. Вызвав эту команду, важно обеспечить корректное завершение работы **progressWindow** даже при отмене операции пользователем.

Наряду с применением такого глобального окна-индикатора, можно создавать отдельные индикаторы, размещенных в окнах. Следующий код позволяет создать окно с индикатором. Для этого используется команда **progressBar**. Вы можете задать диапазон значений и объем выполненной работы. Результаты представлены на рис. 3.46.

```
window;
columnLayout;
string $progCtrl = "progressBar -maxValue 5 -width 300";
button -label "Next step"
    -command ("progressBar -edit -step 1 " + $progCtrl);
showWindow;
```

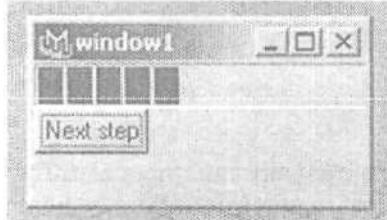


Рис. 3.46. Пример элемента **progressBar**

3.7. Выражения

Как известно, в Maya вы можете анимировать любой атрибут, создав для этого ряд ключевых кадров. Другой способ анимации атрибутов - с использованием выражений - имеет одно отличие: значение атрибута определяется не установкой ключевых кадров, а командами языка MEL. Эти команды выполняются в каждом кадре, а их результат сохраняется в том атрибуте, с которым связано данное выражение. Такой подход широко известен как *процедурная анимация*. О ней говорят в том случае, когда анимацией атрибута управляет программа (команды MEL), а не серия ключевых кадров. Выражения можно использовать для создания сложных анимационных решений при минимальном объеме ручного труда или при полном его отсутствии. Это очень действенный метод анимации атрибутов, так как по своему желанию вы вправе совершенно свободно обращаться к любым командам или операторам языка MEL. Использование выражений позволяет управлять, вообще говоря, любым атрибутом. Объектом процедурной анимации, по сути, может стать практически любой компонент Maya.

К сожалению, слово *выражение* (*Expression*) получает теперь два значения. Первое из них, принятое в языках программирования, - это одна или несколько программных конструкций; выражением, например, считается запись $1 + 5$. Второе значение связано с использованием операторов MEL для анимации атрибутов, что представляет собой тему этого раздела. Чтобы подчеркнуть различие между ними, будем употреблять написанное с заглавной буквы слово *Expression* для обозначения выражений, позволяющих анимировать атрибуты, а написанное со строчной буквы слово *выражение* станем применять там, где оно относится к программным операторам MEL.

3.7.1. Освоение выражений

Данный раздел посвящен применению выражений Expression для анимации атрибутов. Не вдаваясь в изучение синтаксиса и других вопросов, вы приступите к активным действиям, сразу же начав пользоваться элементами Expression. Это позволит вам быстро понять, насколько они просты и полезны.

Прыгающий мяч

Первое упражнение - демонстрация приемов управления движением объекта посредством Expression. Основываясь на применении Expression, мы заставим прыгать вверх и вниз шар, который перемещается по сцене.

1. Откройте сцену **Wiggle.ma**.
2. Нажмите клавиши **Alt+v** или щелкните по кнопке **Play**.
По сцене станет передвигаться небольшая сфера с именем **ball**.
3. Выделите объект **ball**.
4. Выберите пункт меню **Window | Animation Editors | Expression Editor...**
(Окно | Редакторы анимации Редактор выражений...).
Окно редактора выражений **Expression Editor** станет именно тем окном, где вы будете создавать и редактировать все свои выражения.
5. В поле **Expression Name** (Название выражения) наберите текст **WiggleBall**.
Назначение имени выражению Expression важно потому, что это облегчит его отыскание в дальнейшем, когда количество таких выражений вырастет. Так как выделенным объектом является сейчас **ball**, то в разделе **Attributes** (Атрибуты) содержится перечень его атрибутов. Задача состоит в анимации вертикального движения шара, а значит, необходимо составить выражение, управляющее атрибутом **translateY** объекта **ball**.
6. В списке **Attribute** щелкните по атрибуту **translateY**.
7. В поле **Expression:** наберите следующий текст:
`ball.translateY = time;`
Вы можете использовать и краткое имя этого атрибута. К примеру, можно было записать:
`ball.ty = time;`
Вслед за этим Maya автоматически выполнит за вас подстановку полного имени. Такими именами пользоваться лучше всего, так как подобная запись более очевидна.
8. Для создания выражения щелкните по кнопке **Create** (Создать).
Если в Expression допущена синтаксическая ошибка или имеются другие проблемы, на экран будет выведено соответствующее сообщение. Тем не менее в редакторе **Expression Editor** подробности ошибки не отражаются. Взамен нужно открыть **Script Editor** и повторить попытку. Так, в редакторе сценариев вы увидите развернутый вариант сообщения об ошибке. Чтобы точно указать место возникновения ошибки в Expression, включите опцию **Show Line Numbers** редактора **Script Editor**.
9. Щелкните по кнопке **Play**.
Теперь в ходе анимации шар монотонно поднимается вверх. Его позиция по оси **у** приравнена к текущему времени. При переходе от одного кадра к другому

выражение Expression вычисляется заново. Если в каждом кадре оно принимает новое значение, то атрибут становится анимированным. Коль скоро позиция по оси *y* имеет значение текущего времени, которое монотонно возрастает с каждым новым кадром, значит, и позиция по оси *y* тоже увеличивается.

Заметьте, что в отличие от стандартных сценариев MEL, где для получения текущего времени требуется вызов команды `currentTime`, в Expression вы можете просто использовать предопределенную переменную `time`. Чтобы сделать написание Expression более удобным, Maya предоставляет две предопределенные переменные: `time` и `frame`. Первая содержит текущее время в секундах, а вторая - номер текущего кадра. Переменная `frame` автоматически рассчитывается на основе текущего времени с учетом настроек частоты кадров. Эти переменные доступны только для чтения, поэтому при написании Expression вам не удастся их изменить.

Взамен тривиального движения вверх хотелось бы добиться отрывистых движений вверх и вниз вдоль оси *y*.

10. Введите в поле **Expression** следующий текст:

```
ball.translateY = rand(0,1);
```

Теперь параметр трансляции шара по оси *y* является результатом выполнения команды `rand`. Каждый раз во время своего вызова эта команда выдает новое случайное число. Вызывая данную функцию с аргументами 0 и 1, можно получать только случайные числа в промежутке от 0 до 1. В результате значение атрибута шара с именем `translateY` носит случайный характер, но гарантированно никогда не выходит за пределы этого диапазона.

11. Чтобы сохранить изменения и потребовать их вступления в силу, щелкните по кнопке **Edit** (Редактировать).

12. Щелкните по кнопке **Play**.

Теперь анимированный шар прыгает вверх-вниз. К сожалению, такое перемещение стало немного странным. Всякий раз команда `rand` порождает абсолютно новое случайное число, поэтому в каждом кадре шар перемещается в совершенно новое положение.

Вы же хотите добиться того, чтобы шар двигался случайным, но не столь эксцентричным образом. К счастью, есть и другая команда - `noise`, которая также придает некоторую случайность, однако легче поддается управлению.

13. Введите в поле **Expression** следующий текст:

```
ball.translateY = noise( ball.translateX );
```

Теперь трансляция шара по оси *y* является результатом выполнения команды noise. Эта команда генерирует случайные числа, однако в отличие от rand, выдает одно и то же случайное число в случае ее вызова с одним и тем же входным значением. В данном примере вход этой команды - трансляция шара вдоль оси *x*. Поэтому когда шар находится в одном и том же положении по *x*, команда выдает одну и ту же случайную позицию по *y*. Коль скоро положение по оси *x* есть результат анимации, шар перемещается по экрану, а его новое значение ball.translateX служит входом команды noise, которая, в свою очередь, управляет атрибутом ball.translateY.

14. Щелкните по кнопке **Edit**.

15. Щелкните по кнопке **Play**.

Движения шара перестали быть такими неуправляемыми, как прежде, однако с ними по-прежнему что-то не так. Хотелось бы сделать их более гладкими. Так как команда noise порождает случайное число на основании значения своего входа, то для получения случайных чисел, изменения которых не столь беспорядочны, входное значение не должно так сильно меняться от одного кадра к другому. Взамен использования положения шара по оси *x* «как есть» можно взять его половину. Стало быть, значение, передаваемое команде noise, не будет изменяться так сильно, а значит, положение шара по оси *y* не будет варьироваться настолько существенно.

16. Введите в поле **Expression** следующий текст:

```
ball.translateY = noise( ball.translateX / 2 );
```

17. Щелкните по кнопке **Edit**.

18. Щелкните по кнопке **Play**.

Шар начнет двигаться по экрану, однако изгибы его траектории вверх и вниз, став более гладкими, перестанут происходить так внезапно.

Использование Expression позволило косвенно установить отношение, согласно которому значение атрибута шара **translateY** зависит от значения атрибута **translateX**. В самом деле, каждый раз при изменении **translateX** выражение Expression автоматически пересчитывается, а значение **translateY** обновляется.

19. Немного уменьшите масштаб изображения [**Zoom** (Масштаб)].

20. Щелкните по инструменту **Move**.

21. В интерактивном режиме переместите шар назад, а затем - вперед вдоль оси *x*. Положение шара по *y* непрерывно обновляется по мере смены положения по оси *x*. Анимация полностью управляет вашим выражением Expression,

поэтому у вас больше нет необходимости в явной установке **ключевых кадров**. Одна из самых замечательных возможностей Expression состоит именно в том, что, как только выражение задано, анимация атрибута производится автоматически.

Вспышки

В следующем примере движение объекта сопровождается автоматическими вспышками.

1. Откройте сцену **Flash.ma**.

2. Щелкните по кнопке **Play**.

Сфера **ball** начнет свое движение по некоторой траектории.

3. Щелкните правой кнопкой по объекту **ball**, затем выберите из выпадающего меню пункт **Materials | Material Attributes...** (Материалы Атрибуты материалов...).

Сейчас в качестве материала объекта **ball** будет выбрана сирена (siren).

4. Для выбора материала **siren** щелкните по кнопке **Select** (Выбрать) в нижней части редактора **Attribute Editor**.

5. Выберите пункт меню **Windows | Animation Editors | Expression Editor...**

6. Введите в поле **Expression Name** имя **Flashing**.

Обратите **внимание** на то, что поля **Selected Obj & Attr** (Выбранные объекты и атрибуты) и **Default Object** (Объект по умолчанию) содержат значение **siren**. При создании Expression текущий выделенный объект становится объектом по умолчанию. Если вы начнете составлять выражение Expression, но не укажете имя **узла**, к которому оно применяется, Maya автоматически отнесет его к тому узлу, что содержится в поле **Default Object**.

7. Введите в поле **Expression** следующий текст:

`colorR = 1;`

Заметьте: вы не указали, цвет какого объекта следует изменить, обозначив лишь атрибут.

8. Щелкните по кнопке **Create**.

Шар сменит свой цвет на розовый.

9. Введите в поле **Expression** следующий текст:

`colorR = 0;`

10. Щелкните по кнопке **Edit**.

На этот раз Expression автоматически обновится, что приведет к использованию объекта по умолчанию, таким образом, выражение будет иметь вид;

```
siren.colorR = 0;
```

Также обратите внимание, что поле **Selected Obj & Attr** получило значение **siren.colorR**.

11. Введите в поле **Expression** следующий текст:

```
siren.colorR = colorG = colorB = 1;
```

12. Щелкните по кнопке **Edit**.

Поле **Expression** автоматически обновится и включит в текст имя объекта по умолчанию.

```
siren.colorR = siren.colorG = siren.colorB = 1;
```

Несмотря на то что при быстром наборе Expression применение объекта и атрибута по умолчанию может оказаться удобным, лучше всего писать полные имена и указывать тот атрибут объекта, который вы хотите изменить. Так как количество и сложность выражений Expression растет, то надежда на такую автоматическую подстановку может обернуться трудноуловимыми ошибками при подстановке непредусмотренного имени. Пользуясь полными именами и атрибутами, вы можете быть уверены, что работаете с нужным объектом.

13. Щелкните по объекту **ball**.

Объект **ball** является выделенным, поэтому выражение Expression, описывающее материал, уже не отображается.

14. Выберите в редакторе **Expression Editor** пункт меню **Select Filter | By Expression Name** (Выбрать фильтр По имени выражения).

Поскольку ваше выражение называется **Flashing**, вы увидите это имя в списке **Expressions**.

15. Щелкните по элементу **Flashing**.

16. Задайте в качестве текста **Expression** следующие команды:

```
vector $pos = <<ball.translateX, ball.translateY, ball.translateZ>>;  
float $dist = raag( $pos );  
siren.colorR = siren.colorG = siren.colorB = $dist;
```

17. Щелкните по кнопке **Edit**.

18. Щелкните по кнопке Play.

Шар станет абсолютно белым. Его цвет будет принят равным расстоянию до начала отсчета (0, 0, 0). Для этого сначала определяется текущее положение шара, представленное в виде вектора, а затем рассчитывается длина этого вектора. С Expression связана одна проблема. Значения цветов должны лежать в диапазоне от 0 до 1, однако вы указываете значения, превышающие 1, так как используете расстояние до объекта. Необходимо задать в Expression допустимые цветовые значения.

39. Задайте в качестве текста Expression следующие команды:

```
vector $pos = <<ball.translateX, ball.translateY, ball.translateZ>>;  
float $dist = mag( $pos );  
siren.colorR = siren.colorG = siren.colorB = (cos( $dist ) + 1) / 2;
```

20. Щелкните по кнопке Edit.**21. Щелкните по кнопке Play.**

Теперь при движении по своей траектории шар медленно вспыхивает и гаснет. Для получения осциллирующей волны, которая превосходно подходит для мерцаний, поскольку позволяет неоднократно и постепенно переходить от высокой яркости к низкой, использована функция cos. Непосредственный результат функции косинуса, к сожалению, принадлежит диапазону между -1 и 1. Нам нужны числа от 0 до 1, поэтому значение косинуса увеличивается на 1. Теперь сумма расположена между 0 и 2. Затем этот результат делится пополам, что дает окончательное значение в диапазоне от 0 до 1. Мерцание уже работает, однако хотелось бы его ускорить. Для этого достаточно изменить масштаб входного значения команды cos. Сделав ее вход больше, вы заставите шар вспыхивать быстрее, сделав ее вход меньше, вы заставите шар вспыхивать медленнее. В данном примере масштаб будет изменен в 3 раза.

22. Задайте в качестве текста Expression следующие команды:

```
vector $pos = «ball.translateX, ball.translateY, ball.translateZ»;  
float $dist = mag( $pos );  
siren.colorR = siren.colorG = siren.colorB = (cos( $dist * 3 ) + 1) / 2;
```

23. Щелкните по кнопке Edit.**24. Щелкните по кнопке Play.**

Теперь движение шара сопровождается быстрыми вспышками.

Магнит

Это упражнение посвящено созданию объекта, обладающего магнитными свойствами и притягивающего другой объект, когда первый из них находится на определенном расстоянии от второго.

1. Откройте сцену **Magnet.ma**.

2. Щелкните по кнопке **Play**.

Вы увидите движущийся по сцене сферический объект **magnet**. Цилиндрический объект **metalObject** остается на месте. Требуется получить эффект притяжения объекта **metalObject** к сфере **magnet** при ее приближении,

3. Выделите **metalObject**.

4. Выберите пункт меню **Windows | Animation Editors | Expression Editor...**

5. Введите в поле **Expression Name** имя **Attractor**.

6. Щелкните по атрибуту **translateX** в списке **Attributes**.

7. Введите в поле **Expression** следующий текст:

```
vector $magPos = <<magnet.translateX,  
                    magnet.translateY,  
                    magnet.translateZ>>;  
  
vector $newPos = $magPos;  
metalObject.translateX = $newPos.x;  
metalObject.translateY= $newPos.y;  
metalObject.translateZ = $newPos.z;
```

Теперь положение **metalObject** в точности совпадает с позицией **magnet**. Сначала положение объекта **magnet** считывается из памяти и сохраняется в векторной переменной **\$magPos**. Затем описывается еще одна переменная **\$newPos**, которой присваивается значение вектора **\$magPos**. Позиция **metalObject** приравнивается к вектору **\$newPos**.

8. Щелкните по кнопке **Create**.

9. Щелкните по кнопке **Play**.

Объект **metalObject** в точности повторяет положение **magnet**. Хотелось бы добиться притяжения **metalObject** к объекту **magnet** лишь во время его нахождения на определенном расстоянии от последнего. **Если** расстояние велико, **metalObject** должен оставаться в исходном положении.

10. Введите в поле **Expression** следующий текст;

```
vector $startPos = <<4.267, 1, -0.134>>;
vector $magPos = <<magnet.translateX, magnet.translateY,
magnet.translateZ>>;
vector $newPos;
if( frame == 1 )
    $newPos = $startPos;
else
    $newPos = $magPos;
metalObject.translateX = $newPos.x;
metalObject.translateY = $newPos.y;
metalObject.translateZ = $newPos.z;
```

Начальное положение **metalObject** указано в окне **Channel Box** и введено вручную. Оно присваивается вектору **\$startPos**. Если текущим является первый кадр, то положение **metalObject** – вектор **\$newPos** – принимается равным **\$startPos**. В ином случае положение **metalObject** приравнивается к вектору **\$magPos**.

И. Щелкните по кнопке **Edit**.

12. Установите бегунок времени на первый кадр.

metalObject вернется в исходную позицию.

13. Установите бегунок времени на следующий кадр.

metalObject займет позицию **magnet**. Начальное и конечное положение объекта теперь определено, но как решить, когда он должен двигаться из одного положения в другое?

14. Введите в поле **Expression** следующий текст:

```
vector $startPos = <<4.267, 1, -0.134>>;
vector $magPos = <<magnet.translateX, magnet.translateY,
magnet.translateZ>>;
vector $newPos;
if( frame == 1 )
    $newPos = $startPos;
else
{
    vector $curPos = <<metalObject.translateX,
        metalObject.translateY,
```

```
metalObject.translateZ >>;  
float $dist = mag( $curPos - $magPos );  
if( $dist < 5 )  
    $newPos = $magPos;  
else  
    $newPos = $curPos;  
}  
metalObject.translateX = $newPos.x;  
metalObject.translateY = $newPos.y;  
metalObject.translateZ = $newPos.z;
```

Текущее положение **metalObject** считывается и сохраняется в переменной **\$curPos**. Далее при помощи Expression **mag(\$curPos - \$magPos)** рассчитывается расстояние от **metalObject** до **magnet**. Это длина вектора между двумя позициями. Если расстояние между ними меньше 5, объект **metalObject** занимает положение **magnet**, а иначе остается на месте.

15. Щелкните по кнопке **Edit**.

16. Щелкните по кнопке **Play**.

Вы почти у цели. Остается только одна проблема – цилиндр не «прилипает» к поверхности сферы, а фактически оказывается внутри нее. Коль скоро вы можете найти радиус сферы **magnet** и радиус цилиндра **metalObject**, то можете определить, где разместить **metalObject** так, чтобы он лишь касался **сферы**.

17. Введите в поле **Expression** следующий текст:

```
vector $startPos = <<4.267, 1, -0.134>>;  
vector $magPos = <<magnet.translateX, magnet.translateY,  
magnet.translateZ>>;  
vector $newPos;  
if( frame == 1 )  
    $newPos = $startPos;  
else  
{  
    vector $curPos = <<metalObject.translateX,  
                    metalObject.translateY,  
                    metalObject.translateZ>>;
```

```
float $dist = mag( $curPos - $magPos );
if( $dist < 5 )
{
    $offset = makeNurbCylinder1.radius + makeNurbSphere1.radius;
    $newPos = <<$magPos.x + $offset, $magPos.y, $magPos.z>>;
}
else
    $newPos = $curPos;
}
metalObject.translateX = $newPos.x;
metalObject.translateY = $newPos.y;
metalObject.translateZ = $newPos.z;
```

Расстояние от магнита, на которое должен быть помещен объект **metalObject**, рассчитывается в векторе с именем **\$offset**. Положение **metalObject** представляет собой смещение на эту величину по оси **x**.

18. Выделите объект **metalObject**.

19. В списке **Inputs** (Входы) окна **Channel Box** щелкните по элементу **makeNurbSphere1**.

20. Перейдите к тому кадру, где **metalObject** притягивается к объекту **magnet**.

21. Измените в диалоговом режиме значение атрибута **Radius**.

Со сменой радиуса **metalObject** остается «при克莱енным» к сфере **magnet**. Если радиус увеличить настолько, что **metalObject** окажется на расстоянии более 5 единиц от магнита, он перестанет к нему «приклеиваться». Если вы хотите исправить эту ситуацию, модифицируйте текст Expression, заменив расчет расстояния между центрами объектов расчетом расстояния между их поверхностями.

3.7.2. Рекомендации по составлению выражений

Как только вы поймете принципы действия и создания Expression, ваша работа, на самом деле, будет заключаться в составлении формул или уравнений, которые нужны для анимации атрибутов тем или иным образом. К счастью, Expression позволяет обращаться почти ко всем командам и процедурам языка MEL. Вы даже можете описывать и применять свои собственные процедуры. В этом разделе более подробно описано применение механизма Expression, включая связанные с ним ограничения и предостережения, о которых следует знать.

Только присваивание атрибутов

Цель Expression - установить значения атрибута при помощи нескольких операторов языка MEL. Ничего больше в Expression делать нельзя. В частности, пользуясь Expression, вы не должны явно или неявно изменять топологию **Dependency Graph**: создавать или разрывать соединения, создавать или удалять узлы и т. д. Подобные действия переводят **Dependency Graph** в противоречивое состояние, А значит, необходимо избегать любых команд **MEL**, существенно изменяющих граф зависимости.

Запустив Expression, вы не сможете отменить ни одну из команд, которые это выражение содержит. Вслед за выполнением обычного сценария MEL его можно сразу же отменить, если это необходимо. Выражения Expression могут запускаться когда угодно. Если вы запросите атрибут, которым управляет Expression, это выражение будет запущено. Если вы передвинете бегунок времени, выражение Expression тоже будет запущено. Для Expression не существует шаблона «начало, выполнение, конец», а значит, отменить действие выражения невозможно.

Будь у вас выражение Expression, которое создает новую сферу в каждом кадре, простая перемотка к началу не привела бы к удалению всех ранее созданных сфер и возврату сцены в исходное состояние. Результатом повторного воспроизведения анимации станет создание средствами Expression новой серии сфер. Аналогично, выражение Expression, которое удаляет объекты, не восстанавливает их при перемотке анимации. По этой причине избегайте любых команд MEL, предназначенных для создания или удаления объектов.

Надлежит пользоваться только теми командами **MEL**, которые не изменяют **Dependency Graph** ни напрямую, ни косвенно. Описание команды должно ясно говорить о том, так это или нет. При отсутствии ясности сохраните сцену прежде, чем дать добро на выполнение потенциально опасного элемента Expression.

Поддерживаемые типы атрибутов

В тексте выражений Expression можете обращаться только к атрибутам типа float, int или boolean. Если же, к примеру, вы попытаетесь обратиться к атрибуту string, matrix или любого другого типа, произойдет ошибка. Так, следующее выражение Expression станет причиной вывода на экран сообщения об ошибке:

```
$wm = obj.worldMatrix;
```

Выражения Expression не принимают во внимание неявные минимальные и максимальные значения атрибута, которые применяются лишь при работе с пользовательским интерфейсом. Фактически атрибуту можно присвоить любое значение

ние с плавающей запятой. В то же время большинство атрибутов обладают вполне определенным допустимым диапазоном, К примеру, задание числа 10 как значения атрибута **cotorR** будет некорректным, поскольку диапазон значений цветовых компонентов лежит между 0 и 1. Важно помнить об этом, потому что установка значения атрибута вне пределов допустимого диапазона ведет к появлению сообщения об ошибке.

Не исключено, что при обращении к тому или иному объекту вам понадобится использовать полный путь по ОАГ. Это особенно важно в тех случаях, когда два объекта сцены обладают одинаковыми именами, но имеют разных родителей. Возьмем для примера сцену с двумя объектами **box**. Первый - потомок узла преобразования **objA**, второй - потомок узла преобразования **objB**. Полные пути к ним, соответственно, имеют вид **objA|box** и **objB|box**. Использование следующего выражения Expression станет причиной ошибки.

```
box.translateX = 1;
```

```
// Error: More than one attribute name matches.  
Must use unique path name: box.translateX //  
// Ошибка: Обнаружено соответствие более чем одного имени атрибута.  
Требуется уникальное путевое имя: box.translateX //
```

Во избежание подобных неопределенностей необходимо указать полный путь, ведущий к объекту **box**. Чтобы исправить положение, перепишите Expression следующим образом:

```
objA|box.translateX = 1;
```

Обратите внимание на отсутствие пробелов перед вертикальной чертой (|) и после нее.

Избегайте команд `setAttr` и `getAttr`

Для получения и установки значения атрибута нужно использовать команды `getAttr` и `setAttr`. Следующие операторы, к примеру, задают, а затем считывают атрибут **box.scaleX**.

```
setAttr box.scaleX (rand(100) * 0.5);  
float $sx = `getAttr box.scaleX`;
```

Ради удобства вы можете считывать и задавать атрибуты в Expression напрямую, пользуясь оператором доступа к членам данных (.). Вспомните, это похоже на обращения к компонентам вектора *x*, *y*, *z*. Пользуясь этим оператором в Expression, можно переписать приведенный код так:

```
box.scaleX = rand() * 0.5;  
float $sx = box.scaleX;
```

В действительности применение команд `setAttr` и `getAttr` в Expression может привести к неожиданным результатам. Это связано с тем, что обе команды способны игнорировать обычный механизм расчета графа **Dependency Graph**. Вызов этих команд из Expression даст неопределенный результат. По сути, следует полностью избегать как той, так и другой команды.

Атрибуты в соединениях

Если в выражении Expression вы попытаетесь задать значение атрибута, являющегося целевым атрибутом соединения, возникнет ошибка. Например, если в следующем выражении Expression атрибут `scaleX` уже анимирован (управляется узлом анимационной кривой), вы увидите сообщение об ошибке.

```
sphere.scaleX = 23;
```

```
// Error: Attribute already controlled by an expression,  
// keyframe, or other connection: sphere.scaleX //  
// Ошибка: Атрибут уже управляет выражением,  
// ключевым кадром или другим соединением: sphere. scaleX //
```

Чтобы исправить ситуацию, нужно, прежде всего, разорвать соединение с атрибутом `scaleX`. Для этого вызовите команду `disconnectAttr`. Когда соединение будет разорвано, Expression сможет свободно управлять этим атрибутом.

Автоматическое удаление

В некоторых ситуациях выражение Expression может быть удалено даже без вашего ведома. При его построении Maya, на самом деле, создает узел *выражения*. Это узел Dependency Graph, соединенный с теми атрибутами, которыми управляет Expression. Фактически ваше выражение содержится в узле категории `expression`. Maya может решить, что узел `expression` следует удалить, если сочтет, что он уже не используется. Скажем, у вас была сцена с объектом `box` и вы создали следующее выражение Expression:

```
box.translateX = 2;
```

Если затем вы удалите объект `box`, Maya удалит и связанный с ним узел `expression`, а стало быть, и ваше выражение Expression тэже. Причина этого состоит в том, что поскольку объект `box` удален, узел `expression` не содержит никаких входящих соединений. Узел `expression`, как и многие другие узлы, спроектирован с расчетом на автоматическое самоуничтожение в подобной ситуации.

ции. Если же у вас, по аналогии, есть выражение Expression, которое ссылается на ряд объектов, и все они удалены, узел **expression** удаляется **тоже**. коль скоро все его выходные соединения разорваны. Узлы анимационных кривых, к примеру, автоматически удаляют себя, когда они перестают использоваться для анимации хотя бы одного атрибута.

Цель подобного поведения - предотвращение существования «висячих» узлов. «Висячими» называют узлы, которые не имеют ни одного входного или выходного соединения. Они просто входят в состав сцены, но ничего не делают.

В большинстве случаев автоматическое удаление не вызывает никаких проблем, так как Expression уже не оказывает никакого влияния. Однако если вы хотите предотвратить какие бы то ни было проблемы, составьте Expression так, чтобы это выражение управляло *пустым* атрибутом узла, который, как вы знаете, удален не будет. Создайте, например, групповой узел **dummyNode**, а затем добавьте к нему атрибут **dummy** типа **float**:

```
group -empty -name dummyNode;  
addAttr -shortName dmy -longName -dummy  
    -attributeType "float" dummyNode;
```

Наконец, задайте в качестве Expression следующее выражение:

```
dummyNode.dummy = 1;  
... // Прочие операторы выражения
```

До тех пор пока существует объект **dummyNode** и связанный с ним атрибут **dummy**, ваше выражение Expression не подвергнется автоматическому удалению.

3.7.3. Отладка выражений

С учетом того, когда и как происходит вычисление выражений Expression, они имеют некоторые особенности и ограничения, связанные с их записью и отладкой. Данный раздел посвящен некоторым из этих вопросов.

Обнаружение ошибок

Если выражение Expression, записанное в редакторе **Expression Editor**, содержит синтаксическую ошибку, вы не увидите ее детального описания. Чтобы познакомиться с подробной информацией об ошибке, откройте **Script Editor**. Также обратите внимание на необходимость проверки наличия ошибок при изменении работающего выражения Expression. Если вновь сделанные изменения ведут к ошибке, новое выражение использоваться не будет. Вместо него будет по-прежнему существовать предыдущее выражение, и именно его результат

станет результатом вычислений. Новое выражение заменяет предшествующее лишь тогда, когда оно свободно от синтаксических ошибок. Если вы изменили выражение, а оно работает по-старому, выполните его проверку.

Трассировка

Так как Expression автоматически вычисляется в тот момент, когда требуется обновить затрагиваемые им атрибуты, вы лишены возможности напрямую контролировать время его выполнения. Кроме того, в отличие от команд, запущенных из строки **Command Line** или редактора **Script Editor**, результаты этих выражений не отображаются на экране. Лучший метод запуска Expression на выполнение в режиме трассировки – применение операторов `print`. Пользуйтесь командой `print` для вывода информации о текущем выполняемом операторе и текущих значениях переменных.

Если, например, вы создали такое выражение **Centered**, которое помещает текущий объект в центр между двумя другими, то трассировку процесса его выполнения, а также отслеживание его переменных можно выполнить так:

```
print("\nExecuting Centered at frame " + frame); // "Выполнение
Centered: кадр "
obj.translateX = objA.translateX + 0.5 * (objB.translateX -
    objA.translateX);
obj.translateY = objA.translateY + 0.5 * (objB.translateY -
    objA.translateY);
obj.translateZ = objA.translateZ + 0.5 * (objB.translateZ -
    objA.translateZ);
print (" x=" + obj.translateX + " y=" + obj.translateY + " z=" +
    obj.translateZ);
```

При анимации или перемещении объекта `objA` либо `objB` Expression будет автоматически выполняться. На экране появится обновленная позиция объекта `obj`:

```
Executing Centered at frame 8 x=-12.90351648 y=-0.6519004166
z=1.691102353
Executing Centered at frame 8 x=-12.97169705 y=-0.6322368845
z=1.677296334
Executing Centered at frame 8 x=-13.07624005 y=-0.5929098204
z=1.643959307
Executing Centered at frame 8 x=-13.2126012 y=-0.5535827562 z=1.61634727
```

Чтобы полностью увидеть результаты работы операторов print, откройте редактор **Script Editor**. В ряде случаев вам захочется отключить трассировку, поэтому для управления этой возможностью вы можете описать глобальную переменную.

```
global int $doTracing = false;
```

Тогда в своих выражениях Expression вы можете выводить сообщения с учетом установки флага \$doTracing.

```
if( $doTracing )
    print ("\nExecuting Centered at frame " + frame);
obj.translateX = objA.translateX + 0.5 * (objB.translateX -
    objA.translateX);
obj.translateY = objA.translateY + 0.5 * (objB.translateY -
    objA.translateY);
obj.translateZ = objA.translateZ + 0.5 * (objB.translateZ -
    objA.translateZ);
if( $doTracing )
    print (" x=" + obj.translateX + " y=" + obj.translateY +
        " z=" + obj.translateZ );
```

Отключение

Занимаясь отладкой нескольких выражений Expression, часто бывает полезно отключить некоторые из них, чтобы полностью сконцентрироваться на оставшихся. Чтобы отключить выражение Expression, откройте его в редакторе **Expression Editor**, а затем снимите флажок **Always Evaluate** (Вычислять всегда). Expression будет по-прежнему существовать, но лишь перестанет вычисляться.

Разорванные соединения

Выражение Expression может получать входную информацию от атрибута одного объекта и управлять атрибутом другого. Вероятно, что входной объект по каким-то причинам был удален. В этом случае правильное вычисление Expression становится невозможным. Если выражение Expression пользуется атрибутом, который был удален, Maya покажет это в тексте выражения посредством специальной нотации. Возьмем для примера следующее выражение Expression:

```
obj.translateX = box.scaleX;
```

Положим, что объект box был удален. Maya автоматически изменит это выражение так:

```
obj.translateX = .I[0];
```

Если атрибут утерян, Maya меняет его имя на `.I[x]`, где x - внутренний индекс атрибута. `.I` является признаком *входного атрибута*. Иначе говоря, это был атрибут, *служивший* входом выражения. Есть и *выходные атрибуты*. Это те атрибуты, что управляются из Expression. Если бы в предыдущем примере вместо box был удален объект `obj`, Maya переписала бы выражение так:

```
.0[0] = box.scaleX;
```

О указывает, что на сей раз утерян выходной атрибут. В этих ситуациях вы должны вручную изменить выражение Expression так, чтобы оно ссылалось на допустимый атрибут.

Переименование объектов

Maya автоматически обрабатывает такие ситуации, в которых объект, на который вы ссылаетесь из Expression, меняет свое имя. Вместе с тем вывод обновленного имени в выражении Expression, показанном в редакторе Expression Editor, может потребовать повторной загрузки. Для этого просто щелкните по кнопке Reload (Повторить загрузку). На экране появится Expression с новым именем объекта.

3.7.4. Выражения для работы с частицами

Частица - это всего лишь точка. Целью анимации частиц нередко становится их перемещение. Каждая частица имеет свою позицию, скорость и ускорение. Кроме того, она может обладать массой. Хотя знание физики и понимание принципов того, как силы воздействуют на тела, значительно поможет при написании выражений для анимации частиц, управлять ими можно, имея лишь базовые познания в области математики.

1. Откройте сцену ParticleGrid.ma.
Сцена состоит из массива неподвижных частиц.
2. Выделите объект particles.
3. Откройте редактор Attribute Editor.
4. Щелкните по вкладке **particlesShape**.
5. Выполнив прокрутку, найдите раздел Per Particle (Array) Attributes [Атрибуты (массивы) уровня частиц].

6. Щелкните правой кнопкой по полю, расположенному рядом с **position**, а затем выберите из выпадающего меню пункт **Creation Expression...** (Выражение времени создания...).

7. Задайте в качестве текста **Expression:** следующее:

```
particlesShape.position = <<1,0,0>>;
```

8. Щелкните по кнопке **Create**.

9. Перейдите к первому кадру.

Кажется, все частицы пропали. На деле произошло нечто иное: теперь все они имеют позицию $(1, 0, 0)$.

Заметьте: опция **Particle:** (Частица) в редакторе **Expression Editor** имеет значение **Creation** (Создание). Только что вы описали *выражение времени создания* для частиц. Выражение создания вызывается в первом кадре. Оно служит для установки начальных параметров всех частиц. В данном примере их положение было задано одинаковым образом.

Чтобы увидеть результат выражения создания, вам надо находиться в первом кадре.

10. Задайте в качестве текста **Expression** следующее:

```
particlesShape.position = sphrand( <<4, 0, 4>> );
```

11. Щелкните по кнопке **Edit**.

Теперь частицы разместились по кругу. Функция **sphrand** выдает случайные точки, не выходящие за пределы сферы. Ее протяженность, т. е. ширина (*x*), высота (*y*) и глубина (*z*), описывается вектором, передаваемым команде **sphrand**. В данном случае сфера сжата до плоского круга, так как радиусы по осям *x* и *z* равны 4, а радиус по оси *y* равен 0.

12. Перейдите к любому другому кадру.

13. **Вернитесь** к первому кадру.

Положение частиц изменилось. Оно станет меняться каждый раз, когда вы будете возвращаться к первому кадру. Несмотря на желание добиться случайного положения частиц, вы не хотите, чтобы при переходе к начальному **кадру** оно всякий раз было абсолютно иным. При первом своем отображении частицы будут занимать одно множество точек. Если позднее вы снова перейдете к первому кадру, частицы получат значения из нового набора позиций. Первое изображение не совпадает со вторым, **хотя** на экран выводится один и тот же кадр. Стоит вам заняться многослойными комбинированными изображениями, и это просто перестанет работать. **Очень** важно, чтобы при инициализации частиц использовалось постоянное множество случайных позиций.

Даже несмотря на то что функция `sphrand` выдает совершенно случайные числа, это делается на основе некоторого исходного значения. Оно служит для первичной установки внутреннего генератора `случайных` чисел, который используется командой `sphrand`. Обычно такое значение называют `начальным числом`. Поэтому все результаты, выдаваемые `sphrand`, рассчитываются на основе этого начального числа. Установив его, вы можете гарантировать, что `sphrand` выдает неизменную серию случайных значений. Для установки начального числа предназначена команда `seed`.

Отсюда логически следует, что `seed` должна вызываться непосредственно перед командой `sphrand`. К сожалению, выражение `создания` вызывается по одному разу для каждой частицы, а значит, команда `seed` будет одинаковым образом вызываться для всех частиц. В действительности хотелось бы, чтобы инициализация `seed` происходила именно для всех частиц сразу. Коль скоро это невозможно в выражении создания, команда `seed` вызывается иным способом.

14. Задайте в качестве текста Expression следующее:

```
seed( particleId );
particlesShape.position = sphrand( <<4, 0, 4>> );
```

15. Щелкните по кнопке Edit,

16. Перейдите к любому другому кадру.

17. Вернитесь к первому кадру.

Теперь положение частиц носит случайный характер, однако их расположение в первом кадре остается `постоянным`. `particleId` - это уникальный индекс, который присвоен каждой отдельной частице. Выражение создания запускается по разу для каждой из частиц. По сути, вызову этого выражения предшествует обновление индекса `particleId`, принимающего значение идентификатора текущей частицы. Используя в генераторе случайных чисел начальное число на основе такого идентификатора, вы можете гарантировать, что вызов `sphrand` будет инициализирован одним и тем же значением.

С помощью `particleId` можно размещать частицы и более структурированным образом.

18. Введите в поле Expression следующий текст:

```
int $nRows = sqrt(particlesShape.count);
float $spacing = 0.5;
float $x = (particlesShape.particleId \ $nRows) * $spacing;
float $z = trunc(particlesShape.particleId / $nRows) * $spacing;
particlesShape.position = <<$x, 0, $z>>;
```

19. Щелкните по кнопке Edit.

Теперь частицы равномерно распределены по сетке. Первый шаг этой операции состоит в определении фактического количества частиц. Это значение хранится в принадлежащем **particlesShape** атрибуте **count**. Извлечение квадратного корня из количества частиц позволяет получить число строк и столбцов, необходимых для создания расстановки в форме квадрата, т. е. для размещения $\$nRows * \$nRows$ частиц. Переменная **\$spacing** определяет расстояние между частицами.

Позиции **\$x** и **\$z** рассчитываются на основе идентификатора частицы. Коль скоро тот находится в диапазоне от 0 до **count-1**, вы можете поделить этот диапазон на полоски, каждая из которых имеет длину **\$nRows**. Далее, для определения окончательного положения точки результат умножается на коэффициент **\$spacing**.

Наряду с выражениями времени создания, частицы поддерживают и *выражения времени выполнения*. Значение такого выражения рассчитывается во всех кадрах за исключением начального. Таким образом, в начальном кадре вычисляется выражение времени создания, а во всех более поздних (> начального кадра) - выражение времени выполнения.

20. Щелкните по переключателю Runtime (Время выполнения) рядом с приглашением Particle:

На экране появится пустое поле **Expression**. Ваше выражение создания по-прежнему находится там. Щелкая по соответствующим переключателям, вы можете переходить от выражения времени создания к выражению времени выполнения и обратно. Форма, где хранятся частицы, содержит и выражение времени создания, и выражение времени выполнения, так что вам не придется создавать соответствующие выражения Expression отдельно.

21. Введите в поле Expression следующий текст:

```
particles.velocity = <<0, 1, 0>>;
```

22. Щелкните по кнопке Edit.**23. Щелкните по кнопке Play.**

Частицы начнут постепенное движение вверх. Регулируя скорость частиц, можно заставить их двигаться в определенном направлении. В данном случае каждая из них смещается в каждом кадре на 1 единицу вдоль оси **y**.

Заметьте, что необходимости в создании нового выражения Expression для изменяемого нами атрибута **velocity** не было. Узел формы с частицами включает только одно выражение времени создания и одно выражение времени выполнения. В отличие от обычных выражений Expression, работая с кото-

рыми вы можете задать одно такое выражение в расчете на атрибут, выражения **времени** создания и времени выполнения, записанные для частиц, позволяют обращаться ко всем их атрибутам.

24. Введите в поле **Expression** следующий текст:

```
vector $pos = particlesShape.position;  
particlesShape.velocity = <<0, sin( $pos.x ),0>>;
```

25. Щелкните по кнопке **Edit**.

26. Щелкните по кнопке **Play**.

Теперь частицы движутся вверх и вниз, будто лежат на поверхности ряда продольных волн.

27. Введите в поле Expression следующий текст:

```
vector $pos = particlesShape.position;  
particlesShape.velocity = <<0, sin( mag($pos) ),0>>;
```

28. Щелкните по кнопке **Edit**.

29. Щелкните по кнопке **Play**.

Теперь частицы движутся **вверх** и вниз, будто лежат на поверхности ряда радиальных волн.

Далее мы создадим объект, который при прохождении вблизи частицы вызовет ее толчок в направлении снизу-вверх.

30. Сверните окно **Expression Editor**.

31. Выберите в главном меню пункт **Create | Locator** (Создать | Локатор).

32. Назовите узел преобразования локатора именем **updraft**.

33. Анимируйте узел **updraft** по осям **x** и **z** так, чтобы он проходил через облако из частиц.

34. Разверните окно **Expression Editor**.

35. Выберите в меню пункт **Select Filter | By Expression Name**.

36. Щелкните по выражению **particleShape**.

37. Введите в поле **Expression** следующий текст:

```
vector $draftPos = <<updraft.translateX, updraft.translateY,  
updraft.translateZ>>;  
vector $partPos = particlesShape.position;  
float $dist = mag( $partPos - $draftPos );  
vector $v = particlesShape.velocity;
```

```
if( $dist <= 2 )  
    $v=<<0,2,0>>;  
    particlesShape.velocity = $v;
```

38. Щелкните по кнопке **Edit**.

39. Щелкните по кнопке **Play**.

Каждая из частиц, находящихся на расстоянии не более двух единиц от локатора **updraft**, отлетает по воздуху вверх. Ради достижения этого эффекта ее скорость принимается равной вектору (0, 2, 0). Если же частица находится на расстоянии, превышающем 2 единицы, она просто сохраняет свою прежнюю скорость.

Вычисление выражений времени создания и времени выполнения

Выражение создания вычисляется тогда, когда текущее значение времени не превышает приписанного частице времени **startFrame**. Таким образом, если значение **startFrame** для данной частицы равно 10, то выражение создания рассчитывается для нее во всех кадрах с 0-го по 10-й. В 11-м и всех последующих кадрах используется выражение времени выполнения. Его значение всегда рассчитывается по истечении времени **startFrame**.

Точнее, выражение времени создания вычисляется для каждой частицы, чей атрибут **age** (возраст) равен 0. Если атрибут **age** не равен 0, то применяется выражение времени выполнения. Атрибут **age** – это доступный только для чтения массив вещественных чисел двойной точности. Именно здесь Maya хранит текущий возраст каждой отдельной частицы. Так как частицы могут рождаться в любое время, их возраст может быть неодинаковым. Выражение создания, по сути, рассчитывается для каждой отдельной частицы, возраст которой равен 0. Для частиц ненулевого возраста рассчитывается выражение времени выполнения.

Возраст каждой частицы **можно**, к счастью, вывести на экран. Выбрав **particleShape**, откройте редактор **Attribute Editor**. Переходя к разделу **Render Attributes** (Отображать атрибуты), установите **Particle Render Type** (Тип отображения частиц) в значение **Numeric** (Числовой). Щелкните по кнопке **Current Render Type** (Текущий тип отображения). Установите поле **Attribute Name** в значение **age**. Теперь каждая частица будет показывать свой нынешний возраст.

Атрибуты уровня объектов и отдельных частиц

Важно понимать тот факт, что в действительности частицы хранятся в виде узла **shape**, который имеет тип **particle**. Коль скоро узел **shape** подобен любым другим, то все его атрибуты, как вы и были вправе ожидать, можно анимировать средствами Expression. Частицы, однако отличаются тем, что можно составить выражение Expression не только для управления узлом **shape** в целом, но и **каждой**

дой отдельной частицей. По сути дела, имеется различие между атрибутами, которые совместно используются всеми частицами, и атрибутами, которыми наделена каждая частица в отдельности. Атрибут, общий для всех частиц, носит название *атрибута уровня объекта*. Его изменение затрагивает каждую частицу. Атрибут, позволяющий определять значения для конкретных частиц, называется *атрибутом уровня частицы*.

1. Откройте сцену **ParticleGrid.ma**.
2. Выберите пункт **Shading > Smooth Shading AИ** (Тонировка | Гладкая тонировка всюду).
3. Выделите объект **particles**.
4. **Откройте редактор Attribute Editor.**
5. Выберите вкладку **particlesShape**.
6. Выполните прокрутку вниз до раздела **Add Dynamic Attributes** (Добавить динамические атрибуты).
7. Щелкните по кнопке **Color** (Цвет).

8. Установите опцию **Per Object Attribute** (Атрибут уровня объекта), затем щелкните по **Add Attribute** (Добавить атрибут).

В разделе **Render Attributes** имеются три дополнительных атрибута [**Color Red** (Красный цвет), **Color Green** (Зеленый цвет), **Color Blue** (Синий цвет)].

9. Установите атрибут **Color Red** в 1.
Цвет всех **частиц** станет красным. Вновь добавленный цветовой атрибут является атрибутом уровня объекта, поэтому он применяется ко всем частицам одновременно. Его изменение оказывает влияние на каждую частицу. При желании управлять цветом частиц по **отдельности** вам нужно создать цветовой атрибут уровня частицы.
10. В разделе **Add Dynamic Attributes** щелкните по кнопке **Color**.
11. Установите опцию **Per Particle Attribute** (Атрибут уровня частицы), затем щелкните по **Add Attribute**.
Теперь в разделе **Per Particle (Array) Attributes** присутствует атрибут **rgbPP**.
12. Щелкните правой кнопкой по полю, расположенному рядом с **rgbPP**, а затем выберите из выпадающего меню пункт **Creation Expression...**
13. Задайте в качестве текста **Expression** следующее:
`particlesShape.gdBPP = <<rand(1), rand(1), rand(1)>>;`
14. Щелкните по кнопке **Create**.

Цвет каждой частицы становится случайным. Так как созданный вами атрибут **rgbPP** является атрибутом уровня частицы, он автоматически перекрывает атрибуты уровня объекта: **colorRed**, **colorGreen**, **colorBlue**. Для удаления атрибута уровня объекта или уровня частицы просто воспользуйтесь функцией **Delete Attributes** (Удалить атрибуты).

15. Щелкните по кнопке **Select** в нижней части редактора **Attribute Editor**.

Узел **particlesShape** будет выделен.

16. Выберите в строке меню **Attribute Editor** пункт **Attributes | Delete Attributes...** (Атрибуты | Удалить атрибуты...)

На экране появится перечень вновь добавленных атрибутов.

17. Выберите атрибут **rgbPP**.

18. Щелкните по кнопке **OK**.

Атрибут уровня частиц удален. Частицы снова отображаются красным цветом, поскольку атрибут уровня частиц их уже не перекрывает. Цветовые атрибуты уровня объекта **снова** в силе. Их также можно удалить из узла, воспользовавшись пунктом **Delete Attributes**.

При начальной организации формы с частицами Maya создает атрибуты, соответствующие лишь наиболее распространенным свойствам частиц. Далее по мере необходимости вы должны самостоятельно добавлять другие атрибуты уровня объектов и отдельных частиц.

19. В разделе **Add Dynamic Attributes** щелкните по кнопке **General** (Общие).

20. Щелкните по вкладке **Particles**.

В список войдут все атрибуты уровня частиц, которые можно добавить в узел. Обратите внимание на присутствие в конце списка атрибутов **userScalarPP** и **userVectorPP**. Вы можете пользоваться ими для описания своих собственных атрибутов, которые хотите связать с частицами.

Компоненты векторов

Несмотря на то что выражения могут свободно считывать и устанавливать векторные значения, они не способны считывать и устанавливать значения отдельных *компонентов* векторов. Вы, должно быть, заметили, что в предыдущем примере атрибут **position** узла **particleShape** был задан следующим образом:

particlesShape.position = <<1, 0, 0>>;

Если атрибут **position** – это вектор, что произошло бы при обращении к компонентам векторах *x*, *y* и *z* напрямую?

particlesShape.position.x = 1;

Результатом стала бы такая ошибка:

```
// Error: Attribute not found or variable missing
'$': particlesShape.position.x //
// Ошибка: Не найден атрибут или пропущен знак '$' в имени
переменной: particlesShape.position.x //
```

Следующий оператор также вызывает аналогичную ошибку:

```
float $x = particlesShape.position.x;
```

Причина неработоспособности каждой команды состоит в том, что оператор «точка» (.) служит для доступа как к атрибутам узлов, так и к компонентам векторов. При использовании оператора «точка» обращение к **атрибуту** имеет приоритет над обращением к компоненту вектора, поэтому в предыдущем операторе Maya пытается найти атрибут с именем x в составе атрибута position, но не пытается обратиться к входящему в него компоненту x. Если вы хотите получить доступ к любому **компоненту** векторного атрибута, сначала поместите значения этого атрибута в локальную переменную **векторного типа**.

```
vector $v = particlesShape.position;
```

Теперь Maya знает о неявном обращении к вектору, поэтому оператор «точка» (.) может свободно использоваться для доступа к его **компонентам**.

```
float $x = $v.x;
```

Аналогично, присваивая значение векторной переменной, вам не удастся установить только один ее компонент. Допустимыми являются лишь присваивания полных векторов, а значит, для изменения **x-компонента** позиции вам придется поступить так:

```
vector $v = particlesShape.position;
$v.x = 34;
particlesShape.position = $v;
```

3.7.5. Выражения повышенной сложности

В этом разделе описаны более сложные примеры применения выражений Expression.

Использование выражений при анимации по ключевым кадрам

Как поступить, если вы захотите создать анимацию, в которой анимированные вручную ключевые кадры будут сочетаться с выражениями Expression? Один из таких примеров- реалистичная анимация самолета. Непосредственное управле-

ние общим перемещением объекта будет дополнено некоторой турбулентностью. Управление объектом можно реализовать путем обычной установки ключевых кадров, а турбулентность добавить с использованием Expression. К сожалению, Maya не предусматривает механизмов непосредственного сочетания этих двух технологий, поэтому взамен вам придется воспользоваться косвенными методами.

1. Откройте сцену **Plane.ma**.

2. Щелкните по кнопке **Play**.

Перед вами пролетит **небольшой** самолет, который затем скроется за пределами экрана.

3. Выберите пункт меню **Window | Outliner....(Окно | Схема сцены...)**

4. В списке щелкните по элементу **plane**.

5. Выведите на экран окно **Channel Box**.

Channel Box показывает, что параметры трансляции и поворота объекта **plane** уже **анимированы**. К этой анимации нужно добавить небольшую автоматическую турбулентность по вертикали. Более того, хотелось бы получить возможность вернуться назад, изменить исходную анимацию по ключевым кадрам и добиться автоматического обновления турбулентности.

6. Откройте редактор **Attribute Editor**.

7. Щелкните правой кнопкой по прямоугольнику **translateY**.

Появится небольшое всплывающее меню. Из него видно, что с этим атрибутом соединен атрибут **plane_translateY.output**. Атрибут **translateY** требует управления через Expression. Выражение Expression будет принимать на вход исходную анимацию и вносить в нее легкие возмущения, имитируя действие турбулентности. Соединение между текущей кривой анимации и атрибутом **translateY** должно быть разорвано.

Если вы удалите все выходные соединения узла анимации, он, к сожалению, будет автоматически **уничтожен**. Изменить его поведение невозможно. Чтобы **предотвратить** удаление, на выходе узла должно всегда присутствовать одно соединение. Создайте в узле **plane** новый атрибут и, прежде чем разорвать исходное соединение, **установите** его связь с анимацией. Это послужит гарантией **того**, что узел анимации не будет автоматически удален, поскольку он уже будет содержать вновь сформированное соединение.

8. Откройте редактор **Script Editor**.

9. Выполните следующее:

```
addAttr -longName "animTranslateY" -attributeType "float" plane;
```

Атрибут **animTranslateY** добавится к узлу **plane**.

10. Выполните следующее:

```
connectAttr plane_translateY.output plane.animTranslateY;  
disconnectAttr plane_translateY.output plane.translateY;
```

Выход анимации соединится с новым атрибутом **animTranslateY**.

И. Выполните следующее:

```
expression -string "plane.translateY = plane.animTranslateY"  
-name "Turb";
```

Будет создано выражение с именем **Turb**. В соответствии с ним описывающий самолет атрибут **translateY** принимается равным новому атрибуту **с именем, animTranslateY**. Так как исходная кривая анимации подается на вход последнего, то атрибут самолета **translateY** теперь косвенно принимает **анимированные** значения.

12. Щелкните по кнопке **Play**.

Самолет движется точно так же, однако окончательное значение **translateY** теперь определяется выражением, а не анимационной кривой. Выражение напрямую возвращает результат, полученный от кривой анимации. Теперь внесем легкую турбулентность.

13. В редакторе **Script Editor** выполните следующую команду:

```
expression -edit -string "plane.translateY = plane.animTranslateY +  
noise( plane.translateX / 3, plane.translateZ / 3 );" Turb;
```

Высота полета самолета рассчитывается теперь на основе исходной анимации, а также небольшой турбулентности, возникающей благодаря использованию команды **noise**. При вызове этой команды ей передаются параметры, равные одной трети координат самолета по осям **x** и **z**.

14. Щелкните по кнопке **Play**.

На сей раз за счет турбулентности самолет получает изрядную порцию встряски. Развивая этот подход, можно добиться автоматического нанесения удара по самолету с учетом данных о положении зенитного орудия.

Итак, конфигурирование объекта завершено, и теперь вы можете изменять анимацию самолета по ключевым кадрам, обеспечивая автоматическое внесение турбулентности.

4. C++ API

4.1. Введение

Язык MEL, безусловно, предоставляет очень мощные и эффективные средства, которые как автоматизируют решение задач в среде Maya, так и упрощают процесс решения. В нем, возможно, содержится весь набор программируемых функций, которые понадобятся вам когда-либо. Однако в том случае, если вам потребуется еще больше возможностей доступа к элементам Maya и управления ими, вы можете обратиться к интерфейсу C++ API. С его помощью вы сможете создавать свои собственные, нестандартные узлы **Dependency Graph**. Они получат возможность непосредственно встраиваться в Maya и бесконфликтно работать со всеми другими узлами. Это позволит вам напрямую внедрять свои функции в самое ядро Maya.

Нельзя пренебрегать и другим преимуществом C++ API - разработанные вами подключаемые модули будут нередко работать намного быстрее аналогичных сценариев MEL. Эти модули проходят стадию компиляции и сборки, а потому результат будет полностью оптимизирован под целевую платформу. Необходимость в интерпретации «на лету», как это было в случае с MEL, отсутствует, и это делает ваш продукт гораздо быстрее.

Общеизвестно, что применение всех возможностей интерфейса C++ API требует хорошего знания языка C++, хотя начать написание некоторых модулей Maya можно и в том случае, если вы имеете лишь минимальные познания в C++.

Вот неполный перечень всех элементов и функций Maya, расширяемых посредством API-интерфейса.

- **Команды**

При помощи API вы сможете создавать пользовательские команды, которые обладают всем набором возможностей собственных встроенных команд Maya, включая меню/повторное выполнение, справку, полный доступ к сцене и любое сочетание командных аргументов. Коль скоро такая команда написана на C++, вы вправе использовать в ней любые функции языка, в том числе внешние библиотеки. В этом наблюдается их резкое отличие от процедур на языке MEL, которые допускают вызов лишь дру-

гих процедур MEL, а также команд. Ваши собственные команды будут обрабатываться точно так же, как и встроенные команды Maya. По сути, вы сможете вызывать их из любого оператора языка MEL.

◆ Узлы графа зависимости

Интерфейс C++ API позволяет создавать нестандартные узлы графа зависимости. Вы можете организовать очень простой узел DG, предоставляющий лишь такие базовые возможности, как сложение двух точек, или очень сложный узел, реализующий анимацию целого персонажа. Когда ваш узел пройдет регистрацию в Maya, вы сможете создавать, удалять и редактировать его подобно всякому стандартному узлу. Прочие узлы могут участвовать в соединениях с ним, а его выходы быть связаны с другими узлами. Ваши узлы могут всецело и бесконфликтно встраиваться в среду Maya.

Наряду с базовыми узлами DG, вы можете расширять более специализированные узлы графа зависимости, такие, как манипуляторы, локаторы, формы и др., с целью добавления в них собственных функций. Порождая свои узлы от таких специализированных предков, вы сможете повторно использовать большую часть имеющихся у них функций, что позволит вам добавить лишь тот минимум кода, который необходим для достижения ваших целей.

4 Инструменты/Контексты

Часто решение задачи требует выполнения некоего набора диалоговых операций. Для разбиения ребер вы, например, должны выделить два ребра, а затем нажать клавишу Enter. Чтобы осуществить такую череду шагов диалога и служит контекст. Вы получаете достаточно большую свободу в том, что касается возможных действий контекста, включая способы интерпретации щелчков и перемещений мыши. Создание собственных контекстов предполагает реализацию специальных операций моделирования и анимации.

◆ Трансляторы файлов

С целью поддержки широкого спектра популярных и малоизвестных файловых форматов в состав Maya включены собственные трансляторы файлов. Они управляют загрузкой и сохранением данных Maya. Трансляторы файлов могут пользоваться любыми средствами, необходимыми для преобразования данных в свои собственные форматы. Более того, существуют даже специальные расширения, служащие для создания игровых трансляторов.

◆ Деформаторы

Любой деформатор описывает механизм перемещения ряда точек. По необходимости способ деформации может быть как простым, так и сложным.

Используя интерфейс C++ API, вы можете создавать свои собственные деформаторы, способные совершенно свободно деформировать разные типы объектов.

- **Шейдеры**

Maya содержит обширный набор узлов тонировки **Shading Network** (Сеть тонировки). В разных сочетаниях такие узлы могут использоваться для нетривиальной раскраски объектов. При помощи C++ API вы можете создавать свои собственные, нестандартные узлы тонировки. Подобно узлам Maya они будут иметь полный доступ ко всей необходимой информации о раскраске, включая сведения о нормалях, позициях, текстурах и т. д. Кроме того, вы можете реализовать и свои весьма специфичные механизмы суперсэмплирования.

Наряду с этим, существует возможность создания нестандартных аппаратных шейдеров.

- ◆ **Манипуляторы**

Хотя значения атрибута узла можно менять, используя **Channel Box**, **Graph Editor** и другие редакторы, Maya предоставляет наглядное средство изменения значений – манипуляторы. Манипуляторы поддерживают ряд визуальных элементов управления, при работе с которыми атрибут узла может быть изменен. К примеру, манипулятор для сферы позволяет переместить одну из своих точек и увеличить, тем самым, радиус данной сферы. Иногда манипуляторы могут обладать большей интуитивностью с точки зрения пользователя, так как с их помощью можно в режиме диалога перемещать или настраивать визуальный элемент управления, что зачастую проще, нежели набор нужного значения на клавиатуре. При интерактивном изменении значений пользователи сразу получают обратную связь, наблюдая за результатом своих действий.

- **Локаторы**

Локаторы призваны предоставить пользователям некие зрительные ориентиры. Они функционируют точно так же, как и любой другой объект Maya, стоя лишь разницей, что их нет в окончательном варианте изображения. Применение C++ API позволяет создавать собственные, нестандартные локаторы. Вы можете свободно рисовать их, пользуясь практически любыми методами OpenGL, поэтому по своему внешнему виду локаторы могут быть как очень простыми, так и очень сложными.

- **Поля**

Поле занимает объем, который может быть местом приложения различных сил. Обычно поле действует на частицы, заставляя их двигаться определенным образом. Среди примеров полей – гравитация, сопротивление среды и турбу-

лентность. Создание своих собственных полей позволит вам управлять силами, которые действуют на частицы.

◆ **Генераторы**

Цель генератора состоит в определении времени и способа получения последовательности частиц. Генераторы управляют количеством создаваемых частиц, а также направлением и скоростью их полета. Даже несмотря на то что в поставку Maya входит обширный набор разнообразных генераторов, вы можете создавать свои собственные, нестандартные генераторы, пользуясь для этого интерфейсом C++ API.

◆ **Формы**

Узлы форм содержат реальные геометрические данные. К формам относятся NURBS-кривые и поверхности, полигональные сетки, частицы и т. д. Maya позволяет создавать свои собственные, нестандартные формы. Они могут полностью встраиваться в среду Maya, поэтому с ними будут работать все стандартные инструменты моделирования и анимации. Вы сможете создавать, удалять и редактировать свои нестандартные формы так же, как и встроенные формы Maya.

◆ **Решатели**

Maya уже содержит широкий спектр инверсно-кинематических (ИК) решателей, однако допускается и создание ваших собственных аналогов. Вы можете построить нестандартный ИК-решатель и интегрировать его в состав Maya. Впоследствии он, как и стандартные средства пакета, сможет найти применение в управлении цепочкой костей.

Наравне с перечисленными элементами вы вправе описывать собственные законы пружин, которые могут служить для полного контроля над реализацией динамики мягких тел.

4.2. Основные понятия

Прежде чем перейти к обсуждению особенностей C++ API, чрезвычайно важно рассказать о некоторых основных понятиях. Так как любая система наделена своим собственным, присущим ей уровнем свободы, а значит, и ограничений, важно получить полное представление о факторах, мотивировавших принятие тех или иных решений при ее разработке. Понимание причин, по которым C++ API был спроектирован именно таким образом, позволяет проникнуть в суть того, как лучше всего реализовывать собственные подключаемые модули, чтобы обеспечить их хорошую работу в рамках системы Maya.

Модель интерфейса C++ API отличается от типичного объектно-ориентированного подхода, поэтому эти отличия немаловажно изучить. Понимание различий поможет вам в выборе верного пути при проектировании и реализации собственных модулей. На самом деле, оно решительно необходимо для разработки рациональных и эффективных подключаемых модулей.

4.2.1. Абстрактный уровень

При работе с C++ API может сложиться впечатление, что вы напрямую распоряжаетесь объектами и структурами данных Maya. В действительности же вы пользуетесь уровнем, расположенным выше реального ядра системы. Доступ к ядру Maya обеспечивается через API посредством вполне определенного множества интерфейсов. Ядро Maya состоит из всего набора внутренних функций и данных, разработанных компанией Alias | Wavefront. Вы никогда не сможете обратиться к ядру Maya напрямую. Все акты взаимодействия, будь то создание, удаление данных или манипулирование ими, должны осуществляться через API-интерфейс. Схема на рис. 4.1 иллюстрирует разные программные уровни и способ их связи между собой.

Тот, кто знаком хотя бы с одной системой [программирования](#), вспомнит, что типичные интерфейсы API содержат набор вызовов функций, которые обеспечивают доступ к базовой системе нижнего уровня. Интерфейс API в целом [образован](#) всем множеством доступных функций. Однако в Maya он представлен набором классов языка C++. Поэтому все обращения к ядру Maya выполняются посредством функций-членов каждого класса. Эти функции-члены могут создавать данные Maya, считывать их и управлять ими.

Итак, почему же потребовалось создавать API поверх ядра Maya? Отчего не предоставить разработчику прямой доступ к внутренним функциям и данным пакета? При создании API поверх ядра разработчик абстрагируется от реальных подробностей текущей реализации Maya. Не сообщая программисту деталей текущей реализации, конструкторы Maya вправе свободно изменять и совершенствовать ее ядро, не беспокоясь о «поломке» кода, созданного сторонними разработчиками. Интерфейс API не претерпевает радикальных изменений, а потому при внесении модификаций в ядро Maya сторонним разработчикам не нужно непрерывно обновлять свои подключаемые модули. Это защищает их инвестиции в те модули, которые уже написаны.

К тому же API предоставляет определенный уровень защиты от возможного некорректного использования. Коль скоро Maya поддерживает свои внутренние данные и управляет ими, она может помешать неправильно работающему модулю удалить жизненно важную информацию. API способен отследить потенциально

опасные вызовы и возвратить ошибку, на деле не выполняя сделанный вызов. По сути, интерфейс играет роль фильтра и предохраняет эти возможных вредоносных операций. Это не говорит о том, что вам не удастся «обрушить» Maya, однако подобные меры защиты существенно уменьшают такую вероятность.



Рис. 4.1. Интерфейс программирования

Пользуясь слоем, выстроенным поверх ядра Maya, вы можете поинтересоваться, нет ли потерь в скорости, связанных с прохождением этого дополнительного уровня. На практике такие потери незначительны. Многие C++-классы являются, по существу, интерфейсными классами, которые очень быстро преобразуют вызовы во внутреннее представление Maya. Кроме того, многие методы классов API имеют среди внутренних классов прямые аналоги, поэтому преобразование одних в другие не отнимает много времени.

4.2.2. Классы

Интерфейс C++ API состоит из ряда классов языка C++. В зависимости от своего типа большинство классов делятся на ряд очевидных иерархий. Сейчас нам совершенно не важно знать о том, что именно делает каждый класс и как он работает, важно лишь понимать общую структуру иерархии, а также то, в каком именно месте этой иерархии находятся те или иные классы. Полную иерархию классов Maya можно найти в документации к пакету:

maya_install\docs\en_US\html\DevKit\PlugInsAPI\classDoc\hierarchy.html

C++-иерархия Maya включает в себя немало различных классов. В самых распространенных модулях, к счастью, применяется лишь относительно небольшая их часть. На практике наиболее часто используется базовый набор классов, в то время как потребность во множестве других классов, известных лишь посвященным, возникает редко. В этой книге, по сути, не содержится подробных сведений о каждом классе, напротив, мы остановимся на самых важных и широко используемых из них.

Соглашение об именах

Имя каждого класса Maya начинается с заглавной буквы M, например **MObject**. Так как Maya не использует пространств имен C++, это помогает избегать любых конфликтов с прочими классами, описанными пользователем. Кроме того, Maya проводит разграничение между классами, помещая их в подклассы по признаку их функциональности. Хотя некоторые классы и не укладываются в типичную объектно-ориентированную иерархию «родитель - потомок», они действительно содержат общий набор функций и совместно пользуются общим префиксом. К примеру, вы обратите внимание на то, что многие классы имеют префикс MPx. От этого подкласса порождены все классы-заместители. Префиксы имен классов представлены в табл. 4.1.

ТАБЛИЦА 4.1 ПРЕФИКСЫ ИМЕН КЛАССОВ

Префикс	Логическая группировка	Примеры
M	Класс Maya	MObject, MPoint, M3dView
MPx	Класс-заместитель	MPxNode
MIt	Класс-итератор	MItDag, MItMeshEdge
MFn	Набор функций	MFnMesh MFnDagNode

Организация

Хотя по своей организации иерархия C++-классов может показаться аналогичной большинству объектно-ориентированных иерархий, существует несколько очень важных различий, которые необходимо понять для эффективного использования API. И действительно, неполное понимание этих отличий часто ведет к большой путанице на более позднем этапе проектирования собственных подключаемых модулей.

Классический подход

Прежде всего обратимся к построению *типичных* иерархий C++-классов, а затем проведем сравнение с моделью иерархии классов C++ API в Maya. Авторы традиционных учебников по объектно-ориентированному проектированию утверждают, что наиболее распространенная методология проектирования иерархии классов требует начинать с построения базового класса. Он является корнем иерархии. Несмотря на то что иерархия может *насчитывать* два и более базовых класса, в иллюстративных целях их число сокращается до одного. Этот корневой базовый класс обычно абстрактен и содержит только самые общие функции-члены и данные, совместно используемые всеми производными классами. Такие функции часто описываются как абстрактные (чистые виртуальные

функции). Классы, порожденные от данного базового класса, реализуют все эти функции-члены или некоторые из них, а возможно, и **дозавлиают** свои собственные абстрактные функции. Решение относительно количества классов, а также того, включать в них реализацию функций или оставить их абстрактными, принимает сам разработчик. Результатом применения этой методологии проектирования к созданию иерархии классов транспортных средств может стать иерархия, показанная на рис. 4.2. Она начинается с корневого класса **Vehicle** (Средство передвижения), далее от него порождается ряд специальных классов транспортных средств [**Motorcycle** (Мотоцикл), **Car** (Автомобиль), **Truck** (Грузовик)].

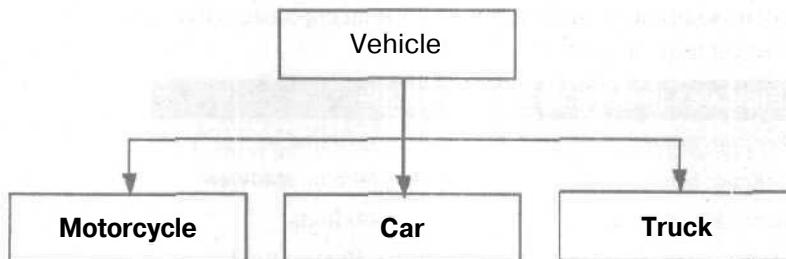


Рис. 4.2. Иерархия классов транспортных средств

Если вы воспользуетесь стандартным **объектно-ориентированным** подходом, то каждый из классов будет содержать как данные, так и функции, которые с ними работают. Каждый подкласс транспортных средств будет, по сути, содержать собственные, характерные для него данные о средстве передвижения. Класс **Car** может, к примеру, хранить информацию о **типе** колес и количестве передач. Эти данные **могут** быть скрыты в собственных или защищенных членах класса либо опубликованы как общедоступные. В ином случае доступ к этим данным может осуществляться только при помощи **функций-членов**. Вне зависимости от используемого подхода класс, в конечном итоге, определяет **интерфейс** с своим **данным**. Он решает, как вы можете к ним обратиться. А значит, каждый класс, в сущности, формирует собственный API для своих функций и своих данных.

Чтобы лучше соответствовать концепции **полиморфизма**, класс **Vehicle** может содержать описание следующего набора чистых виртуальных функций:

```

virtual void drive() = 0;
virtual int numWheels() = 0;
  
```

Подклассы **Motorcycle**, **Car** и **Truck** могут, в свою очередь, содержать код этих функций. Когда классы будут реализованы, их можно использовать следующим образом:

```
Car speedy;  
speedy.drive();
```

Поскольку все основные классы являются потомками **Vehicle**, вы можете воспользоваться полиморфизмом для вызова принадлежащей объекту функции **drive()**, не зная точного типа объекта. В следующем примере указатель **broom** может содержать адрес как объекта **Truck**, так и объекта **Car** и служить для вызова функции **drive()** любого из этих объектов. На деле же будет вызвана та функция **drive()**, которая соответствует нужному классу.

```
Truck t;  
Car c;
```

```
Vehicle *broom = &c;  
broom->drive();  
broom = &t;  
broom->drive();
```

Представьте себе, что иерархия транспортных средств – это API-интерфейс, предложенный вам как разработчику. Для расширения иерархии с целью внедрения в нее своих собственных классов вы можете породить их от **Vehicle** или любого другого класса. Если бы, например, вы захотели создать более специализированный тип автомобиля, который применяется для поездок по бездорожью, то могли бы унаследовать от класса **Car** новый класс **Offroad**. Новая иерархия классов показана на рис. 4.3.

Новый класс **Offroad** беспрепятственно внедрен в иерархию. Он может использоваться так же легко, как и любой из исходных классов, порожденных от базового средства передвижения. Даже несмотря на возможность добавлять классы, вы не вправе вносить изменения в начальную иерархию. Вы можете делать свои **расширения**, порождая собственные классы. Вам не удастся при такой организации добавить новые виртуальные функции в классы **Vehicle** или **Truck**. Это могут делать лишь авторы проекта исходной иерархии. Если они решат внести в иерархию те или иные изменения, это затронет и ваш класс. Объем **работы**, которую вам предстоит проделать со своим классом, зависит от глубины этих изменений. Кроме того, реализации в вашем классе могут потребовать и **какие-то**

новые функции-члены. Если в класс Vehicle добавлена, к примеру, следующая виртуальная функция:

```
virtual bool isElectric() = 0;
```

то ваш класс **Offroad** должен содержать реализацию этой функции, иначе вам не удастся создать экземпляр своего класса, поскольку тот станет абстрактным.

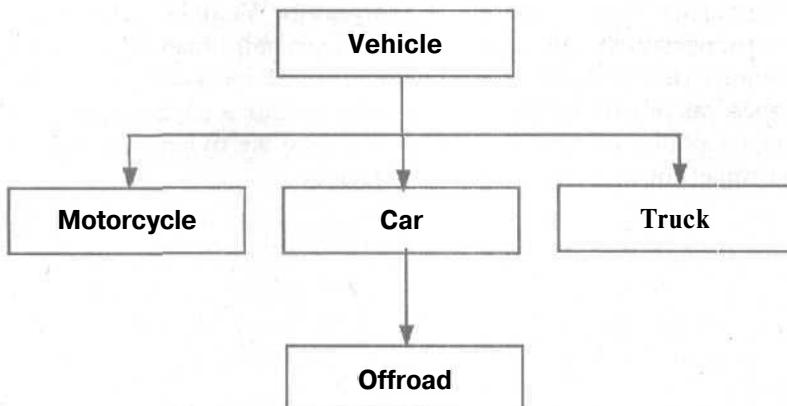


Рис. 4.3. Расширенная иерархия классов

Подход Maya

Узнав о сущности классического подхода к объектно-ориентированному проектированию иерархий, посмотрим теперь, чем же отличается подход, принятый в Maya. Чтобы продемонстрировать различия, спроектируем тот же набор классов транспортных средств еще раз, с использованием подхода Maya. Предыдущая модель объединяла данные и функции для их обработки в единый класс. В Maya они отделены друг от друга. Поэтому для хранения данных при создании иерархии средств передвижения организуется класс **Object**. Он позволяет хранить данные многих различных типов. Тип хранимых в нем данных зависит от того, каким классом транспортных средств он используется. Иерархия классов данных показана на рис. 4.4.

Класс **Object** будет содержать общие данные всех классов иерархии, тогда как в порожденные классы **MotorcycleObj**, **CarObj** и **TruckObj** войдут более конкретные сведения об отдельной категории средств передвижения. Эти классы содержат данные, а также собственные функции-члены для доступа к ним.

```
virtual void drive() = 0;
virtual int numWheels() = 0;
```

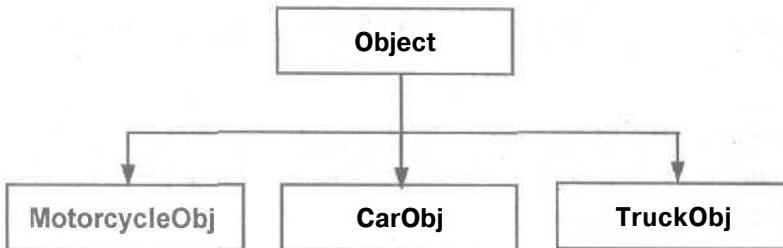


Рис. 4.4. Иерархия данных

Другая отдельная иерархия классов призвана обеспечить доступ к разнообразным классам данных и их обработку. Классы для обработки данных в Maya называются *наборами функций*. В этом примере вам предстоит создать классы, которые не содержат никаких данных, но при этом предоставляют функции-члены, способные работать с объектами типа **Object**. В объектно-ориентированной терминологии такие классы известны как *функции-члены*. Для обработки данных о транспортных средствах создана целая иерархия классов функциональных наборов. Она показана на рис. 4.5.

Корнем этой иерархии является класс **VehicleFn**. Он содержит то же множество функций-членов, что и класс исходной иерархии.

```

virtual void drive();
virtual int numWheels();
  
```

Отличие состоит в том, что этот класс не содержит данных, с которыми он работает. Взамен данные будут получены через классы иерархии **Object**. Таким образом, класс **MotorcycleFn** будет работать с объектом **MotorcycleObj**. Объект данных присоединяется к классу набора функций, а значит, при любом вызове функции-члена она будет работать с этими данными.

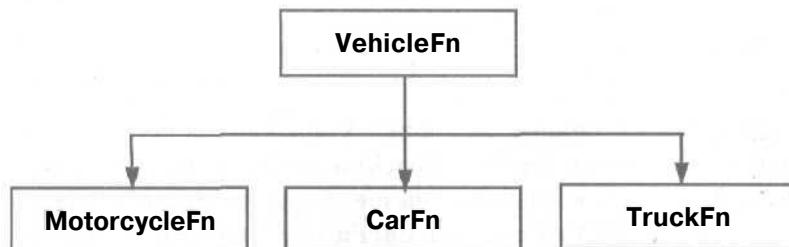


Рис. 4.5. Иерархия наборов функций

Класс **VehicleFn** содержит private-указатель на объект данных, с которым он работает.

```
Object *data;
```

В базовый класс **VehicleFn** введена еще одна функция-член, позволяющая задать тот объект данных, с которым будет работать **класс**.

```
virtual void setObject( Object *obj ) { data = obj; }
```

Как же при таком новом варианте организации можно реализовать отдельные операции, аналогичные тем, что имели место в исходной иерархии? Чтобы заставить автомобиль тронуться с места, нужно теперь сделать следующее:

```
CarObj carData;  
VehicleFn speedyFn;  
speedyFn.setObject( &carData );  
speedyFn.drive();
```

Первым создается объект **CarObj**. Он хранит информацию об автомобиле. Следом строится набор функций **VehicleFn** и присоединяется к сведениям о машине. Вызываемая функция **drive()** оперирует объектом **carData**.

Класс **VehicleFn** содержит описание общих функций всех порожденных классов, а значит, он может их вызывать. Этот же класс может работать и с объектом типа **TruckObj**.

```
TruckObj t;  
CarObj c;
```

```
VehicleFn broomFn;  
broomFn.setObject( &t );  
broomFn.drive();  
broomFn.setObject( &c );  
broomFn.drive();
```

Проницательный программист заметит, что класс **VehicleFn** не располагает информацией о реализации классов **CarFn** и **TruckFn**, а потому вызов **drive()** в предыдущем фрагменте на деле означает вызов функции **drive()** класса **VehicleFn**, но не функции **drive()** из состава **CarFn** или **TruckFn**. Хитрость заключается в том, что хотя интерфейс выполнения операций над данными описан в классах наборов функций, именно классы данных заняты реальной работой.

Как было показано ранее, в классах данных описаны свои собственные внутренние виртуальные функции `drive()`. Недоступная постороннему взгляду реализация функции `drive()` класса `VehicleFn` имеет вид

```
virtual void VehicleFn::drive()
{
    data->drive();
}
```

Исходя из того, какой объект данных, пользуясь `setObject()`, вы передали набору функций, вызывается соответствующая функция данных `drive()`. По сути, один и тот же функциональный набор может работать с разными типами объектов данных.

Но тогда может возникнуть вопрос, с чем связана суета вокруг наборов функций, если можно напрямую обращаться к объектам данных и вызывать их функции-члены. В Maya вы *никогда* не получите доступа к иерархии классов данных. Вам дано право обращаться лишь к одному классу с именем **MObject**. В нем хранится необходимая информация об иерархии. Так как вся иерархия данных от вас скрыта, то для работы со скрытыми объектами данных Maya вы должны пользоваться иерархией классов функциональных наборов. Именно через эти классы наборов функций вы будете получать доступ ко всем внутренним данным Maya.

Итак, основное различие между исходной организацией иерархии и ее реализацией в Maya состоит в том, что последняя из них раскрывает иерархию функций лишь посредством собственных классов функциональных наборов. В этом и заключается отличие от исходной организации, где была опубликована как иерархия данных, так и иерархия функций.

4.2.3. Класс MObject

Если в предыдущем разделе иерархия классов Maya была описана на уровне абстрактных понятий, то теперь мы рассмотрим ее конкретные особенности.

Описанные в приведенном примере объекты данных были вам хорошо известны. Вы знали о существовании классов **Object**, **MotorcycleObj**, **CarObj** и **TruckObj**. Зная же о различных объектах, вы могли обращаться к ним напрямую. В Maya разработчику открывается лишь корневой класс **Object**. Все остальные классы от него скрыты. В Maya эквивалент упомянутого в примере класса **Object** носит название **MObject**.

Класс **MObject** участвует в обращении к любым данным и потому служит для доступа ко всему спектру типов данных Maya. Вам может показаться, что он сам

фактически содержит данные, которые вы используете. В действительности он представляет собой лишь *описатель* другого *объекта*, расположенного на уровне ядра, А поскольку это только описатель, его можно считать указателем на некий внутренний элемент данных, который входит в ядро, и лишь ядро может распоряжаться этим указателем. Помимо названного указателя, объект **MObject** как таковой не содержит никаких данных. Эта ситуация проиллюстрирована на рис. 4.6.

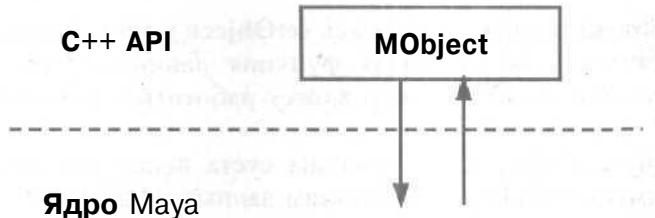


Рис. 4.6. Интерфейс **MObject**

Реально объект **MObject** содержит «пустой» указатель (`void *`) на некие внутренние данные, подробные сведения о которых никогда не раскрываются, так что в своем коде вам не удастся преобразовать этот указатель в нечто значащее. Только ядро **Maya** точно знает о том, на что ссылается указатель. Класс **MObject** не содержит никаких данных, а напротив, включает в себя лишь ссылку; поэтому удаляя или создавая объект *класса*, вы просто удаляете и создаете описатель. На деле вы не уничтожаете и не порождаете никаких внутренних данных *Maya*.

Данное обстоятельство может стать источником **многих** проблем, а потому понять его очень важно. *Maya* владеет внутренними данными и никогда не позволяет к ним обращаться. Взамен вы получаете описатель данных в форме объекта **MObject**. Вам никогда не удастся напрямую удалить внутренние данные, поскольку удаление **MObject** ведет лишь к удалению описателя, **но не данных** как таковых. *Maya* контролирует все свои внутренние данные и управляет ими, будь то узлы, атрибуты или прочая информация в системе. Вы получаете к ним доступ при помощи API, но никогда не имеете **возможности** управлять ими непосредственно. Доступ к данным и манипуляции с ними всегда выполняются через API-интерфейс.

Это важно запомнить, поскольку, как и в случае с любым **другим API**, вам дается лишь интерфейс к некой структуре или системе нижнего уровня. Вы ни-

когда не получите к ней прямого доступа. Внутренняя реализация Maya ни в коем случае не будет опубликована явно. Немаловажным побочным эффектом отсутствия прямого доступа является то, что вы как разработчик на самом деле никогда не приобретете статус владельца и не получите полного контроля над узлами Maya или любыми другими объектами. Maya действительно удерживает полный и абсолютный контроль над всеми своими объектами. Этот непреложный факт настолько важен, что нужно заострить на нем внимание еще раз.

Maya владеет всеми данными, вы не владеете ничем!

4.2.4. Наборы функций **MFn**

Познакомившись с действующим в Maya механизмом представления собственных данных пакета, обратимся к вопросам их создания, редактирования и удаления. Прежде чем начать работу с данными, нужно получить их описатель. В качестве такого описателя Maya использует класс **MObject**. Когда адрес данных будет занесен в объект **MObject**, следующим шагом станет создание набора функций для дальнейшей обработки данных.

Как упоминалось ранее, все классы с префиксом **MFn** являются функциональными наборами. Они предназначены для создания, редактирования и удаления данных. Чтобы создать узел **transform**, используется следующий набор функций с именем **MfnTransform**.

```
MFnTransform transformFn;  
MObject transformObj = transformFn.create();
```

Объект **transformObj** содержит описатель (**MObject**) вновь созданного узла **transform**. Теперь для работы с этим узлом можно вызывать функции-члены объекта **MFnTransform** **transformFn**. Имя узла можно получить, воспользовавшись функцией **name()**:

```
MString nodeName = transformFn.name();
```

Наборы функций организованы в виде иерархии классов в соответствии с типом данных, который они обрабатывают. Все классы функциональных наборов основаны на **MFnBase**. Предки набора функций **MFnTransform** представлены на рис. 4.7.

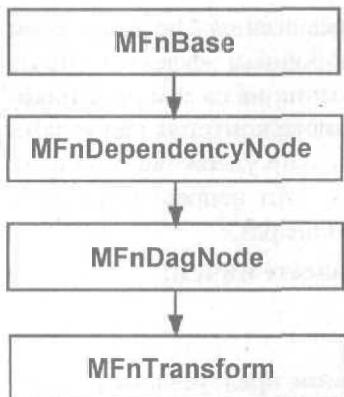


Рис. 4.7. Предки класса **MFnTransform**

Каждый очередной порожденный класс вносит свои новые функции, способные работать с более специализированными типами данных. Коль скоро **MFnTransform** порожден от **MFnDagNode**, он может оперировать всеми объектами, являющимися узлами ОАГ. Аналогично, его предком является **MFnDependencyNode**, а значит, **MFnTransform** может обрабатывать любые узлы зависимости. Эта иерархия функциональности позволяет производным классам работать со всеми **данными**, которые умеют обрабатывать их предки. Класс **MFnDagNode** может, например, оперировать узлами, созданными **MFnTransform**, потому что первый из них представляет собой базовый класс второго.

```

MFnTransform transformFn;
MObject transformObj = transformFn.create();
MFnDagNode dagFn( transformObj );
MString name = dagFn.name();
  
```

Заметьте, что набор функций **dagFn** может применяться к **transformObj**, поскольку созданный описатель **MObject** указывает на з'zel **transform**, который сам порожден от **dagNode**.

Что произойдет при передаче набора функций запроса на обработку **MObject**, для которого он не предназначен? Функциональный набор **MFnNurbsSurface** создан для работы с NURBS-поверхностями. Функциональный набор **MFnPointLight** оперирует точечными источниками света. Что случится, если набор функций **MFnNurbsSphere** получит запрос на обработку объекта, являющегося точечным источником света?

```
MFnPointLight pointLightFn;
MObject pointObj = pointLightFn.create();

MFnNurbsSurface surfaceFn( pointObj );
double area = surfaceFn.area();
```

Передав элемент данных `pointObj` конструктору `surfaceFn`, мы потребовали от этого набора, чтобы `pointObj` был обработан. В ходе вызова функции `area()` Maya проверяет, действительно ли объект **данных**, с которым она работает, является **NURBS-поверхностью**. Если это не так, Maya возвращает ошибку. Объект представляет собой точечный источник света, а не **NURBS-поверхность**, поэтому выдается ошибка. В данном случае она не фиксируется, так что программа продолжает свою работу. Следовательно, значение переменной `area` является некорректным. Обнаружение ошибок и вывод сообщений о них будут рассмотрены ниже.

Как Maya узнает о том, на какой тип данных ссылается объект **MObject**? Класс **MObject** содержит возвращающую тип объекта функцию-член `apiType()`. Вызвав ее, каждый класс набора функций способен определить, совместим ли он с **данным** объектом. Кроме того, может использоваться еще одна аналогичная функция **MObject** с именем `hasFn()`.

Пользовательские наборы функций

Иерархия классов исходного примера была дополнена классом **Offroad**. Для этого мы выполнили простое наследование от **Car**. Затем в порожденном классе потребовалось реализовать необходимые функции-члены класса **Vehicle** и класса **Car**. Однако Maya проводит разделение данных и функций, и этот подход здесь не работает.

Интуитивно понятно, что вы можете попытаться породить новый набор функций на основе существующего. Новый набор повторно реализует те функции, которые планируется перекрыть. Скажем, к примеру, что вы хотите реализовать новый геометрический тип NURBS вашей собственной разработки. Даже невзирая на то, что это будет новый тип со своими собственными возможностями, он будет обладать и некоторыми общими свойствами, присущими текущей реализации NURBS в Maya. Не исключено, что новый класс-набор функций **MFnMyNurbs** можно было бы породить от существующего класса **MFnNurbsSurface**. Располагая новым классом функционального набора, вы могли бы работать со своей новой NURBS-формой.

К сожалению, у вас ничего не получится. В действительности вместо создания нового типа NURBS вы попросту получили бы новый набор функций **NURBS**, который по-прежнему может работать только с существующими **NURBS-поверхностями**. Вспомните, что **MObject** указывает на некие **данные**, которые известны лишь Maya. С такими объектами **MObject** работают различные функции **MFn**. Конструкторы и проектировщики Maya точно **знают**, на какие данные ссылается **MObject**, и потому могут ими манипулировать. Никакие подробности относительно данных, на которые указывает **MObject**, неизвестны сторонним разработчикам, поэтому вам не удастся **обратиться** к этим данным каким-то осмысленным образом. Даже породив новый класс-набор **функций**, вы не получите больших прав **доступа**, чем вы имели прежде. Новый класс может, по сути, вызывать только те функции, которые уже реализованы в его базовом классе. Вы не сможете внедрить свои функции, потому что будете не способны оперировать реальными данными Maya. Они для вас **недоступны**, а значит, в создании новых наборов функций нет никакого смысла.

4.3. Разработка подключаемых модулей

Чтобы ввести в состав Maya свои собственные **нестандартные узлы**, команды и т. д., вам нужно создать подключаемый модуль. Он **представляет** собой динамически связываемую библиотеку. Во время работы **Maya** загружает этот модуль (библиотеку), что позволяет встроить в систему ваши новые функции.

Подключаемые модули в Windows - это стандартные динамически связываемые библиотеки со специальным расширением имени файла **.mll**, отличающимся от привычного расширения **.dll**. В разных средах **на** платформе Unix подключаемые модули являются совместно используемыми динамическими объектами с расширением имени файла **.so**. На самом деле вы можете использовать произвольное имя файла, однако названные общепринятые расширения обеспечивают большую степень унификации.

Допустим, что в среде Maya уже установлен инструмент разработчика. Возможно, вы помните, что он был одним из необязательных пакетов, доступных в процессе инсталляции. В него входят все заголовочные и библиотечные **файлы**, необходимые для создания модулей. Если инструмент разработчика установлен правильно, в главном каталоге **maya** должны находиться следующие подкаталоги:

maya_install\include
maya_install\libs
maya_install\devkit

Если они отсутствуют, вам нужно установить их прежде, чем вы будете двигаться дальше.

Выпуск новой версии Maya может потребовать изменения среды разработки подключаемых модулей. Это может означать то, что вам понадобится новая версия компилятора, или то, что модули должны располагаться теперь в другом каталоге. Эти и многие другие требования могут меняться от одной версии к другой. Так как среда разработки сильно зависит от конкретной версии Maya, то любые указания, приведенные в этой книге, могут оказаться неприменимыми в отношении будущих версий. Вместо того чтобы сообщать вам устаревшую информацию, мы привели все указания по разработке модулей для различных платформ на Web-сайте этой книги по адресу www.davidgould.com.

Web-сайт содержит самые актуальные инструкции по разработке подключаемых модулей для Windows, Irix, Linux, Mac OS X и более поздних версий платформ, на которых работает Maya. Размещение инструкций в Интернете означает, что их можно быстро обновлять и исправлять, а это гарантирует вам получение точного набора указаний по конкретной версии Maya, разработкой для которой вы занимаетесь.

4.3.1. Разработка подключаемых модулей под Windows

В данном разделе приведены указания по разработке подключаемых модулей для Maya 4.x, и он предназначен для тех, кто занят разработкой для платформы Windows. Ранее было сказано, что эти указания могут быть неприменимы по отношению к более поздним версиям пакета, поэтому за самой последней информацией мы просим вас обращаться на Web-сайт книги. Систему Maya поддерживают операционные системы Windows NT 4.0, Windows 2000 и Windows XP. Более поздние версии Windows, которые будут разрабатываться на основе любой из перечисленных, также должны работать с этим пакетом.

Для разработки подключаемых модулей Maya вам понадобится копия Microsoft Visual C++ 6.0 или выше. Проверьте, установили ли вы самые последние пакеты обновлений среды Microsoft Visual C++. Если вы знакомы с созданием DLL под Windows, то написание модулей должно стать для вас относительно простым делом, так как в основе того и другого лежит один и тот же процесс.

Убедитесь в надлежащем выполнении очередных *разделов*, где каждое следующее действие предполагает выполнение предыдущего.

Первые шаги

Если во время инсталляции Maya пакет Microsoft Visual C++ был уже установлен, мастер **Maya Plug-in Wizard** (Мастер подключаемых модулей Maya) должен был инсталлироваться автоматически. Он будет выполнять за вас большую работу по настройке подключаемых модулей. Вам будут заданы вопросы об установке целого ряда опций, а затем сгенерированы все необходимые файлы рабочей среды, проекта и исходного кода. Хотя подключаемый модуль можно настроить и без этого мастера, пользоваться им гораздо удобнее. Для создания модуля **Hello World** выполните следующие шаги.

1. Откройте Microsoft Visual C++.
2. Выберите пункт меню **File | New** (Файл Новый).
3. В окне диалога **New** щелкните по вкладке **Projects** (Проекты).
4. В списке доступных типов проекта щелкните по элементу **Maya Plug-in Wizard**.
5. Укажите имя **модуля**, введя в поле **Project Name:** (Имя проекта) текст **HelloWorld**.
6. В поле **Location:** (Каталог) укажите каталог, где вы хотите сохранить файлы разработки подключаемого модуля.
7. Щелкните по кнопке **OK**.
8. В **следующем** диалоговом окне выберите нужную вам **версию Maya**.
9. Если вы поместили инструмент разработчика в каталог, отличный от **стандартного**, установите **Location of the Developer Kit** (Каталог Developer Kit) в **custom location** (собственный каталог). Если, к примеру, **devkit** является подкаталогом **c:\Maya4.0**, то в поле каталога укажите **c:\Maya4.0**, а не **c:\Maya4.0\devkit**.
10. Введите свое имя в поле **Vendor Name** (Название производителя).
11. Щелкните по кнопке **Next** (Далее).
12. Оставьте в качестве типа модуля **Mel Command** (Команда MEL).
13. Введите в поле **Maya type name** (Имя типа Maya) строку **HelloWorld**.
14. Щелкните по кнопке **Finish** (Готово).

Если вместо кнопки **Finish** вы нажали бы **Next**, то могли бы установить имя выходного файла подключаемого модуля, а также указать прочие библиотеки Maya, которые требуется присоединить к модулю. Чаще всего вас будут вполне удовлетворять настройки по умолчанию, поэтому на этой странице нет необходимости что-то менять.

15. Далее на экран выводится перечень всех файлов нового проекта.
16. Щелкните по кнопке **OK**.
17. Щелкните по вкладке **Fileview** (Файлы) в расположенной слева области **Workspace** (Рабочая среда).
18. Раскройте элемент **HelloWorld Files**.
19. Раскройте элемент **Source Files** (Исходные файлы) и найдите имя файла `helloWorldCmd.cpp`.
20. Дважды щелкните по элементу `helloWorldCmd.cpp`, чтобы его открыть.
21. Ниже строки `#include <maya/MSimple.h>` добавьте следующий текст:

```
#include <maya/MGlobal.h>
```
22. Измените следующую строку, заменив текст

```
setResult( "helloWorld command executed!\n" );
```

на другой:

```
MGlobal::displayInfo( "Hello World\n" );
```
23. Сохраните файл `helloWorldCmd.cpp`.
24. Выполните компоновку модуля, выбрав в главном меню пункт **Build | Build helloWorld.mll** (Компоновка Компоновка `helloWorld.mll`) либо нажав клавишу F7.
Полученный в результате файл подключаемого модуля с именем `helloWorld.mll` будет находиться в подкаталоге Debug. Этот каталог нижнего уровня расположен в каталоге, указанном вами ранее в качестве месторасположения модуля.
Файл подключаемого модуля скомпонован. Теперь он готов к использованию его в среде Maya.

Выполнение

Смысл следующих шагов в запуске Maya и загрузке вашего нового модуля.

1. Откройте Maya.
2. Выберите в **главном** меню пункт **Windows | Settings/Preferences | Plug-in Manager** (Окна | Установка/Параметры Менеджер подключаемых модулей).
3. Щелкните по кнопке **Browse** (Обзор).
4. Перейдите в каталог, где хранится файл вашего модуля `helloWorld.mll`.
5. Выберите файл `helloWorld.mll`.
6. Щелкните по кнопке **Load** (Загрузить).

После небольшой паузы Maya загрузит указанный модуль. В разделе **Other Registered Plugins** (Другие зарегистрированные модули) вы должны увидеть `helloWorld.dll` с установленным флагом **loaded**. Он указывает на то, что сейчас данный модуль действительно загружен в память.

7. Закройте окно **Plug-in Manager**.
8. Нажмите клавишу с символом обратной кавычки (`) или щелкните по полю **Command Line**.
9. Наберите `helloWorld`, а затем нажмите клавишу **Enter**.
В строке **Command Feedback** будут выведены слова `Hello World`.

Только что вы с успехом создали, загрузили и выполнили свой первый подключаемый модуль для Maya.

Редактирование

Теперь, когда начальный вариант модуля был создан, вы неизбежно захотите изменить его исходный код.

1. Вернитесь в среду Visual C++.
2. Измените следующую строку файла `helloWorld.cpp`:
`MGlobal::displayInfo("Hello World\n");`
заменив ее на другую:
`MGlobal::displayInfo("Hello Universe \n");`
3. Сохраните файл `helloWorld.cpp`.
4. Еще раз выполните компоновку модуля, нажав клавишу F7 или выбрав в главном меню пункт **Build | Build helloWorld.dll**.
При компиляции будет выведена следующая ошибка.

```
-----Configuration: HelloWorld - Win32 Debug-----
Compiling...
helloWorldCmd.cpp
Linking...
LINK : fatal error LNK1168: cannot open Debug\helloWorld.dll for
      writing
Error executing link.exe

helloWorld.dll - 1 error(s), 0 warning(s)
```

Файл модуля `helloWorld.dll` нельзя записать на диск. Дело в том, что Maya по-прежнему удерживает в памяти предыдущую версию этого файла. До тех пор пока `helloWorld.dll` загружен в Maya, переписать его невозможно. Перед повторной компиляцией подключаемый модуль должен быть выгружен.

5. Вернитесь в Maya.
6. Выберите в главном меню пункт **Windows | Settings/Preferences | Plug-in Manager...**.
7. Щелкните по флагку **loaded** справа от элемента `helloWorld.dll`. Это приведет к выгрузке модуля.
8. Оставьте окно **Plug-in Manager** открытым.
9. Вернитесь в Visual C++.
10. Снова скомпонуйте модуль, нажав F7 или выбрав в главном меню пункт **Build | Build helloWorld.dll**.
На этот раз компоновка должна завершиться успешно.
- I. Вернитесь в Maya.
12. В окне **Plug-in Manager** щелкните по флагку **loaded** рядом с элементом `helloWorld.dll`. Тем самым модуль будет загружен.
13. Щелкните по строке **Command Line**.
14. Наберите `helloWorld`, а затем нажмите клавишу **Enter**.
В строке **Command Feedback** будут выведены слова `Hello Universe`.

Процесс редактирования и перекомпиляции модуля ничем не отличается от подобных процессов при разработке любого **программного** продукта. Отличие, связанное с Maya, состоит в необходимости проверки, выгружен ли модуль из памяти перед сборкой. **Когда** модуль будет перекомпилирован, вы сможете загрузить его снова.

Отладка

Отладку в Visual C++ можно выполнять в режиме диалога.

1. Закройте Maya.
2. Вернитесь в Visual C++.

Во время создания проекта `helloWorld` мастер автоматически сгенерировал два варианта **конфигурации** – отладочный и итоговый. По умолчанию используется конфигурация для отладки. Она обеспечивает хранение в подключаемом модуле отладочной информации. Даже несмотря на то что настройки

проекта уже должны соответствовать варианту для отладки, сейчас мы рассмотрим шаги, связанные с выбором активной конфигурации.

3. Выберите из главного меню пункт **Build | Set Active Configuration...** (Компоновка | Установить активную конфигурацию...).
4. Выберите из списка **HelloWorld - Win32 Debug**.
5. Щелкните по кнопке **OK**.
Если эта конфигурация задана, вам не придется повторно выполнять только что названные шаги, при условии что активная конфигурация не изменилась.
6. Выберите в главном меню пункт **Project | Settings...** (Проект Настройки...).
7. Щелкните по кнопке **Debug** (Отладка).
8. Щелкните по кнопке со стрелкой рядом с полем **Executable for debug session:** (Исполнимый файл для запуска при отладке),
9. Щелкните по кнопке **Browse...**
10. Найдите исполнимый файл Maya с именем `maya.exe`. Обычно он находится в подкаталоге `bin` главного каталога Maya.
11. Щелкните по кнопке **OK**.
12. Щелкните по элементу **Category:** (Категория).
13. Выберите из выпадающего списка элемент **Additional DLLs** (Дополнительные библиотеки).
14. Щелкните по списку **Modules...** (Модули...).
На экране появится редактируемое поле.
15. Щелкните по кнопке **...**
На экране появится диалоговое окно **Browse**.
16. Установите в поле **Files of type:** (Тип файлов) значение **All Files(*.*)** [Все файлы(*.*)].
17. Найдите файл подключаемого модуля `helloWorld.dll`. Он находится в подкаталоге **Debug** основного проекта `helloWorld`.
18. Щелкните по кнопке **OK**.
19. Щелкните по кнопке **OK**, чтобы закрыть диалоговое окно **Project Settings** (Настройки проекта).
Теперь проект готов к началу отладки. Перечисленные шаги нужно выполнить для каждого проекта только один раз.

20. Находясь в файле `helloWorld.cpp`, щелкните в произвольном месте следующей строки текста:

```
MGlobal::displayInfo( "Hello Universe\n" );
```

21. Для установки контрольной точки нажмите клавишу F9.
Слева от этой строки появится красная точка.

22. Чтобы приступить к отладке, нажмите клавишу F5 или выберите в главном меню пункт **Build | Start Debug... | Go** (Компоновка Запуск отладки... | Начать).
Если программа `maya.exe` впервые запускается под отладчиком, вы увидите сообщение о том, что исполняемый файл не содержит отладочной информации.

23. Выделите флајок **Do not prompt in the future** (Не напоминать в будущем).

24. Щелкните по кнопке **OK**.

Maya будет загружена. Следующие шаги, как и прежде, связаны с загрузкой модуля.

25. Выберите в главном меню пункт **Windows | Settings/Preferences | Plug-in Manager**.

26. Щелкните по кнопке **Browse**.

27. Перейдите в каталог, где хранится файл вашего модуля `helloWorld.dll`.

28. Выберите файл `helloWorld.dll`.

29. Щелкните по кнопке **Load**.

После небольшой паузы Maya загрузит модуль.

30. Закройте окно **Plug-in Manager**.

31. Нажмите клавишу с символом обратной кавычки (`) или щелкните по полю **Command Line**.

32. Наберите `helloWorld`, а затем нажмите клавишу **Enter**.

Модуль начнет свое выполнение, а затем, как только достигнет контрольной точки, сразу же вернет вас в Visual C++. Теперь вы можете решать обычные задачи отладки, связанные с проверкой значений переменных, пошаговым выполнением кода и т. д.

Выпуск программного продукта

Как упоминалось в предыдущем разделе, проект `helloWorld` был автоматически создан в двух конфигурациях - отладочной и итоговой. В ходе разработки и отладки модуля должна применяться отладочная конфигурация. Когда же вы решите выпустить окончательную версию продукта, нужно использовать итоговый вариант. Он гарантирует, что ваш модуль работает максимально быстро и не содержит не-

нужной отладочной информации. Чтобы воспользоваться итоговой конфигурацией, выполните следующие действия.

- 1 Выберите в главном меню пункт **Build | Set Active Configuration...**
- 2 Выберите из списка **HelloWorld - Win32 Release**.
3. Щелкните по кнопке **OK**.
Подключаемый модуль должен быть скомпонован еще раз, с использованием текущей конфигурации.
4. Выполните повторную компоновку модуля, нажав клавишу F7 или выбрав в главном меню пункт **Build | Build helloWorld.mll**.
Предназначенная для окончательного выпуска версия файла `helloWorld.mll` находится в подкаталоге **Release**.

4.3.2. Инициализация и деинициализация

Поскольку подключаемый модуль является *динамически связываемой библиотекой*, он должен обладать *точкой входа* и *точкой выхода*. Так именуют функции, которые вызываются при первой загрузке модуля (функция *входа*) и его окончательной выгрузке (функция *выхода*), соответственно. Под Windows данные процессы обычно обрабатываются функцией `DllMain`, написанной самим программистом. В модуле `helloWorld` эти точки входа и выхода были добавлены автоматически, посредством макроопределения `DeclareSimpleCommand`. Оно служит для автоматического создания команд и генерирует для вас функции инициализации и деинициализации.

Модуль HelloWorld2

Рассмотрим теперь процессы инициализации и деинициализации модуля более внимательно. В этом примере создается команда `helloWorld2`, которая, как и прежде, всего лишь выводит на экран текст `Hello World`. Полный исходный код приведен ниже.

Модуль: HelloWorld2

Файл: HelloWorld2.cpp

```
#include <maya/MPxCommand.h>
#include <maya/MGlobal.h>
#include <maya/MFnPlugin.h>

class HelloWorld2Cmd : public MPxCommand
```

```
{  
public:  
    virtual MStatus doIt ( const MArgList& )  
    { MGlobal::displayInfo( "Hello World\n" ); return MS::kSuccess; }  
    static void *creator() { return new HelloWorld2Cmd; }  
};  
  
MStatus initializePlugin( MObject obj )  
{  
    MFnPlugin pluginFn( obj, "David Gould", "1.0" );  
  
    MStatus stat;  
    stat = pluginFn.registerCommand( "helloWorld2",  
                                    HelloWorld2Cmd::creator );  
    if ( !stat )  
        stat.perror( "registerCommand failed" );  
        // "аварийное завершение registerCommand"  
  
    return stat;  
}  
  
MStatus uninitializedPlugin( MObject obj )  
{  
    MFnPlugin pluginFn( obj );  
  
    MStatus stat;  
    stat = pluginFn.deregisterCommand( "helloWorld2" );  
    if ( !stat )  
        stat.perror( "deregisterCommand failed" );  
  
    return stat;  
}
```

Первый фрагмент кода попросту создает команду `helloWorld2`. Этот раздел книги посвящен **лишь** некоторым основам создания команд. В разделе 4.4 данный вопрос рассматривается намного подробнее. Класс команды содержит простую функцию `doIt()`, которая вызывается при выполнении команды. Как и прежде, команда всего лишь выводит в строку **Command Feedback** фразу Hello World.

```
virtual MStatus doIt( const MArgList& )  
{ MGlobal::displayInfo( "Hello World\\n" ); return MS::kSuccess; }
```

Команда также содержит статическую функцию `creator()`, которая размещается в памяти и возвращает объект команды.

```
static void *creator() { return new HelloWorld2Cmd; }
```

За описанием самой команды следуют две функции инициализации и деинициализации.

```
MStatus initializePlugin( MObject obj )  
MStatus uninitializedPlugin( MObject obj )
```

Обе эти функции должны входить в любой подключаемый модуль Maya. Если они **отсутствуют**, сборка модуля закончится неудачно. Функция `initializePlugin` принимает на вход объект типа `MObject`. Он является описателем внутренних данных Maya.

```
MStatus initializePlugin( MObject obj )  
{
```

В следующей строке создается объект `MFnPlugin`, инициализируемый переданной функции переменной `obj`. Присоединение `MFnPlugin` к объекту `MObject` дает возможность впоследствии вызывать функции `MFnPlugin`, которые, в свою очередь, обрабатывают `MObject`.

```
MFnPlugin pluginFn( obj, "David Gould", "1.0" );
```

Далее команда `helloWorld2` регистрируется. Регистрация команды делает ее известной для `Maya`, что позволяет **пользоваться** этой командой. Регистрация включает назначение имени команды и указание **функции-создателя**. Имя - это текст, **используемый** для вызова команды, в данном случае, это просто `helloWorld2`. Функция-создатель представляет собой статическую функцию, выделяющую память для одного экземпляра команды. Эта информация требует регистрации со стороны Maya, так как иначе Maya не узнает о том, как создать экземпляр вашей команды.

```
MStatus stat;
stat = pluginFn.registerCommand( "helloWorld2",
                                  HelloWorld2Cmd::creator );
```

Затем результат регистрации проверяется, и в случае неудачи выводится сообщение об ошибке.

```
if ( !stat )
    stat.perror( "registerCommand failed" );
```

Наконец, функция инициализации возвращает этот результат.

```
return stat;
```

```
}
```

Если признак возврата функции не равен `MS::kSuccess`, то подключаемый модуль завершает работу и динамическая библиотека автоматически выгружается. Кроме того, в строке Command Feedback выводится сообщение об ошибке. Важно заметить, что при неудачном завершении функции `initializePlugin` функция `uninitializePlugin` вызываться не будет. На деле, при возникновении ошибки в функции `initializePlugin` все операции очистки должны выполняться в этой же функции до ее завершения.

Функция `uninitializePlugin` совершает действия, противоположные тем, что были выполнены в `initializePlugin`. Она отменяет регистрацию команды, зарегистрированной в функции `initializePlugin`.

```
stat = pluginFn.deregisterCommand( "helloWorld2" );
```

При этом функция возвращает признак того, успешно ли завершилась deinициализация. Если возвращаемый признак не равен `MS::kSuccess`, модуль не выгружается. Он остается в памяти Maya и не удаляется из нее до тех пор, пока пакет не закончит свою работу.

В данном конкретном примере в Maya была зарегистрирована единственная команда. В действительности же количество таких регистраций не ограничено. В следующих главах вы узнаете о регистрации других функциональных элементов, таких, как пользовательские узлы и пользовательские данные. Однако как бы то ни было, функции `initializePlugin` и `uninitializePlugin` всего лишь сообщают Maya о том, какие новые возможности предоставляет ей модуль, что позволяет использовать его в приложении.

4.3.3. Ошибки

Обнаружение

Проверка на наличие ошибок, надлежащая обработка и выдача сообщений об их обнаружении очень важны для создания надежного и устойчивого приложения. В Maya предусмотрен качественный механизм сообщений об ошибках, доступный через использование класса **MStatus**. Класс описывает возможные состояния результата конкретной операции. Если такая операция закончится неудачно, признак будет переведен в соответствующее состояние.

Почти все функции классов Maya принимают необязательный указатель на объект **MStatus**. При передаче указателя на **MStatus** Maya присваивает этому объекту значение результата вызова функции. Полное описание функции **name()** класса **MFnDependencyNode** имеет вид:

```
MString name( MStatus * ReturnStatus = NULL ) const
```

Не проверяя результат функции, ее можно вызвать следующим образом:

```
MString dagName;  
MFnDagNode dagFn( obj );  
dagName = dagFn.name();
```

Чтобы правильно реализовать обнаружение ошибок, важно выяснить, успешно ли завершилась функция. Для этого функции **name()** передается указатель на объект **MStatus**.

```
MStatus stat;  
dagName = dagFn.name( &stat );  
if( !stat )  
    MGlobal::displayError( "Unable to get dag name" );  
    // "Невозможно получить имя"
```

Результат вызова функции сохраняется в объекте **stat**. Он проверяется на предмет установки признака неудачного завершения, и, если такой признак установлен, вы можете предпринять соответствующее действие. В данном примере на экран выводится сообщение об ошибке. Пример иллюстрирует наиболее широко применяемую методику обнаружения ошибок в подключаемых модулях.

Существует немало способов проверки результирующего значения **MStatus**. Наравне с использованной ранее конструкцией **if(!stat)** можно воспользоваться функцией **error()**.

```
if( stat.error() )  
    ... // ошибка
```

Можно извлечь из объекта точное значение признака и сравнить его с тем или иным кодом состояния. Чтобы проверить, имеет ли признак значение кода MS::kSuccess, выполните следующий оператор:

```
if( stat.statusCode() != MS::kSuccess )
    ... // ошибка
```

или его более простой вариант:

```
if( stat != MS::kSuccess )
    ... // ошибка
```

Используемый в Maya механизм сообщений об ошибках, к сожалению, заставляет разработчика пребывать в напряжении и сохранять бдительность при обнаружении проблем. Чаще всего программист должен проверять результат вызова *каждой* функции Maya. Это может стать утомительным, и многие переходят к проверкам от случая к случаю. В то же время постоянный контроль ошибок чрезвычайно важен на протяжении всего модуля. Наличие единственной проверки в серии вызовов функций, не содержащих иных проверок на предмет неудачных вызовов, сильно затрудняет обнаружение места, где произошла ошибка. Выполняя проверку в каждом фрагменте кода, можно гораздо быстрее находить ошибки и разрешать проблемы, возникающие на этапе выполнения,

Чтобы облегчить себе написание кода для обнаружения ошибок, вы можете воспользоваться следующими макроопределениями:

```
inline MString MyFormatError( const MString &msg, const MString
                             &sourceFile, const int &sourceLine )
{
    MString txt( "[MyPlugin] " );
    txt += msg;
    txt += ", File: "; // "Файл: "
    txt += sourceFile;
    txt += " Line: "; // "Строка: "
    txt += sourceLine;
    return txt;
}

ftdefine MyError( msg ) \
< \
MString „txt = MyFormatError( msg, __FILE__, __LINE__ ); \
```

```
MGlobal::displayError( __txt ); \
cerr << endl << "Error: " << __txt; //Ошибка:
} \

#define MyCheckBool( result ) \
if( !(result) ) \
{ \
MyError( #result ); \
}

#define MyCheckStatus( stat, msg ) \
if( !stat ) \
< \
MyError( msg ); \
}

#define MyCheckObject( obj, msg ) \
if( obj.isNull() ) \
{ \
MyError( msg ); \
}

#define MyCheckStatusReturn( stat, msg ) \
if( !stat ) \
{ \
MyError( msg ); \
return stat; \
}
```

Обратите внимание на то, что эти макроопределения включают в сообщение об ошибке имя исходного файла и соответствующий номер строки. Кроме того, они автоматически выводят это сообщение в стандартный поток ошибок. Вы можете использовать эти макросы так:

```
MObject dagObj = dagFn.object();
MyCheckObject( dagObj, "invalid dag object" ); // "неверный объект DAG"
```

Другой пример содержит следующие строки:

```
MStatus stat;
dagName = dagFn.name( &stat );
MyCheckStatusReturn( stat, "Unable to get name" ); // "Невозможно
получить имя"
```

Важно заметить тот факт, что во многих примерах в этой книге не выполняется постоянная проверка ошибок. Причина этого в предпринятой автором книги попытке не перегружать код и не затемнять его смысл. Когда все проверки ошибок удалены, основные идеи становятся более понятными для читателя. Именно ради этой цели сохранения простоты и краткости в код примеров введено **очень** мало проверок на наличие ошибок. Однако при разработке модулей нужно поступать как раз наоборот. Каждая операция должна проверяться, и это надо делать всегда и везде. В результате ваши модули будут более надежными и стабильными.

Вывод сообщений

Класс **MStatus** содержит также ряд дополнительных функций, связанных с выдачей сообщений об ошибках. Функция **errorString()** возвращает строку в соответствии с текущим кодом ошибки. Функция **errortext()** позволяет вывести сообщение об ошибке в текущий поток **stderr**. Приведем пример применения этих функций.

```
if( stat.error() )
    stat.perror( MString( "Unable to get name. Error: " ) +
                 // "Невозможно получить имя. Ошибка: "
                 stat.errorString() );
```

Как и язык **MEL**, интерфейс C++ API содержит функции вывода предупреждений и информации об ошибках. Основной способ выдачи сообщений об ошибках состоит в использовании статической функции **displayError()** класса **Mglobal**.

```
MGlobal::displayError( "object has been deleted" ); // "объект был
удален"
```

Текст сообщения об ошибке отображается в строке **Command Feedback** красным цветом. Чтобы вывести предупреждение, обратитесь к функции **displayWarningC()**.

```
MGlobal::displayWarning( "the selected object is of the wrong type" );
// "выделенный объект имеет неверный тип"
```

Предупреждение отображается в строке **Command Feedback** пурпурным цветом. Наконец, пользуясь функцией `displayInfo()`, можно вывести на экран информацию общего характера.

```
MGlobal::displayInfo( "select a light, then try again" );
// "выделите источник света и повторите попытку"
```

Несмотря на наличие большого числа способов уведомить пользователя об ошибках и предупреждениях, `MGlobal::displayError()` и `MGlobal::displayWarning()` - это те из них, которые следует использовать всегда. Они не перестают работать и при запуске Maya в пакетном режиме, при отсутствии интерфейса. Если вы хотите отобразить результат сравнения с участием `MStatus`, воспользуйтесь функцией `errorString()` класса `MStatus` в сочетании с методом `MGlobal::displayError()`.

```
if( !stat )
{
    MGlobal::displayError( MString("Unable to get name. Error: ") +
                           stat.errorString() );
}
return stat;
```

Не применяйте для вывода ошибок и предупреждений всплывающие окна. Вы не знаете того, будет ли ваша команда вызываться многократно, а отображение всплывающего окна всякий раз может только помешать пользователю. Окна не будут появляться и в том случае, если Maya работает в пакетном режиме, тогда выдачи сообщений для просмотра пользователем также не произойдет.

Кроме того, вывод ошибок и предупреждений в стандартный поток сообщений об ошибках, на самом деле, должен использоваться лишь для отладки. Пользователи не смогут увидеть этих ошибок, если запустят Maya с графическим интерфейсом. Вы должны спроектировать свой модуль так, чтобы подобная отладочная и тестовая информация не отображалась в окончательной версии продукта.

Класс `MGlobal` также предоставляет некоторые возможности журнализации ошибок. Возможно, вы сочтете их удобными, хотя используются они далеко не всегда. Полный перечень этих функций приведен в руководстве по данному классу.

Интеграция

Часто, приступая к написанию своих функций в составе подключаемого модуля, лучше **всего** остановиться на принятом в Maya механизме вывода информации об ошибках. Тогда ваши классы будут нормально работать с другими классами C++ API. В простейшем случае вы, как и разработчики Maya, можете предусмотреть в своих функциях указатель на дополнительный признак. Следующая функция показывает, как этот прием можно реализовать.

```
int myNumPoints( points *data, MStatus *returnStatus = NULL );
```

Внутри функции вы должны просто проверить указатель на корректность и при дальнейшем возврате признака установить, успешно завершилась функция или нет.

```
int myNumPoints( points *data, MStatus *returnStatus )
{
    int num = 0;
    MStatus res = MS::kSuccess;
    if( data )
        num = data->nElems();
    else
        res = MS::kFailure;
    if( returnStatus )
        *returnStatus = res;
    returnnum;
}
```

Некоторые функции Maya не принимают указатель на объект класса **MStatus**, а наоборот, сами возвращают подобный объект. Иллюстрацией этого служит следующий пример.

```
MStatus myInitialize()
{
    bool ok;
    ... // выполните инициализацию
    return ok ? MS::kSuccess : MS::kFailure;
}
```

Правильное написание функций с учетом возврата **MStatus** - напрямую или через указатель – гарантирует, что они ведут себя подобно функциям Maya. и*

Таким образом, выбранный вами подход к обнаружению ошибок и выдаче сообщений о них будет неизменным на протяжении всего модуля. К тому же это означает, что и другим программистам, читающим или использующим ваш код, не придется применять разные схемы выявления ошибок и вывода информации.

4.3.4. Загрузка и выгрузка

Как упоминалось ранее, подключаемый модуль должен быть загружен в Maya прежде, чем его функции станут доступными для работы. Также его можно выгрузить в том случае, если он больше не нужен. И, наконец, вы обязаны выгрузить модуль из Maya во время перекомпиляции.

Коль скоро задачи загрузки и выгрузки модуля решаются так часто, лучше всего попытаться их автоматизировать. Следующие указания содержат подробное описание того, как на панели Shelf создать две кнопки для выгрузки и загрузки подключаемого модуля.

1. Откройте Maya.
2. В редакторе Script Editor наберите следующий текст, но не выполняйте его. Задайте в строке \$pluginFile полный путь к подключаемому модулю, который вы разрабатываете.

```
|  
string $pluginFile = "c:\\\\helloWorld\\\\Debug\\\\helloWorld.mll";  
if( 'pluginInfo -query -loaded $pluginFile' && ! 'pluginInfo -query -  
unloadOK $pluginFile' )  
    file -f -new;  
unloadPlugin helloWorld.mll;  
}
```

Обратите внимание на то, что операторы MEL заключены в скобки. Это сделано с целью создания временного блока, с тем чтобы все созданные переменные были локальными, а значит, не нарушали глобального пространства имен.

Команда pluginInfo с флагами -query и -unloadOK вызывается для определения возможности успешной выгрузки модуля. Наиболее распространенная причина, по которой нельзя выгрузить модуль, состоит в том, что одна из его команд или один из его узлов все еще используется сценой. Даже если вы удалите эти узлы, они могут по-прежнему находиться в очереди отмены Maya. В таком случае самым простым решением является создание новой сцены. Такая операция вручную удаляет все команды и все узлы. Для этого

служит оператор `file -f -new`. Наконец, по команде `unloadPlugin` происходит фактическая выгрузка модуля.

3. Выделите набранный текст, а затем перетащите его на панель **Shelf**.
На панели будет создана кнопка. Мы будем именовать эту кнопку **Unload** (Выгрузить)
4. В редакторе **Script Editor** наберите следующий текст, но не выполняйте его.
Снова задайте в переменной `$pluginFile` путь к разрабатываемому вами модулю.

```
{  
string $pluginFile = "c:\\helloWorld\\Debug\\helloWorld.dll";  
loadPlugin $pluginFile;  
}
```

По команде `loadPlugin` модуль попросту загружается.

5. Выделите набранный текст, а затем перетащите его на панель **Shelf**.
На панели будет создана кнопка. Мы будем именовать эту кнопку **Load** (Загрузить).

Когда обе эти кнопки настроены, процесс загрузки и выгрузки модулей производится гораздо быстрее. Кнопки можно использовать следующим образом.

- ◆ Напишите подключаемый модуль.
- * Щелкните по кнопке **Load**.
- ◆ Протестируйте модуль.
- Φ Щелкните по кнопке **Unload**.
- Φ Отредактируйте исходный код и перекомпилируйте его.
- * Щелкните по кнопке **Load**.
- * Протестируйте модуль.
- Φ ... Повторите свои действия еще раз.

Наряду с упомянутым выше методом, вы можете автоматически загружать модули при запуске Maya, выполнив для этого следующие шаги.

1. Установите в окне **Plug-in Manager** для данного модуля признак **auto load** (автозагрузка).
2. Включите путь к каталогу с модулем в переменную окружения `MAYA_PLUG_IN_PATH` или воспользуйтесь для установки этой переменной окружения файлом `maya.env`.
3. Перезапустите Maya, и ваш модуль загрузится автоматически.

4.3.5. Размещение

Если вам нужно сделать свой модуль доступным широкому кругу пользователей, вы должны выбрать схему его размещения. Метод размещения подключаемых модулей будет, по всей видимости, тем же, что и при размещении сценариев.

1. Если работа ведется в производственной среде, где каждый пользователь может обращаться к центральному серверу, задача размещения существенно упрощается. Просто создайте на сервере совместно используемый каталог, например `\server\mayaPlugins`. В этом примере используется формат, соответствующий универсальному коду имен **UNC** (Universal Naming Code), однако вы должны пользоваться **тем** форматом пути, которого требует ваша сеть.

Если у вас нет центрального сервера, а вместо этого каждый пользователь имеет свою собственную локальную машину, создайте на каждой машине каталог, например `c:\mayaPlugins`.

Будем обозначать его как *каталог_модулей*.

2. Скопируйте двоичные файлы модуля в этот каталог.
3. На каждой из машин пользователей установите переменную окружения с именем `MAYA_PLUG_IN_PATH` и включите в нее путь к каталогу на сервере.

```
set MAYA_PLUG_IN_PATH=$MAYA_PLUG_IN_PATH; каталог_модулей
```

Также вы можете обновить пользовательский файл `maya.env`, чтобы он содержал установку переменной окружения:

```
MAYA_PLUG_IN_PATH=$MAYA_PLUG_IN_PATH; каталог_модулей
```

Хотя вы можете сохранять подключаемые модули непосредственно в каталоге `maya_install\plugins`, делать этого не рекомендуется. Лучше всего хранить свои модули отдельно от стандартных модулей Maya. Это позволит избежать путаницы и поможет при нахождении и обновлении ваших модулей.

Обновление

ЕСЛИ в тот момент, когда вы размещаете свои модули, пользователи уже работают с Maya, им не удастся получить самую последнюю версию автоматически. Большинство операционных систем действительно запрещают переписывать старые файлы модулей, коль скоро их использует Maya. Для корректного обновления модулей пользователи должны либо покинуть среду Maya и запустить ее **вновь**, либо выгрузить, а затем загрузить **модуль**. Если сцена содержит данные, специфичные для этого модуля, выгрузить его невозможно. Самое простое, что можно сделать в такой ситуации, - запустить Maya еще раз.

4.4. Команды

Составляя сценарии на языке MEL, вы, по всей вероятности, будете пользоваться множеством команд Maya. Работая с интерфейсом C++ API, вы получите возможность добавлять собственные нестандартные команды, которые могут применяться точно так же, как и стандартные команды пакета. Подобно им, вы можете вызывать свои команды при написании сценариев MEL и в других ситуациях, допускающих вызов команд. В минимальном варианте команда представляет собой просто функцию, которая вызывается при выполнении этой команды. В предыдущем разделе мы создали команду `helloWorld`. При ее выполнении на экран выводился текст Hello World.

Данный раздел посвящен созданию более сложных команд. Они могут иметь намного больше возможностей, таких, как получение множества входных значений (аргументов), предоставление справочной информации и работа со встроенным в Maya механизмом отмены и повторного выполнения.

4.4.1. Создание команд

Чтобы понять, как создаются более сложные команды, подробно рассмотрим команду с именем `posts`. Она принимает на вход кривую и генерирует ряд стоек (цилиндров), расположенных вдоль кривой. На рис. 4.8 изображена исходная «направляющая» кривая.

На втором рисунке, рис. 4.9, показан результат выполнения этой команды.

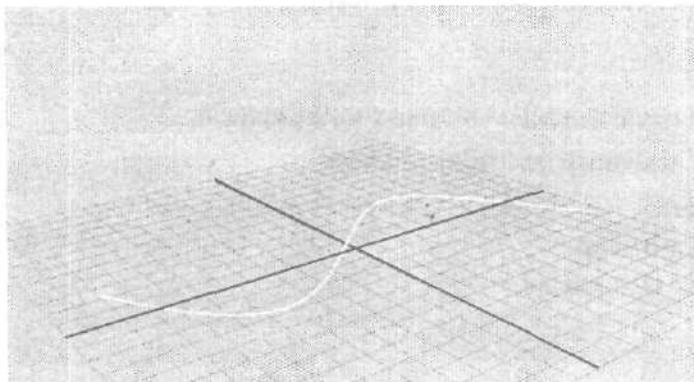


Рис. 4.8. Направляющая кривая

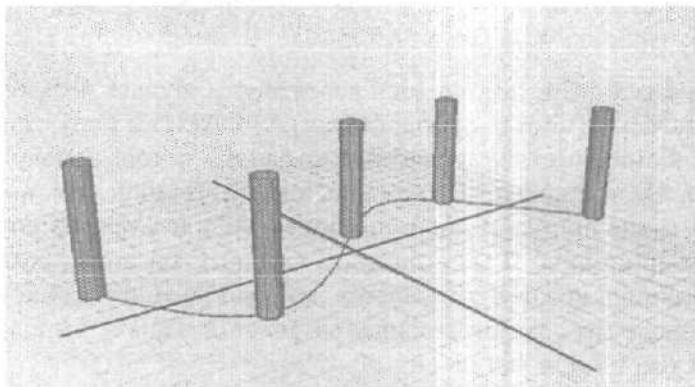


Рис. 4.9. Цилиндры как результат выполнения команды posts

4.4.2. Модуль Posts1

Мы начнем этот раздел с простой версии команды **posts**, которую будем постепенно достраивать. Первая версия команды получит название **posts1**.

1. Откройте рабочую среду Posts1.
2. Скомпилируйте ее и загрузите полученный модуль **posts1.mll** в среде Maya.
3. Откройте сцену **PostsCurve.ma**.
4. Выделите кривую.
5. В строке Command Line наберите следующий текст, а затем нажмите клавишу **Enter**.

posts1

Будут созданы пять цилиндров, расположенных вдоль кривой.

Подробно рассмотрим исходный код этой команды.

Модуль: Posts1

Файл: postsCmd1.cpp

```
class Posts1Cmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& );
    static void *creator() { return new Posts1Cmd; }
};
```

```
MStatus Posts10Cmd::doIt( const MArgList & )  
{  
    const int nPosts = 5;  
    const double radius = 0.5;  
    const double height = 5.0;  
  
    MSelectionList selection;  
    MGlobal::getActiveSelectionList( selection );  
  
    MDagPath dagPath;  
    MFnNurbsCurve curveFn;  
    double heightRatio = height / radius;  
  
    MItSelectionList iter( selection, MFn::kNurbsCurve );  
    for ( ; !iter.isDone(); iter.next() )  
    {  
        iter.getDagPath( dagPath );  
        curveFn.setObject( dagPath );  
  
        double tStart, tEnd;  
        curveFn.getKnotDomain( tStart, tEnd );  
  
        MPoint pt;  
        int i;  
        double t;  
        double tIncr = (tEnd - tStart) / (nPosts - 1);  
        for( i=0, t=tStart; i < nPosts; i++, t += tIncr)  
        {  
            curveFn.getPointAtParam( t, pt, MSpace::kWorld );  
            pt.y += 0.5 * height;  
  
            MGlobal::executeCommand( MString( "cylinder -pivot " ) +  
                pt.x + " " + pt.y + " " + pt.z + " -radius 0.5  
                -axis 0 1 0 -heightRatio " + heightRatio );  
        }  
    }  
}
```

```

        }

    }

    return MS::kSuccess;
}

MStatus initializePlugin( MObject obj )
{
    MFnPlugin pluginFn( obj, "David Gould", "1.0");
    MStatus stat;
    stat = pluginFn.registerCommand( "posts1", Posts1Cmd::creator );
    if ( !stat )
        stat.perror( "registerCommand failed" );
    return stat;
}

MStatus uninitializedPlugin( MObject obj )
{
    MFnPlugin pluginFn( obj );
    MStatus stat;
    stat = pluginFn.deregisterCommand( "posts1" );
    if ( !stat )
        stat.perror( "deregisterCommand failed" );
    return stat;
}

```

Первым шагом является описание нового класса команды **Posts1Cmd**. Этот класс является производным от **MPxCommand** - класса, который служит предком всех создаваемых команд.

```

class Posts1Cmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& );
    static void *creator() { return new Posts1Cmd; }
};

```

Есть только две функции, которые нужно реализовать, - `doIt` и `creator`. Функция `doIt` является виртуальной функцией, вызываемой при выполнении команды. Вся реальная работа в команде возложена на эту функцию. Она производит все операции, которые должна осуществить команда для получения результата.

Функция `creator` служит для создания экземпляра команды. Обратите внимание на то, что она является статической, а потому ее вызов не требует наличия экземпляра класса. На самом деле от вас не требуется, чтобы эта функция была статической функцией-членом или даже называлась `creator`. Функция создания — это просто обычная функция, возвращающая размещенный в памяти экземпляр команды.

```
static void *creator() { return new Posts1Cmd; }
```

Эта функция регистрируется средствами Maya в функции `initializePlugin`, поэтому Maya знает о том, как создать экземпляр команды. При получении запроса на выполнение команды Maya, пользуясь функцией `creator`, сначала размещает в памяти ее экземпляр. Следом вызывается принадлежащая экземпляру команды функция `doIt`. Каждый раз, когда вы станете выполнять команду, Maya будет повторять эти шаги. А значит, всякий раз при запросе на выполнение команды будет создаваться ее новый экземпляр. Причина этого связана со встроенным в Maya механизмом отмены/повторного выполнения. Вскоре он станет предметом нашего рассмотрения.

Так как всю фактическую работу выполняет функция `doIt`, сейчас мы опишем ее более подробно. Первый фрагмент функции содержит код инициализации количества стоек, а также их радиуса и высоты.

```
MStatus Posts1Cmd::doIt ( const MArgList & )  
{  
    const int nPosts = 5;  
    const double radius = 0.5;  
    const double height = 5.0;
```

Далее создается список объектов, выделенных в настоящий момент. Для хранения списка используется объект `selection`.

```
MSelectionList selection;  
MGlobal::getActiveSelectionList( selection );
```

К сожалению, примитив `cylinder` не допускает явного указания его `высоты`. Вместо этого, высота цилиндра определяется значением `heightRatio`. Это отношение высоты цилиндра к его ширине. `heightRatio` рассчитывается на основе желаемой высоты и радиуса цилиндра.

```
double heightRatio = height / radius;
```

Далее организуется цикл по списку выделенных объектов. Этой цели служит объект **MItSelectionList**. Так как команду интересуют исключительно NURBS-кривые, в цикле указан *фильтр*, который исключает все прочие объекты, такими кривыми не являющиеся. Объект **iter** инициализируется объектом **selection**, который был создан ранее.

```
MItSelectionList iter( selection, MFn::kNurbsCurve )
```

Цикл по всем NURBS-кривым выполняется, как показано ниже. Поскольку итератор просматривает все выбранные NURBS-кривые, вы можете воспользоваться командой **posts1** для работы с несколькими кривыми, и она построит цилиндры на каждой из них.

```
for ( ; !iter.isDone(); iter.next() )  
{
```

Для идентификации текущей NURBS-кривой определяется полный путь к ней по ОАГ.

```
iter.getDagPath( dagPath );
```

Путь по ОАГ дополняется функциональным набором для работы с NURBS-кривыми **MFnNurbsCurve**. Таким образом, все дальнейшие операции набора функций будут выполняться в отношении объекта, который определяется путем по ОАГ.

```
curveFn.setObject( dagPath );
```

NURBS-кривая является параметрическим примитивом. Ее расчет при известном значении параметра *t* позволяет получить точку, лежащую на кривой. Значение *t* обычно находится в пределах от 0 до 1, однако диапазон изменения параметра кривой может быть произвольным. Следующий фрагмент кода определяет начало и конец этого диапазона.

```
double tStart, tEnd;  
curveFn.getKnotDomain( tStart, tEnd );
```

Наша команда порождает *nPosts* стоек, расположенных по всей длине кривой. Диапазон изменения параметра делится на необходимое число стоек. Так как и в начале, и в конце кривой должна находиться стойка, кривая будет иметь *nPosts - 1* сегмент.

```
double tIncr = (tEnd - tStart) / (nPosts - 1);
```

Следующий фрагмент функции действительно является ее ядром. Стойки устанавливаются на кривой по всей ее длине при каждом очередном значении параметра *t*.

```
for( i=0, t=tStart; i < nPosts; i++, t += tIncr)
{
```

Функция `getPointAtParam` возвращает точку, лежащую на кривой и соответствующую тому или иному значению параметра. В данном примере запрашивается точка, описанная в мировых координатах (`MSpace::kWorld`), а не в принятых по умолчанию координатах объекта (`MSpace::kObject`). Это делается потому, что каждая стойка должна занять окончательное положение в мировом пространстве независимо от иерархии преобразований кривой,

```
curveFn.getPointAtParam( t, pt, MSpace::kWorld );
```

По умолчанию опорной точкой цилиндра является его центр. Так как основание цилиндра должно покоиться на кривой, надо переместить его опорную точку на половину значения высоты.

```
pt.y += 0.5 * height;
```

Теперь, когда всевозможные параметры нашей команды готовы, можно выполнять команды MEL, необходимые для фактического создания цилиндров. Для построения цилиндра с данной опорной точкой (`-pivot`), данным радиусом (`-radius`) и данным коэффициентом высоты (`-heightRatio`) используется команда `cylinder`. Чтобы из кода на C++ выполнить оператор MEL, вызовите функцию `MGlobal::executeCommand`. Ее входом является строка операторов MEL, которые необходимо выполнить.

```
MGlobal::executeCommand( MString( "cylinder -pivot " ) + pt.x + " " + pt.y
+ " " + pt.z + " -radius " + radius + " -axis 0 1 0 -heightRatio "
heightRatio );
```

```
}
```

Действительно, выполнение операторов MEL в модуле на языке C++ можно увидеть достаточно часто. Во многих случаях это имеет больший смысл, нежели попытки выполнить те же операции с опорой на ряд вызовов C++ API. Кроме того, имеется ряд команд MEL, которые отсутствуют в C++ API, а потому вам не остается ничего иного, как выполнять эти команды именно на языке MEL.

4.4.3. Поддержка аргументов

Команда `posts1` реализована весьма неплохо, однако она не позволяет задавать разное число цилиндров на кривой или устанавливать радиус и высоту каждого из цилиндров. Сейчас эти параметры заданы в виде фиксированных значений в тексте функции `doIt`. Следующая команда `posts2` расширяет нынешний вариант и позволяет указывать в командной строке разнообразные параметры.

1. Откройте рабочую среду Posts2.
2. Скомпилируйте ее и загрузите полученный модуль `posts2.mll` в среде Maya.
3. Откройте сцену `PostsCurve.ma`.
4. Выделите кривую.
5. В строке Command Line наберите следующий текст, а затем нажмите клавишу Enter.

```
posts2 -number 10 -radius 1
```

Вдоль кривой будут установлены десять цилиндров. Каждый из них теперь шире прежнего.

Заметьте, что высота стоек не изменилась. Поскольку она не указана, команда использует значение по умолчанию. Далее мы рассмотрим те конкретные изменения, которые были внесены в текст команды.

4.4.4. Модуль Posts2

Модуль: Posts2

Файл: posts2Cmd.cpp

```
/*  
 * MStatus Posts2Cmd::doIt( const MArgList &args )  
{  
     int nPosts = 5;  
     double radius = 0.5;  
     double height = 5.0;  
  
     unsigned index;  
     index = args.flagIndex( "n", "number" );  
     if( MArgList::kInvalidArgIndex != index )  
         args.get( index+1, nPosts );
```

```
index = args.flagIndex( "r", "radius" );
if( MArgList::kInvalidArgIndex != index )
    args.get( index+1, radius );

index = args.flagIndex( "h", "height" );
if( MArgList::kInvalidArgIndex != index )
    args.get( index+1, height );
```

Так как основные изменения сосредоточены исключительно в начале функции `doIt`, проанализируем только эту часть кода. Сама функция `doIt` не изменилась. Ее единственным входом по-прежнему является ссылка на `MArgList`. В предыдущей команде `posts1` этот аргумент функции попросту игнорировался. Класс `MArgList` предназначен для хранения списка аргументов, переданных команде. Этот класс позволяет получать *флаги и значения* параметров. Флаги – это имена параметров, указанные после черточки. Флаг `-radius` определяет параметр радиуса. Аргумент, записанный вслед за флагом, обычно содержит значение, присваиваемое данному параметру. В нашем случае это значение равно 1.

Обратите внимание на то, что параметры (`number`, `radius`, `height`) больше не являются фиксированными значениями. Вместо этого они инициализируются соответствующими значениями по умолчанию. В случае если один из параметров не установлен в командной строке явным образом, будет использовано значение по умолчанию. В данном примере в командной строке не был указан параметр высоты, поэтому он принимает стандартное значение, равное 5.0.

```
int nPosts = 5;
double radius = 0.5;
double height = 5.0;
```

Так как практически все параметры имеют значение по умолчанию, в командной строке их задавать **необязательно**. Наличие флага параметра, по сути, следует проверять. Индекс аргумента, содержащего указанный флаг, возвращает функция `flagIndex`. Флаги существуют в двух вариантах: краткой и полной форме. Каждая из них может употребляться пользователем, а потому для проверки наличия флага параметра `number` функции `flagIndex` передаются обе строки: `"n"` и `"number"`.

```
index = args.flagIndex( "n", "number" );
```

Если в командной строке флага нет, index принимает значение `MArgList::kInvalidArgIndex`.

```
if( MArgList::kInvalidArgIndex != index )
```

Если же флаг имеет допустимое значение, то указанный вслед за ним аргумент считывается. Он имеет индекс `index+1`. Его значение сохраняется в соответствующей переменной.

```
args.get( index+1, nPosts );
```

Те же шаги выполняются и для остальных флагов. Если эти флаги установлены, их значения считаются из командной строки. В противном случае используются значения по умолчанию.

Те, кто знаком с конструкциями языка С `argc` или `argv`, обнаружат их функциональное сходство с классом `MArgList`. При этом необходимо отметить, что первый аргумент в `MArgList` является первым аргументом команды, а не ее именем, как при работе с `argv`.

В этом примере количество командных параметров сравнительно невелико, поэтому нам было проще проверять каждый из них в отдельности. Более сложные команды могут иметь намного больше параметров, а потому применение класса `MArgList` быстро может стать обременительным. Maya предлагает другой, более совершенный механизм синтаксического разбора флагов аргументов и получения их значений.

4.4.5. Модуль Posts3

Теперь мы изменим команду `posts2` так, чтобы в ней использовались классы `MSyntax` и `MArgDatabase`. Оба названных класса обеспечивают больше гибкости в плане числа и типов параметров, которые вы можете использовать. Кроме того, они обеспечивают более качественную проверку типов аргументов. При написании надежно работающих команд следует действительно отдавать предпочтение именно этим классам. Класс `MArgList` по этой причине используется нечасто, а в случае его применения он служит для написания простых команд.

◆ Откройте рабочую среду `Posts3`.

Модуль: Posts3

Файл: posts3Cmd.cpp

В основе класса `Posts3Cmd` лежит класс `Posts2Cmd`. К нему добавлена следующая статическая функция. Ее задача - вернуть соответствующий команде объект `MSyntax`.

```
static MSyntax newSyntax();
```

Класс **MSyntax** предоставляет удобные средства задания всех возможных параметров вашей команды. Следующий код определяет в краткой и полной форме набор флагов, которые будет принимать команда.

```
const char *numberFlag = "-n", *numberLongFlag = "-number";
const char *radiusFlag = "-r", *radiusLongFlag = "-radius";
const char *heightFlag = "-h", *heightLongFlag = "-height";
```

Далее указываются типы данных аргументов. Пользуясь функцией `addFlag`, вы можете привести до шести различных типов данных, указываемых вслед за флагом. В этом примере параметр `number` принимает единственное значение `long`, в то время как параметры `radius` и `height` принимают единственное значение `double`.

```
MSyntax Posts3Cmd::newSyntax()
```

```
{  
    MSyntax syntax;  
    syntax.addFlag( numberFlag, numberLongFlag, MSyntax::kLong );  
    syntax.addFlag( radiusFlag, radiusLongFlag, MSyntax::kDouble );  
    syntax.addFlag( heightFlag, heightLongFlag, MSyntax::kDouble );  
    return syntax;  
}
```

Начало функции `doIt` такое же, как и прежде.

```
MStatus Posts3Cmd::doIt ( const MArgList &args )  
{  
    int nPosts = 5;  
    double radius = 0.5;  
    double height = 0.5;
```

Теперь в ней используется класс **MArgDatabase**, предназначенный для синтаксического разбора и выделения отдельных флагов и их значений. **MArgDatabase** инициализируется функцией `syntax()` и объектом типа **MArgList**. Функция `syntax()` возвращает объект синтаксиса данной команды. Здесь `syntaxQ` возвращает объект **MSyntax**, подготовленный функцией `newSyntax()`.

```
MArgDatabase argData( syntax(), args );
```

Выполняется поочередная проверка установки каждого флага. Если флаг установлен, то его значение считывается из командной строки, а связанная с ним переменная инициализируется.

```
if( argData.isFlagSet( numberFlag ) )
    argData.getFlagArgument( numberFlag, 0, nPosts );

if( argData.isFlagSet( radiusFlag ) )
    argData.getFlagArgument( radiusFlag, 0, radius );

if( argData.isFlagSet( heightFlag ) )
    argData.getFlagArgument( heightFlag, 0, height );
```

Чтобы системе Maya стало известно об использовании вашего собственного нестандартного объекта **MSyntax**, вы должны об этом ей сообщить. В теле `initializePlugin` функция `registerCommand` вызывается с дополнительным параметром - функцией `newSyntax`.

```
stat = pluginFn.registerCommand( "posts3",
                                  Posts3Cmd::creator,
                                  Posts3Cmd::newSyntax );
```

Хотя этот пример может и не вполне убедить вас; в том, что **MSyntax** и **MArgDatabase** действительно стоят **ТОГО**, чтобы их использовать, на практике они предоставляют гораздо больше функциональных возможностей по сравнению с **MArgList**. Класс **MSyntax** позволяет указать, что в качестве параметров могут использоваться объекты определенных типов. Работая с классом **MSyntax**, вы можете автоматически включать в список аргументов команды те элементы, которые выделены в настоящее время. В целом, это гораздо более надежный и мощный инструмент синтаксического разбора, извлечения и проверки аргументов **ваших** команд.

4.4.6. Организация справочной системы

Нет почти ни одной команды Maya, которая не поддерживала бы определенную форму помощи. Помощь обычно имеет вид того или иного текста справочного характера, который объясняет, что делает команда и каковы ее параметры. Такие базовые возможности организации справки предоставляются командой `help`. Эта команда выводит на экран краткое и лаконичное справочное описание любой заданной команды.

- * В редакторе **Script Editor** наберите следующий текст, а затем выполните его `help sphere;`

Результат ваших действий будет таким:

// Result:

Synopsis: `sphere [flags] [String...]`

Flags:

<code>-e -edit</code>			
<code>-q -query</code>			
<code>-ax -axis</code>	Length	Length	Length
<code>-cch -caching</code>	on off		
<code>-ch -constructionHistory</code>	on off		
<code>-d -degree</code>	Int		
<code>-esw -endSweep</code>	Angle		
<code>-hr -heightRatio</code>	Float		
<code>-n -name</code>	String		
<code>-nds -nodeState</code>	Int		
<code>-nsp -spans</code>	Int		
<code>-o -object</code>	on off		
<code>-p -pivot</code>	Length	Length	Length
<code>-po -polygon</code>	Int		
<code>-r -radius</code>	Length		
<code>-s -sections</code>	Int		
<code>-ssw -startSweep</code>	Angle		
<code>-tol -tolerance</code>	Length		
<code>-ut -useTolerance</code>	on off		

//

На экране появится полный перечень всех флагов команды и допустимых для них типов данных. Помимо работы со встроенными командами Maya, `help` может использоваться и при работе с командами, которые написали вы.

Автоматическая справка

Если вы используете объект `MSyntax`, а это и в самом деле так, коль скоро вы добавили новую функцию `newSyntax()`, то команда `help` способна определить, какие флаги и значения принимает ваша команда. Она строит список этих значений и флагов автоматически.

1. Откройте рабочую среду **Posts3**.
2. Скомпилируйте ее и загрузите полученный модуль **posts3.mll** в среде Maya.
3. В редакторе **Script Editor** наберите следующую команду, а затем выполните ее.

```
help posts3;
```

На экране появится следующий текст справки:

```
// Result:
```

Synopsis: **posts3 [flags]**

Flags:

-h	-height	Float
-n	-number	Int
-r	-radius	Float

```
//
```

Модуль Posts4

В дополнение к быстрой помощи, которая автоматически формируется командой **help**, вы можете создать и отобразить свое собственное справочное сообщение. В следующем примере к команде будет добавлен флаг помощи **-h/-help**. Для предотвращения любых конфликтов краткая форма параметра **height** была переименована из **-h** в **-he**. Краткая форма параметра **может** содержать до трех символов.

1. Откройте рабочую среду **Posts4**.
2. Скомпилируйте ее и загрузите полученный модуль **posts4.mll** в среде Maya.
3. В редакторе **Script Editor** наберите, а затем выполните следующую команду:

```
posts4 -h;
```

На экран будет выведено следующее справочное сообщение:

```
// Result:
```

The **posts4** command is used to create a series of **posts** (cylinders) along all the selected curves.

It is possible to set the number of **posts**, as well as their width and height.

For further details consult the help documentation.

For quick help instructions use: **help posts4 //**

[Команда posts4 предназначена для создания стоек (цилиндров) вдоль всех выделенных кривых. Вы можете задавать количество стоек, а также их ширину и высоту. За более подробной информацией обращайтесь к справочной документации. Для доступа к быстрой помощи используйте команду `help posts4`]

Реализация своей подсказки потребует от вас внесения лишь нескольких простых дополнений в текст существующей команды.

Модуль: Posts4

Файл: Posts4Cmd.cpp

Добавьте новый флаг помощи⁶.

```
const char *helpFlag = "-h", *helpLongFlag = "-help";
```

В функции `newSyntax()` присоедините новый флаг к объекту `MSyntax`.

```
syntax.addFlag( helpFlag, helpLongFlag );
```

Затем опишите справочное сообщение, которое будет выведено на экран.

```
const char *helpText =
"\nThe posts4 command is used to create a series of posts
(cylinders) along all the selected curves."
"\nIt is possible to set the number of posts, as well as
their width and height."
"\nFor further details consult the help documentation."
"\nFor quick help instructions use: help posts4";
```

Вы вправе задать такой текст сообщения, какой пожелаете. Важно, однако помнить о том, что автоматически выдаваемая подсказка должна быть лаконичной, а потому ;; делайте ее слишком большой. Ваше справочное сообщение может заключать в себе любую необходимую, на ваш взгляд, дополнительную информацию. Если же приведенные в справке указания занимают значительно больше места или требуют специальных изображений либо анимации, лучше всего просто отослать пользователя к соответствующей документации.

Последнее требуемое изменение состоит в добавлении нескольких строк к функции `doIt()`. Флаг помощи, как и другие флаги, проверяется на предмет установки. Если он установлен, в качестве результата функции принимается

⁶ А также не забудьте внести изменения в краткую форму параметра `height`. — Примеч. перев.

описанное ранее справочное сообщение. После чего команда немедленно завершается, возвращая признак успешной работы.

```
if( argData.isFlagSet( helpFlag ) )
{
    setResult( helpText );
    return MS::kSuccess;
}
```

Результаты команд могут быть весьма разнообразны. При вычислении длины кривой команда возвращает единственное значение расстояния. При запросе параметра трансляции узла преобразования возвращается последовательность из трех вещественных чисел двойной точности, по одному для каждой оси (x, y, z). Количество и тип результатов определяется самой [командой](#). В данном примере команда posts4 возвращает строку, содержащую текст [справки](#).

Так как команда возвращает строковое значение, вы можете занести его в соответствующую переменную. Следующие операторы языка [MEL](#) позволяют сохранить результат запроса информации о [команде](#).

```
string $text = `posts4 -h`;
print $text;
```

Теперь справочное сообщение содержится в [переменной](#) \$text, поэтому вы можете, к примеру, записать его в файл или вывести в окно на экране.

4.4.7. Отмена и повторное выполнение

Очень важная тема, требующая понимания при составлении [команд](#), - это их обязательная совместимость с реализованным в Maya [механизмом](#) отмены и повторного выполнения. Подобная совместимость чрезвычайно важна для того, чтобы ваша команда правильно работала в среде Maya. Действительно, команда, которая так или иначе модифицирует сцену, но не обеспечивает возможности отмены этих изменений, фактически является [недопустимой](#). Если пользователь попытается отменить результат этой команды, она может вызвать переход Maya в неопределенное состояние.

1. Откройте рабочую среду Posts4.
2. Скомпилируйте ее и загрузите полученный модуль posts4.mll в среде Maya.
3. Откройте сцену PostsCurve.ma.
4. Выделите кривую.

5. В строке **Command Line** наберите, а затем выполните следующий текст:
posts4
6. Выберите в главном меню пункт **Edit | Undo** (Правка | Отменить команду).
При этом ничего не произойдет. Команда posts4 не содержит средств поддержки отмены или повторного выполнения, поэтому ее результат отменить невозможно.

Механизм отмены и повторного выполнения в Maya

Благодаря наличию механизма отмены результатов Maya обладает способностью делать откат последовательности команд. При выполнении команды **MEL** состояние сцены, возможно, как-то меняется. Отмена команды есть не что иное, как аннулирование этих изменений и, соответственно, возврат сцены к тому состоянию, которое предшествовало выполнению команды. Встроенный в Maya механизм отмены, по сути, позволяет осуществлять откат операций, выполненных серией команд.

Очередь отмены

В Maya поддерживается очередь последних выполненных команд. Размер этой очереди определяет количество команд, которые можно аннулировать.

1. Выберите в главном меню пункт **Window | Settings/Preferences | Preferences...**
2. В разделе **Settings** щелкните по элементу **Undo** (Отмена команды).
3. Убедитесь в том, что элемент **Undo** имеет значение **On**.
4. Установите **Queue** (Очередь) в положение **Finite** (Конечная), приайте элементу **Queue Size** (Размер очереди) значение 30.

Если переключатель **Queue** установлен в положение **Infinite** (Бесконечная), то количество команд, которые можно отменить, не ограничено. В то же время такая установка увеличивает расход памяти со стороны Maya, так как система должна сохранять каждую из команд, выполненных с начала сеанса работы, на случай их отмены в будущем. Задание приемлемого, однако конечного размера очереди - это наилучший компромисс между расходованием памяти и потребностью в отмене последовательности команд.

Согласно своим установкам, очередь отмены может содержать три команды, т. е. параметр **Queue Size** равен 3. Очередь изначально пуста. Пустая очередь показана на рис. 4.10. Указатель «последняя команда» содержит последнюю из числа выполненных команд. «Голова» **очереди** – это ее начало, «хвост» очере-

ди - ее конец. Как и в любой другой очереди, элементы добавляются к ее «хвосту» и, в конце концов, продвигаются к ее «голове».

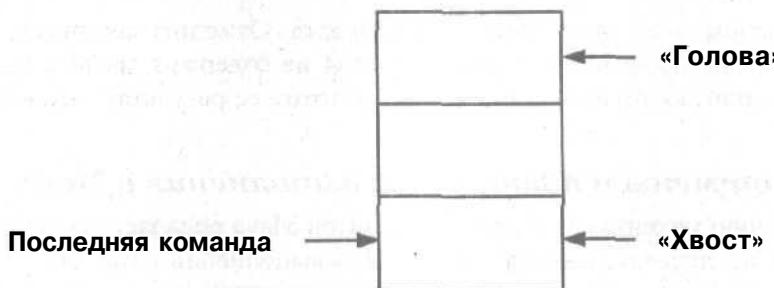


Рис. 4.10. Пустая очередь отмены

Теперь посмотрим на результат выполнения нескольких команд MEL:

```
sphere;
move 10 0 0;
scale 0.5 0.5 0.5;
```

Полученная в результате очередь отмены показана на рис. 4.11.



Рис. 4.11. Заполненная очередь отмены

К «хвосту» очереди присоединяется команда `sphere`. Вслед за ней добавляется команда `move`. Освобождая для нее место, команда `sphere` перемещается на одну ячейку вверх. Затем в «хвост» очереди помещается команда `scale`, прорастывающая обе другие команды еще на ячейку **выше**. Таким образом, очередь отмены заполнена.

Что произойдет с очередью отмены при выполнении еще одной команды?

```
rotate 45;
```

Команда `rotate` ставится в «хвост» очереди, перемещая все остальные команды на одну ячейку вверх. Так как в «голове» очереди свободное пространство отсутствует, команда `sphere` из очереди удаляется. При заполнении очереди наиболее старый элемент удаляется из нее первым. Очередь отмены после выполнения команды `rotate` представлена на рис. 4.12.

Следствием удаления команды из очереди является ее утрата. По сути, команда `sphere` была просто стерта. Ее результат отменить уже невозможно, поскольку в очереди отмены ее больше нет. Теперь вам, должно быть, ясно, что установка размера очереди отмены влияет на количество команд, которые можно отменить. Вам не удастся отменить команду больше, чем их содержится сейчас в очереди отмены.



Рис. 4.12. Очередь отмены после выполнения команды `rotate`

Что произойдет, если выбрать теперь отмену команды? Команда `rotate` будет аннулирована. Последней станет команда `scale`. Если же пользователь выберет отмену еще раз, будет выполнен откат и команды `scale`, а последней станет команда `move`. Полученная в результате очередь отмены показана на рис. 4.13.



Рис. 4.13. Очередь отмены после аннулирования двух команд

Теперь сцена пребывает в том состоянии, в каком она находилась до выполнения команд scale и rotate. Из возможности отменять команды логически вытекает то, что должна быть предусмотрена и возможность их повторного выполнения. Если пользователь решит, что он не хотел отменять масштабирование, то может просто выполнить команду повторно. Команда scale будет восстановлена, став при этом последней. Результат изображен на рис. 4.14.



Рис. 4.14. Очередь отмены после повторного выполнения

Вы можете продолжать повторное выполнение столько раз, сколько команд остается в очереди. Вместо повторного выполнения выполним другую команду: cylinder;

Последняя команда scale стоит в середине очереди, при этом выполняется новая команда, а стало быть, все последующие команды удаляются. Таким образом, команда rotate аннулируется. Если бы очередь была больше, все команды после rotate тоже были бы уничтожены. После этого команда cylinder ставится в конец очереди. Полученная в результате очередь отмены представлена нарис. 4.15.



Рис. 4.15. Очередь отмены после выполнения команды cylinder

Немаловажно понять причину того, почему были удалены последние команды. Помните, что очередь отмены может содержать лишь один возможный путь, ведущий по истории выполнения команд. Делая серию отмен, вы двигались по пути истории, пока не достигли команды scale. Если бы очередь действительно допускала существование более одного пути, то выполнение команды cylinder привело бы к созданию ветви истории. В этот момент стали возможны два пути, следующих за scale. Одна ветвь заканчивалась бы исходной командой rotate, вторая – командой cylinder. Ввиду отсутствия механизма поддержки множественного ветвления, предыдущая ветвь при выполнении новой команды попросту отсекается. Вследствие этого повторное выполнение более ранних команд уже невозможно. Они были отброшены, и на их месте остается лишь последняя выполненная команда.

Кроме того, важно заметить, что очередь отмены существует лишь на протяжении текущего сеанса работы Maya. Она не сохраняется вместе со сценой, поэтому вам не удастся отменить результат операций над файлом, который был сохранен, а затем вновь загружен. При загрузке новой сцены очередь отмены полностью очищается.

Класс MPxCommand для поддержки отмены/повторного выполнения

Познакомившись с принципами работы очереди отмены, обратимся к приемам разработки команд с учетом поддержки их отмены и повторного выполнения. Когда Maya выполняет команду, в памяти размещается ее экземпляр. Этой цели служит принадлежащая команде функция creator. Вслед за ней вызывается функция командного объекта с именем doIt. Она выполняет фактическую работу, свойственную данной команде.

Что произойдет, если после этого пользователь потребует отменить действие команды? В дополнение к функции doIt класс MPxCommand содержит несколько других функций-членов, предназначенных для поддержки аннулирования результатов. Вот эти функции:

```
virtual MStatus undoIt();
virtual MStatus redoIt();
virtual bool isUndoable() const;
```

В примерах предыдущих команд ни одна из этих функций не была реализована, потому что класс MPxCommand содержит свою собственную реализацию этих функций по умолчанию. Стандартная реализация функции isUndoable возвращает значение false. Практический смысл этого состоит в том, что результат

команды не подлежит отмене. В свою очередь, функции `undoIt` и `redoIt` не будут вызваны ни при каких обстоятельствах.

Если предыдущие команды имели необратимый характер, то как же они взаимодействовали с механизмом отмены, имеющимся в составе Maya? Перечислим ряд шагов Maya при выполнении одной из команд, скажем, `posts1`.

1. С помощью функции `creator` создается экземпляр команды `posts1`.
2. После этого для выполнения команды вызывается принадлежащая командному объекту функция `doIt`.
3. Далее, чтобы определить обратимость команды, Maya вызывает функцию `isUndoable`. Так как команда возвращает `false`, ее результат необратим, поэтому она не помещается в очередь отмены, а вместо этого немедленно удаляется.
4. Затем, по причине удаления команды, вызывается деструктор ее объекта.

Почему же после своего выполнения командный объект был немедленно удален? Если команду нельзя отменить, она и не ставится в очередь отмены. Что бы произошло, будь эта команда обратимой? Для придания команде обратимого характера функция `isUndoable` должна быть реализована так, чтобы возвращать значение `true`. Тогда при выполнении команды произойдет следующее:

1. С помощью функции `creator` создается экземпляр команды `posts1`.
2. Вызывается принадлежащая командному объекту функция `doIt`.
3. Далее, чтобы определить обратимость команды, вызывается функция `isUndoable`. Теперь она возвращает `true`, поэтому команда помещается в очередь отмены.

Заметьте, что в этом случае команда не удаляется. Теперь командный объект находится в очереди отмены. Что произойдет, если в этот момент пользователь решит отменить результат? Maya обратится к последней команде в очереди отмены и вызовет ее функцию `undolt`. Эта функция должна аннулировать все изменения, которые внесены командой. По существу, `undolt` должна вернуть Maya именно в то состояние, в котором та находилась до вызова команды. Сцена, к примеру, теперь станет такой, как будто команда никогда не выполнялась. Если пользователь продолжит отмену результатов, функция `undolt` будет вызываться для каждого следующего командного объекта в очереди. Процесс может продолжаться до тех пор, пока в очереди отмены не останется больше ни одного командного объекта.

Важно отметить тот факт, что командные объекты по-прежнему существуют после вызова функций `undolt`. Они не удаляются потому, что пользователь мо-

жет пожелать выполнить любую из этих команд еще раз. Выбор повторного выполнения ведет к вызову функции `redoIt`, принадлежащей объекту той команды, которая была отменена самой последней. Это эквивалентно повторному выполнению команды. В результате, `redoIt` должна обладать тем же набором возможностей, что и функция `doIt`. Фактически, имена `doIt` и `redoIt` можно считать синонимами.

Обратимые и необратимые команды

Если вы можете спроектировать команду без поддержки ее отмены, как же тогда определить, нужны ли вам средства обеспечения отката? Ответ достаточно прост.

Если команда, так или иначе, изменяет состояние Maya, то она должна предусматривать отмену и повторное выполнение!

Так, вы можете написать команду, которая интерпретирует состояние Maya как доступное только для чтения. К примеру, можно реализовать команду, подсчитывающую количество сфер в составе сцены. Эта команда не изменяет текущего состояния Maya, поэтому ей не требуется поддержка отмены результата. В Maya есть специальное обозначение для команд, не поддерживающих отмену, - это *действия*. Действия представляют собой команды, которые в ходе своей работы не создают объектов и не изменяют состояние Maya, а значит, их можно свободно вызывать в любой момент времени. Действием считается всякая команда, функция `isUndoable` которой возвращает `false`.

Это не означает, что вам не удастся написать команду, которая случайно изменяет состояние Maya, но не обладает средствами обеспечения отката. В этом случае вы должны изменить свою команду так, чтобы она стала обратимой.

Модуль Posts5

Так как предыдущие команды `posts` изменяли состояние Maya, они и в самом деле должны содержать функции отмены и повторного выполнения.

1. Откройте рабочую среду **Posts5**.
2. Скомпилируйте ее и загрузите полученный модуль `posts5.mll` в среде Maya.
- 3. Откройте сцену `PostsCurve.ma`.**
4. Выделите кривую.
5. В строке **Command Line** наберите, а затем выполните следующий текст:

`posts5 -number 7 -radius 1`

По всей длине кривой будут установлены семь цилиндров.

6. Для отмены команды выберите пункт Edit | **Undo** или нажмите клавиши **Ctrl+z**.

Действие команды **posts5** аннулируется. В результате этого вновь созданные цилиндры уничтожаются.

7. Для повтора команды выберите пункт Edit | **Redo** (Правка Повторить команду) или нажмите клавиши **Shift+z**.

Команда будет выполнена повторно, что приведет к появлению семи новых цилиндров,

Посмотрим **теперь**, какие изменения были внесены в команду **posts** для поддержки ее отмены и повторного выполнения.

Модуль: Posts5

Файл: posts5Cmd.cpp

Класс команды описывается, как и прежде, однако в него добавлено несколько новых функций-членов и новых объектов данных.

```
class Posts5Cmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& );
    virtual MStatus undoIt();
    virtual MStatus redoIt();
    virtual bool isUndoable() const { return true; }

    static void *creator() { return new Posts5Cmd; }
    static MSyntax newSyntax();

private:
    MDGModifier dgMod;
}
```

Были добавлены функции-члены **undoIt**, **redoIt** и **isUndoable**. Обратите внимание на реализацию функции **isUndoable**, которая возвращает теперь значение **true**, что указывает на поддержку отмены результата. Самое существенно добавление - это новый член **dgMod**. Он имеет тип **MDGModifier**. Класс **MDGModifier** служит для создания, удаления и модификации узлов

- ✓ **Dependency Graph.** Несмотря на то что для этой цели можно использовать и другие классы, самым большим преимуществом **MDGModifier** является автоматическая поддержка отмены и повторного выполнения всех своих операций. Это убережет вас от необходимости реализовывать то же самое своими силами.

В момент вызова любой функции **MDGModifier**, предназначенной для редактирования **Dependency Graph**, класс сохраняет запись об этом вызове. В действительности при вызове какой-либо из таких функций происходит регистрация той операции, которая должна была выполняться. Класс содержит две дополнительные функции: **doIt** и **undoIt**. На деле, все зафиксированные операции выполняются лишь тогда, когда поступает вызов функции **doIt**. Аналогично, все они операции отменяются в тот момент, когда вызывается функция **undoIt**. Отсюда логически вытекает тот факт, что функцию **undoIt** можно вызывать только после вызова функции **doIt**; в ином случае, вам было бы нечего отменять.

При выполнении команды **posts5** происходит вызов ее функции **doIt**. Эта функция претерпела изменения. Обращение к **MGlobal::execute** было заменено обращением к функции **commandToExecute** класса **MDGModifier**.

```
dgMod.commandToExecute( MString( "cylinder -pivot " ) + pt.x + " " + pt.y
+ " " + pt.z + ' -radius ' + radius + " -axis 0 1 0 -heightRatio " +
heightRatio );
```

Серия вызовов **commandToExecute** позволяет зарегистрировать соответствующую серию операций. В данный момент команды еще не **выполнены**, а всего лишь зарегистрированы. Самая последняя строка функции **doIt** содержит очень важное изменение. Теперь здесь вызывается функция **redoIt**.

```
return redoIt();
```

Реальные действия команды выполняет функция **redoIt**. Первичное выполнение команды (**doIt**) ничем не отличается от повторного (**redoIt**), поэтому вам не придется писать две отдельные функции, реализующие ту же самую операцию. Вместо этого, сами операции команды переносятся в функцию **redoIt**, тогда как **doIt** просто ее вызывает. Функция повторного выполнения описана так, что она осуществляет вызов функции **doIt** объекта **dgMod**. Все операции над графом зависимости зарегистрированы этим объектом в командной функции

`doIt`, поэтому функция `doIt` объекта `dgMod` вызывается для их фактического выполнения.

```
MStatus Posts5Cmd:: redoIt()
{
    return dgMod.doIt();
}
```

Аналогично, функция `undoIt` описана следующим образом:

```
MStatus Posts5Cmd:: undoIt()
{
    return dgMod.undoIt();
}
```

Класс `MDGModifier` отвечает и за отмену всех зарегистрированных операций. Этот пример служит иллюстрацией наиболее правильного подхода к проектированию отменяемых команд. Функция `doIt` должна лишь подготовить и зарегистрировать все операции, которые необходимы команде. Эту информацию надо сохранить в надлежащем классе. Для выполнения самой команды функция `doIt` должна просто вызвать метод `redoIt`. Тот получает хранимую информацию и реализует каждую операцию. Функция `undolt` должна принять на вход аналогичную информацию и отменить все действия, совершенные во время работы `redolt`.

В этом конкретном примере мы организовали серию узлов графа DG (цилиндров), поэтому операция отмены должна просто их удалить. В более сложных командах вам, вероятно, понадобится предварительно занести в память гораздо больше такой информации, как сведения об управляющих точках сетки до деформации или других необходимых данных. Фактически, до выполнения команды вы должны зафиксировать всю информацию, которая требуется, чтобы вернуть Maya в предыдущее состояние. В сложных операциях запись данных, которые вы намерены изменить, может оказаться весьма тягостной. Этого, к сожалению, не избежать.

Проектируя команду, которая не является *действием*, очень важно подумать над *организацией* поддержки ее отмены и повторного выполнения. Часто команду, которая спроектирована как необратимая, трудно изменить позднее, сделав ее обратимой. Создавая свои команды, помните об этом с самого начала, и в дальнейшем вам не придется вносить в них большие изменения. Когда команда реализована, вы можете быть уверены, что она будет бесконфликтно работать со встроенным в Maya механизмом отмены/повторного выполнения.

4.4.8. Редактирование и запросы

В предыдущем разделе команда `posts` была расширена за счет включения в нее нескольких нестандартных флагов, которые позволяли указать такие параметры, как высота и радиус стоек. При выполнении команды используются текущие значения ее параметров. Позднее вам может потребоваться их исправить. В отношении многих параметров **важно**, чтобы пользователь имел возможность запросить их текущие значения и установить новые. Для этого команда должна выполняться в **разных режимах**. Когда выполнение команды нацелено на **создание** чего-то **нового**, она запускается *в режиме создания*. Когда выполнение команды имеет целью возврат значения параметра, она работает *в режиме запроса*. Когда выполняемая команда должна изменить существующий параметр, она функционирует *в режиме правки*. Большинство команд поддерживают один или несколько таких режимов.

Эти разнообразные режимы не являются настоящими, подлинными состояниями команды. В действительности команда не меняет своего состояния. *Она* может делать что угодно и когда угодно. Разные режимы – это лишь соглашение, призванное описать, когда те или иные операции могут выполняться, а когда нет. Ввиду такой распространенности операций создания, правки и запросов для них были разработаны свои собственные соглашения, которым должна следовать каждая команда. Если, согласно общепринятым соглашениям, команда поддерживает режимы запроса и правки, она добавляет к списку своих флагов флаг запроса и флаг редактирования. В краткой и полной форме они описываются, соответственно, как `-q/query` и `-e/edit`.

Команду `sphere` можно вызывать в различных режимах.

1. Создайте новую сцену, выбрав пункт меню **File | New Scene**.
2. В редакторе **Script Editor** выполните следующую команду:

```
sphere -radius 2;
```

Будет построена **NURBS-сфера** радиуса 2. Так как команда `sphere` не отвечает на запрос и не изменяет значение параметра, она выполняется в режиме создания. В результате появляется новый сферический объект.

Теперь увеличим радиус сферы.

3. Выполните следующую команду:

```
sphere -edit -radius 10;
```

Радиус сферы становится больше. В данном случае команда `sphere` была выполнена в режиме правки. Запуск команды не привел к созданию новой сферы, вместо этого были изменены атрибуты существующего объекта.

- Чтобы запросить текущий радиус сферы, выполните следующую команду:

```
sphere -query -radius;
```

Тогда вы увидите результат:

```
// Result: 10 //
```

Команда `sphere` была запущена в режиме запроса. Она не создает новую сферу и не изменяет существующую, а вместо этого возвращает текущее значение запрашиваемого параметра.

Если вызвать команду `sphere` с флагом `-query`, она вернет соответствующий результат. Вы можете сохранить его в переменной и позднее использовать. Важно отметить тот факт, что вам не удастся за один раз запросить целый ряд параметров. Нельзя, например, одновременно запросить радиус сферы `radius` и значение атрибута `startSweep`, так как в одно и то же время команда может вернуть единственное значение.

// Можно ли вернуть radius или startSweep?

```
sphere -query -radius -startSweep;
```

Если же вы хотите запросить несколько параметров, просто разбейте свои запросы на серию командных вызовов, каждый из которых содержит единственный запрос.

```
sphere -query -radius;
```

```
sphere -query -startSweep;
```

Правка в одном и том же командном вызове целого ряда параметра, однако допускается.

```
sphere -edit -radius 2 -startSweep 20;
```

Смешивать режимы в одном и том же вызове запрещено. Поэтому выполнение команды как в режиме правки, так и в режиме запроса одновременно невозможно. Команду нужно запустить несколько раз, занимаясь правкой и запросами по отдельности.

Модуль *Clock*

Чтобы продемонстрировать более сложный пример запросов и редактирования, рассмотрим реализацию команды `clock`. Зная текущее значение времени, она устанавливает стрелки трехмерного циферблата. Пример работы команды `clock` над такими часами показан на рис. 4.16.

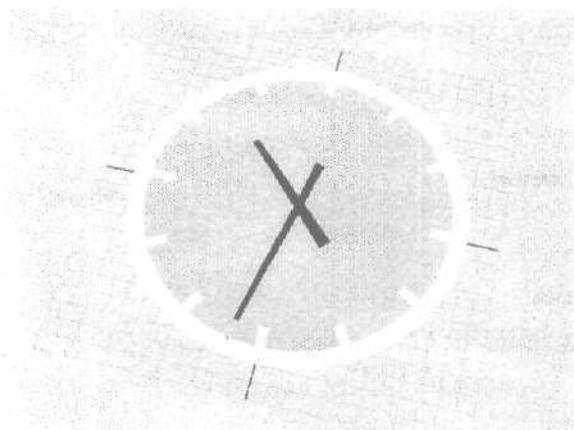


Рис. 4.16. Трехмерные часы, управляемые командой `clock`

1. Откройте рабочую среду Clock.
2. Скомпилируйте ее и загрузите полученный модуль `clockCmd.mll` в среде Maya.
3. Откройте сцену `Clock.ma`.
4. Выделите оба объекта-стрелки: `hour_hand` и `minute_hand`.
5. В редакторе Script Editor выполните следующую команду:
`clock -edit -time 745;`
Часовая и минутная стрелка повернутся и покажут указанное время 7:45,
При записи времени в команде сначала идут часы, а затем минуты: `ччмм`.
Например, 2:31 будет записано как 231, а 3:00 как 300.
6. Выполните следующую команду:
`clock -e -time 1018;`
Стрелки повернутся и покажут заданное время 10:18. Обратите внимание на использование краткой формы флага редактирования (`-e`).
Не зная заранее, который час, вы можете запросить текущее время на циферблате.
7. Выполните следующую команду:
`clock -query -time;`
На экране будет показан результат запроса:
`// Result: 1018 //`

Команда обладает как возможностями отмены, так и возможностями повторного выполнения.

8. Выполните следующую команду:

```
undo;
```

Стрелки вернутся в прежнее положение.

9. Выполните следующую команду:

```
redo;
```

Стрелки опять покажут новое время.

Теперь рассмотрим исходный код команды `clock` более подробно.

Модуль: Clock

Файл: clockCmd.cpp

Команда реализована в классе `ClockCmd`. Как и все прочие команды Maya, она является производной от `MPxCommand`. Ранее было показано, что этот класс имеет средства поддержки отмены и повторного выполнения. Кроме того, в него входит ряд дополнительных данных-членов и методов. Мы будем их пояснять по мере их использования в команде.

```
class ClockCmd : public MPxCommand
{
public:
    virtual MStatus doIt( const MArgList& );
    virtual MStatus undoIt();
    virtual MStatus redoIt();
    virtual bool isUndoable() const;

    static void *creator() { return new ClockCmd; }
    static MSyntax newSyntax();

private:
    bool isQuery;
    int prevTime, newTime;
    MDagPath hourHandPath, minuteHandPath;

    int getTime();
    void setTime( const int time );
};
```

Как и прежде, команда содержит описание принимаемых параметров и тех значений, которые эти параметры могут иметь. Команда `clock` содержит только один флаг параметра `time`, заданный как `-t/-time`.

```
const char *timeFlag = "-t", *timeLongFlag = "-time";
```

```
MSyntax ClockCmd::newSyntax()
{
    MSyntax syntax;

    syntax.addFlag( timeFlag, timeLongFlag, MSyntax::kLong );
```

Наряду с добавлением флага `time`, объект синтаксиса указывает на то, что он принимает как флаг запроса (`query`), так и флаг редактирования (`edit`). Этой цели служат вызовы функций класса **MSyntax**: `enableQuery` и `enableEdit`. Теперь объект синтаксиса команды поддерживает три флага: `time`, `query` и `edit`,

```
syntax.enableQuery(true);
syntax.enableEdit(true);
```

```
return syntax;
```

Принадлежащая команде функция `doIt` является чуть более сложной, поскольку отмена и повторное выполнение команды должны обрабатываться вручную. Для выполнения отката вы должны знать текущее состояние объекта, которое будет изменено командой. Если оно сохранится, команда сможет восстановить исходное состояние объекта, когда пользователь аннулирует ее результат. Аналогично, вы должны точно знать то состояние, в которое объект будет переведен. Это важно как для первичного (`doIt`), так и для повторного выполнения (`redoIt`) команды.

Так как в действительности команда имеет всего лишь один параметр `time`, то и сохраняется именно он. Текущее время хранится в переменной класса с именем `prevTime`. Это значение, которое используется для аннулирования команды. Новое значение времени будет храниться в переменной класса с именем `newTime`. Это значение, которое станет использоваться при первичном и повторном выполнении команды.

```
int prevTime, newTime;
```

Функция `doIt` принимает параметры команды и сохраняет их. Кроме того, она сохраняет в принадлежащей классу переменной `isQuery` признак запуска команды в режиме запроса или в режиме правки.

```
bool isQuery;
```

Так как команда ничего не создает, ей и не нужно поддерживать режим создания. Если команда запущена в режиме правки, она фиксирует новое время. Наконец, в переменные класса `hourHandPath` и `minuteHandPath` записываются выделенные в данный момент объекты, соответствующие стрелкам часов.

```
MDagPath hourHandPath, minuteHandPath;
```

Далее мы поясним разнообразные операции, выполняемые функцией `doIt`.

```
MStatus ClockCmd::doIt ( const MArgList &args )
{
```

```
    MStatus stat;
```

Объект `MArgDatabase` инициализируется, как и прежде. Заметьте, что при невозможности его инициализации команда тоже закончится неудачно. Инициализация завершится с ошибкой, если любой из аргументов команды окажется недопустимым.

```
MArgDatabase argData( syntax(), args, &stat );
if( !stat )
    return stat;
```

Далее устанавливается член данных `isQuery`. Класс `MArgDatabase` имеет в своем составе удобную функцию `isQuery()`, которая возвращает `true`, если флаг запроса был установлен в командной строке. Его установка практически переводит команду в режим запроса. Коль скоро команда `clock` ничего не создает, она работает лишь в двух возможных режимах: в режиме запроса и в режиме правки.

```
isQuery = argData.isQuery();
```

Если команда выполняется в режиме правки, она считывает значение члена данных `newTime`.

```
if( argData.isFlagSet( timeFlag ) && !isQuery )
    argData.getFlagArgument( timeFlag, 0, newTime );
```

В следующем фрагменте кода организуется цикл по всем выделенным объектам и происходит считывание их имен. При обнаружении объекта с именем `hour_hand` или `minute_hand` ведущий к нему путь по ОАГ запоминается.

```
// Получить список текущих выделенных объектов
MSelectionList selection;
MGlobal::getActiveSelectionList( selection );

MDagPath dagPath;
MFnTransform transformFn;
MString name;

// Организовать цикл по узлам преобразования
MITSelectionList iter( selection, MFn::kTransform );
for ( ; !iter.isDone(); iter.next() )
{
    iter.getDagPath( dagPath );
    transformFn.setObject( dagPath );

    name = transformFn.name();
    if( name == MString("hour_hand") )
        hourHandPath = dagPath;
    else
    {
        if( name == MString("minute_hand") )
            minuteHandPath = dagPath;
    }
}
```

Если объекты, соответствующие стрелкам часов, не найдены, команда выдает сообщение об ошибке.

```
// Ни часовая, ни минутная стрелка не выделена
if( !hourHandPath.isValid() || !minuteHandPath.isValid() )
{
    MGlobal::displayError( "Select hour and minute hands" );
    // "Выделите часовую и минутную стрелку"
    return MS::kFailure;
}
```

Затем определяется текущее время, представленное положением обеих стрелок.

```
prevTime = getTime();
```

Если команда пребывает в режиме запроса, достаточно принять текущее время за результат и закончить работу.

```
if( isQuery )  
{  
    setResult( prevTime );  
    return MS::kSuccess;  
}  
  
Если команда находится в режиме правки, то для решения задачи редактирования вызовем функцию redoIt.  
return redoIt();  
}
```

Функции undoIt и redoIt просто вызывают функцию-член setTime. Функция undoIt переводит стрелки часов так, чтобы они показывали предыдущее время, а функция redoIt устанавливает их в соответствии с новым значением.

```
MStatus ClockCmd::undoIt()  
{  
    setTime( prevTime );  
    return MS::kSuccess;  
}  
  
MStatus ClockCmd::redoIt()  
{  
    setTime( newTime );  
    return MS::kSuccess;  
}
```

Функция getTime определяет время, которое показывают часы, зная расположение их стрелок. Понимать эту математику необязательно, однако важно заметить, что функция находит объект hour_hand, а затем определяет угол его поворота относительно оси y. Текущее время определяется величиной поворота. Наконец, время приводится к формату ччмм и выдается в качестве возвращаемого значения.

```
int ClockCmd::getTime()  
{  
    // Найти время по повороту стрелок
```

```
MFnTransform transformFn;
transformFn.setObject( hourHandPath );
MEulerRotation rot;
transformFn.getRotation( rot );

// Определить время и отформатировать его
int a = int(-rot.y * (1200.0 / TWOPI));
int time = (a / 100 * 100) + int( floor( (a % 100) *
(6.0 / 10.0) + 0.5 ) );

return time;
}
```

Функция setTime выполняет вращение объектов **hour_hand** и **minute hand**,
призванное показать заданное время.

```
void ClockCmd::setTime( const int time )
{
    MFnTransform transformFn;

    // Рассчитать часы и минуты
    int hour = (time / 100) % 12;
    int minutes = time % 100;

    // Повернуть часовую стрелку на требуемую величину
    transformFn.setObject(hourHandPath);
    transformFn.setRotation( MEulerRotation( MVector( 0.0, hour +
        (-TWOPI / 12) + minutes * (-TWOPI / 720.0), 0 ) ) );

    // Повернуть минутную стрелку на требуемую величину
    transformFn.setObject(minuteHandPath);
    transformFn.setRotation( MEulerRotation( MVector( 0.0, minutes *
        (-TWOPI / 60.0), 0 ) ) );
}
```

Функция `isUndoable` содержит важное изменение. Обычно она возвращает `true` для любой команды, поддерживающей отмену результата. Вспомните из предыдущего раздела, что `команда`, не являющаяся обратимой, рассматривается как действие. Действие при своем выполнении не изменяет состояния Maya. Запрос параметра `time` отвечает этому требованию. Происходит лишь запрос параметра, поэтому состояние Maya не изменяется. Фактически, функция `isUndoable` возвращает `false`, когда команда выполняется в режиме запроса, и `true` в противном случае, так как тогда она будет работать в режиме правки.

```
bool ClockCmd::isUndoable() const
{
    !
    return isQuery ? false : true;
}
```

Что произойдет, если команда не вернет `false` при выполнении в режиме запроса? Как и в случае с любой обратимой командой, этот командный объект будет поставлен в очередь отмены. Во время отката результата команды ничего не произойдет, так как отмена запроса не приведет к какому-либо восстановлению. Коль скоро ничего не изменилось, значит, нечего и отменять. По сути, возврат `true` при работе в режиме запроса можно считать неопасным. Но так как объект команды будет помещен в очередь `отмены`, большое число запросов приведет к неоправданному заполнению стека отмены. В общем случае, лучший вариант таков: функция `isUndoable` возвращает `true`, если нужно что-нибудь отменить, и `false` в ином случае.

4.5. Узлы

Создавая свои команды, а также пользуясь средствами языка `MEL`, вы уже сейчас можете получить доступ к немалой части возможностей Maya. Однако даже эти методы не позволяют вам внедрить элементы собственной разработки в базовое ядро Maya- граф **Dependency Graph**. Для организации своих фрагментов и их внедрения в этот механизм вам придется создавать узлы. Они непосредственно встраиваются в **Dependency Graph** и потому становятся неотъемлемой частью сцены вообще. Вы можете создать большое количество разных узлов, каждый из которых будет отличаться своим характерным способом обработки или порождения данных.

4.5.1. Модуль GoRolling

Для создания общего представления о написании узлов и их интеграции в среду Maya рассмотрим несложный узел. Он продемонстрирует нам, как управлять вращением колеса. В ходе анимации колеса аниматору проще всего задать начальное и конечное положение объекта. После этого нужно установить ключевые кадры поворота колеса, с тем чтобы при движении оно вращалось. Нетрудно представить себе, что последний этап поддается полной автоматизации. Нами разработан нестандартный **узел**, который располагает информацией о положении колеса и автоматически его поворачивает. Аниматор может использовать новый узел, просто двигая колесо, вращение которого будет осуществляться автоматически. На рис. 4.17 показан объект «колесо», автоматически вращающийся в результате работы нового узла.

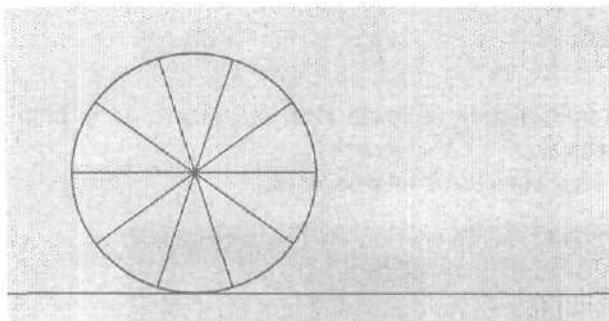


Рис. 4.17. Автоматически вращающееся колесо

Проект в целом включает в себя создание как команды, так и узла. Команда goRolling отвечает за построение узла RollingNode, его добавление в граф DG и установку всех необходимых соединений с атрибутами. Узел выполняет фактическое вращение объекта.

1. Откройте рабочую среду GoRolling.
2. Скомпилируйте ее и загрузите полученный модуль `GoRolling.mll` в среде Maya.
3. Откройте сцену **GoRolling.ma**.
4. Щелкните по кнопке **Play**.
Колесо начнет двигаться по земле, но не будет вращаться.
5. Выделите объект `rim` (обод).

6. Откройте редактор **Attribute Editor**.
Заметьте: параметр трансляции обода уже анимирован, а параметр вращения - нет.
 7. В редакторе **Script Editor** выполните следующую команду:

```
goRolling;
```

В редакторе **Attribute Editor** вы можете обнаружить, что атрибут **Rotate Z** уже анимирован.
 8. Щелкните по кнопке **Play**.
Теперь движение колеса сопровождается вращением.
 9. Щелкните по кнопке **Stop**, а затем выберите инструмент **Move**. Не снимая выделения с объекта **rim**, переместите его в направлении оси **x** (красная стрелка). Сдвиньте обод вперед, затем назад и посмотрите, как происходит автоматический поворот колеса при его перемещении. Изучим теперь происходящее внутри более внимательно.
 10. Откройте окно **Hypergraph**.
- И. Сохраняя выделение узла преобразования объекта **rim**, щелкните по кнопке **Up and Downstream Connections**. Вы увидите окно **Hypergraph**, изображенное на рис. 4.18.

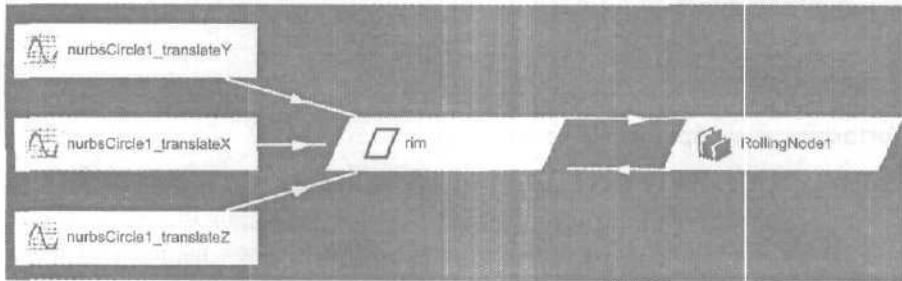


Рис. 4.18. Окно **Hypergraph** после выполнения команды `goRolling`

Этот рисунок говорит о том, что атрибуты трансляции обода: `translateX`, `translateY` и `translateZ`- соответственно управляются **тремя** кривыми анимации: `nurbsCircle1_translateX`, `nurbsCircle1_translateY` и `nurbsCircle1_translateZ`. Таковы стандартные установки, принятые при анимации **любого** атрибута.

Вновь добавленный узел – **RollingNode1**. От узла **rim** к нему ведут два соединения. Возможно, их нелегко увидеть, поскольку одно из соединений изображено практически поверх другого. Кроме того, есть еще одно соединение,

ведущее назад, от узла **RollingNode1** к узлу **rim**. Эти соединения более отчетливо видны на рис. 4.19.

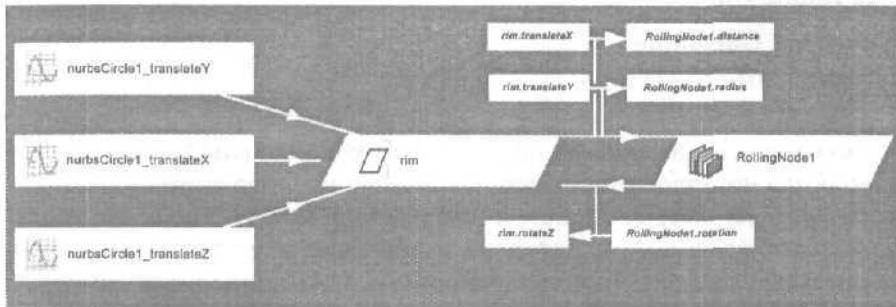


Рис. 4.19. Соединения между узлами

Эта схема позволяет легче понять, что происходит. Значение атрибута трансляции объекта **rim** по оси **x**, **translateX**, передается на вход атрибута **distance** узла **RollingNode1**. Атрибут **distance** определяет, насколько далеко переместился объект. В данном случае это лишь расстояние, на которое продвинулся объект вдоль оси **x** и которое непосредственно приравнивается к значению **translateX** узла **rim**.

Коль скоро созданный обод имеет круглую форму, его узел **transform**, т. е. **rim**, лежит в центре обода. Поэтому его протяженность по оси **y**, **translateY**, эквивалентна радиусу колеса. Отсюда следует, что значение атрибута **translateY** узла **rim** передается атрибуту **radius** узла **RollingNode**. Стало быть, атрибуты **translateX** и **translateY** узла **rim** служат, соответственно, входами **distance** и **radius** узла **RollingNode**. Пользуясь этой парой входных значений, узел **RollingNode1** рассчитывает единственное выходное значение **rotation**. Выход **rotation** передается обратно, атрибуту узла **rim** с именем **rotateZ**. Это значение является результатом вычисления величины вращения, произведенного узлом **RollingNode1**.

Перемещение колеса вперед и назад было причиной изменения значения **translateX**, которое, в свою очередь, приводило к смене значения на входе **distance** узла **RollingNode**. Тогда узел повторно вычислял новое значение **rotation**, которое затем передавалось значению **rotateZ** узла **rim**. Конечный результат этого процесса состоит в том, что при перемещении объекта **rim** вдоль оси **x** его атрибут вращения по оси **z** изменяется.

Теперь проясним подробности, касающиеся узла **RollingNode**. Проект **GoRolling** содержит описание нестандартной команды `goRolling` и пользовательского узла **RollingNode**.

Модуль: GoRolling

Файл: GoRollingCmd.h

Команда `goRolling` предназначена для создания нового узла **RollingNode** и его связи с текущими выделенными объектами. В команде `posts` было продемонстрировано создание узлов-цилиндров. Нынешняя команда тоже создает узлы, однако теперь она устанавливает и соединения между ними. Команда `goRolling` поддерживает отмену и повторное выполнение, а это гарантирует, что по запросу ее результаты могут быть аннулированы. Описание класса самоочевидно и логически вытекает из формата предыдущей команды.

```
class GoRollingCmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& );
    virtual MStatus undoIt();
    virtual MStatus redoIt();
    virtual bool isUndoable() const { return true; }

    static void *creator() { return new GoRollingCmd; }
    static MSyntax newSyntax();

private:
    MDGModifier dgMod;
};
```

Команда содержит единственный член данных типа **MDGModifier** с именем `dgMod`, который служит для создания узла **RollingNode**, а также установки необходимых соединений. Как и прежде, **MDGModifier** упрощает обеспечение отмены и повторного выполнения.

Модуль: GoRolling

Файл: GoRollingCmd.cpp

Основные действия команды производятся в функции `doIt`.

```
MStatus GoRollingCmd::doIt ( const MArgList &args )
<
    MStatus stat;
```

Создается список текущих выделенных объектов.

```
MSelectionList selection;
MGlobal::getActiveSelectionList( selection );
```

Далее с учетом фильтра `MFn::kTransform` организуется цикл по всем узлам преобразования.

```
MDagPath dagPath;
MFnTransform transformFn;
MString name;
```

```
// Цикл по всем узлам преобразования
MItSelectionList iter( selection, MFn::kTransform );
for ( ; !iter.isDone(); iter.next() )
{
    iter.getDagPath( dagPath );
    transformFn.setObject( dagPath );
```

Для каждого из узлов `transform` решаются следующие задачи. Путем вызова функции `createNode` класса `MDGModifier` создается новый узел `RollingNode`.

```
MObject rollNodeObj = dgMod.createNode( "RollingNode" );
```

К вновь созданному объекту-узлу присоединяется набор функций `depNodeFn` класса `MFnDependencyNode`. Пользуясь `depNodeFn`, можно обращаться к содержимому этого узла.

```
MFnDependencyNode depNodeFn( rollNodeObj );
```

Доступ к атрибутам узла требует использования *подключений*. Более подробно их сущность будет описана позднее, а пока подключения можно рассматривать как механизм получения значений атрибутов узла. Узел `transform` служит источником подключения `translateX`, узел `RollingNode` - источником подключения `distance`. Для соединения обоих подключений используется функция `connect`, входящая в класс `MDGModifier`.

```
dgMod.connect( transformFn.findPlug( "translateX" ),
                depNodeFn.findPlug( "distance" ) );
```

Те же шаги повторяются и для связывания подключения `translateY` с подключением `radius`.

```
dgMod.connect( transformFn.findPlug( "translateY" ),
                depNodeFn.findPlug( "radius" ) );
```

Наконец, подключение **rotation** узла **RollingNode** соединяется с подключением **rotationZ** узла преобразования.

```
dgMod.connect( depNodeFn.findPlug( "rotation" ),  
                transformFn.findPlug( "rotateZ" ) );
```

Теперь, когда объект **MDGModifier** **dgMod** сформирован после всех необходимых операций над графом **Dependency Graph**, для выполнения реальных действий вызывается функция **redoIt**.

```
    return redoIt();  
}
```

redoIt просто вызывает функцию **doIt** класса **MDGModifier**, которая выполняет все запрашиваемые операции.

```
MStatus GoRollingCmd::redoIt()  
{  
    return dgMod.doIt();  
}
```

Отмена команды опять-таки упрощается благодаря использованию класса **MDGModifier**, обеспечивающего автоматический откат.

```
MStatus GoRollingCmd::undoIt()  
{  
    return dgMod.undoIt();  
}
```

На этом анализ команды подошел к концу, и теперь мы рассмотрим организацию узла.

Модуль: **GoRolling**

Файл: **RollingNode.h**

Пользовательский узел порожден от класса **MPxNode**. Он весьма отличается от команды. Узел не имеет в своем составе функции **doIt**, а вместо этого содержит ряд других функций, самой важной из которых является функция **compute**. Она выполняет всю реальную работу по обработке данных и потому считается «мозгом» любого узла.

```
class RollingNode : public MPxNode  
{  
public:  
    virtual MStatus compute( const MPlug& plug, MDataBlock& data );
```

```
static void *creator();
static MStatus initialize();

Следующие статические переменные-члены - это еще одно важное дополнение узла. Они содержат описание имеющихся в узле атрибутов. Каждому входному и выходному атрибуту узла соответствует одна переменная.

static MObject distance;
static MObject radius;
static MObject rotation;

static MTypeId id;
};
```

Модуль: **GoRolling**

Файл: **RollingNode.cpp**

Каждый тип узла в Maya, не важно, является ли он встроенным или нестандартным, пользовательским типом, обладает уникальным идентификатором, поэтому Maya знает, как создавать узел, а также - как хранить его и восстанавливать при чтении с диска. В качестве такого идентификатора используется класс MTTypeId.

Если системе Maya нужно загрузить узел с диска, она должна знать о том, какой узел записал те или иные данные. Для распознавания узла, которым были созданы данные, вместе с данными сохраняется его идентификатор. При считывании Maya загружает идентификатор узла и порождает новый узел того типа, который был указан в идентификаторе. После этого узел сам считывает свои данные с диска.

Из этого описания ясно, что если узлы двух различных типов имеют одинаковый идентификатор, то при восстановлении данных с диска Maya может создать узел неверного типа. Это особенно важно в отношении двоичных файлов Maya, поскольку они содержат только идентификатор узла, но не имя класса. Если узел был сохранен в двоичном файле с данным идентификатором, а позднее этот идентификатор был изменен, Maya не сможет прочитать данные этого узла из файла. Новый идентификатор узла будет не соответствовать идентификатору, хранящемуся на диске, поэтому Maya окажется неспособной обнаружить владельца первоначальных данных. Итак, если вы сохраняете свои узлы в двоичные файлы Maya, дайте гарантию того, что позднее вы никогда не станете изменять идентификаторы своих узлов. Что же касается ASCII-файлов, в них

смена идентификатора не всегда ведет к подобным проблемам, поскольку эти файлы содержат сами типы узлов, а не их идентификаторы.

Таким образом, важно обеспечить уникальность идентификатора каждого нового узла. Что произойдет, если вы получите файлы Maya от другого программиста, который, как оказалось, пользуется аналогичными идентификаторами в узлах собственной разработки? Это приведет к тем же проблемам с загрузкой данных, о которых упоминалось выше. Чтобы предотвратить назначение узлам аналогичных идентификаторов, компания Alias | Wavefront выдает разработчикам уникальные наборы идентификаторов, которыми те могут пользоваться. Эти числа присваиваются только вам, и никто другой во всем мире не получит те же самые значения. При строгом следовании этому соглашению каждый когда-либо созданный узел будет иметь уникальный идентификатор. В результате у вас никогда не должно возникать никаких проблем, связанных с совместным использованием файлов, поступивших от различных компаний или людей, пользующихся нестандартными узлами.

Поэтому прежде чем начать разработку, как правило, следует обратиться в Alias | Wavefront за набором уникальных идентификаторов. Подробности относительно регистрации в качестве разработчика и получения идентификаторов приведены на Web-сайте Alias | Wavefront по адресу www.aliaswavefront.com.

Если по какой-то причине вы не можете получить уникальные идентификаторы для своих узлов, прежде чем приступить к их созданию, тогда вы можете воспользоваться временными идентификаторами, пригодными «только для внутренней разработки». Такие идентификаторы должны применяться лишь ограниченное время, пока вы ждете получения окончательных значений. Соответственно, они никогда не должны встречаться в узлах, используемых за пределами вашего «производственного участка». Так, вы можете использовать любые временные идентификаторы в диапазоне от 0 до 0xFFFF. Для примера с узлом **RollingNode** выберем произвольный временный идентификатор 0x00333.

```
MTTypeId RollingNode::id( 0x00333 );
```

Важно, чтобы вы как можно скорее заменили это число корректно отведенным для вас идентификатором. Хотя это обстоятельство может показаться вам несущественным, пока вы занимаетесь разработкой узла, оно может вызвать большие проблемы, если вы сохраните идентификатор в файле сцены Maya, а затем позднее его замените. Такое вполне может произойти, если во время предварительного тестирования бета-тестеры будут создавать файлы Maya с вашим узлом, а в дальнейшем, при выпуске окончательной версии модуля, вы

смените его идентификатор. Данные этого узла уже не будут загружаться ни в одном из файлов, созданных в ходе предварительного тестирования.

Следующий раздел кода содержит описание спецификаторов атрибутов узла. Здесь они просто описываются. Их инициализация произойдет позже.

```
MObject RollingNode::distance;  
MObject RollingNode::radius;  
MObject RollingNode::rotation;
```

Следом описана функция `compute`. Она принимает на вход два аргумента: первый из них - это объект **MPlug**, который указывает, какое подключение (какой атрибут узла) требует пересчета; второй - это объект **MDataBlock**, где хранятся текущие данные для работы узла.

```
const double PI = 3.1415926535;  
const double TWOPI = 2.0 * PI;
```

```
MStatus RollingNode::compute( const MPlug& plug, MDataBlock& data )  
{  
    MStatus stat;
```

Узел может иметь произвольное число выходных атрибутов. Поскольку пересчету подлежат лишь выходные атрибуты, к тому же не каждый из них может требовать обновления, Maya вызывает функцию `compute` для каждого из них в отдельности. По этой причине вы должны проверять, какой из выходных атрибутов указан в запросе на обновление. Данный узел имеет только один выходной атрибут **rotation**, поэтому проверяется только он.

```
if( plug == rotation )  
{
```

При выполнении запроса на пересчет атрибута **rotation** сначала считывается набор входных атрибутов. Эти значениячитываются из узла с помощью функции `inputValue` класса **MDataBlock**. Данная функция воспринимает атрибуты узла как доступные лишь для чтения. Поэтому вы можете прочитать их значения, однако вам не удастся их записать. Аналогично, существует функция `outputValue`, которая позволяет установить значение атрибута, но не допускает его считывания. Даже несмотря на то что Maya не различает входные и выходные атрибуты, наличие отдельных функций для работы с входными и выходными значениями позволяет вам более эффективно считывать данные и управлять ими.

Вы должны указать атрибут, который требует считывания. В данном случае вам нужны атрибуты **distance** и **radius**. Функция `inputValue` возвращает объект **MDataHandle**. Он предназначен для доступа к фактическим данным атрибута.

```
MDataHandle distData = data.inputValue( distance );
MDataHandle radData = data.inputValue( radius );
```

Шаги, связанные с получением значений атрибутов, могут показаться нетривиальными, однако для этого есть все причины, о чем мы подробно расскажем в разделе, посвященном классу **MDataBlock**. Данные входных атрибутов можно получить, воспользовавшись объектом **MDataHandle**. Каждый атрибут, как известно, содержит значение типа `double`, поэтому применяется функция `asDouble`.

```
double dist = distData.asDouble();
double rad = radData.asDouble();
```

Результат вычислений записывается в атрибут **rotation**, поэтому требуется получить его описатель. Для создания описателя вызывается функция `outputValue` класса **MDataBlock**. Она возвращает объект **MDataHandle**, который может использоваться для записи атрибута.

```
MDataHandle rotData = data.outputValue( rotation );
```

Атрибут **rotation** вычисляется на основе входных атрибутов **distance** и **radius**. С помощью функции `set` класса **MDataHandle** результат сохраняется в атрибуте **rotation**.

```
rotData.set( -dist / rad );
```

На следующем шаге подключение объявляется достоверным. Это позволит Maya узнать о том, что оно было вычислено повторно и содержит новое текущее значение.

```
data.setClean( plug );
}
```

Сам класс **MPxNode** содержит большое количество атрибутов. Его потомком является **RollingNode**, который их автоматически наследует. Если функция `compute` обнаружит атрибут, который ей неизвестен, она не потребует от производного класса **RollingNode** перерасчета унаследованных атрибутов, а лишь должна будет вернуть значение `MS::kUnknownParameter`. Вслед за этим Maya вызовет функцию `compute` базового класса и проверит, может ли та найти значение атрибута. Таким образом, производные классы должны вычислять только те атрибуты, которые введены ими явно. Все другие атрибуты обрабатываются прямыми или косвенными предками порожденных классов.

```
else
    stat = MS::kUnknownParameter;
Функция compute возвращает значение типа MStatus, указывающее, было ли
вычисление успешным.
return stat;
}
```

Так же, как и команды, узлы имеют функцию создания экземпляра своего класса. Функция `creator` – это статическая функция, которая просто организует новый экземпляр узла.

```
void *RollingNode::creator()
{
    return new RollingNode();
}
```

Функция `initialize` тоже является статической. Она вызывается только тогда, когда узел проходит первичную регистрацию. Регистрация узла будет подробно описана при анализе следующего файла исходного кода.

```
MStatus RollingNode::initialize()
{
```

Задача принадлежащей узлу функции `initialize` состоит в настройке всех его атрибутов. Каждый атрибут узла: `distance`, `radius` и `rotation` – это статический элемент `данных`, поэтому существует только одна копия этих атрибутов, совместно используемая всеми его экземплярами. Так сделано потому, что атрибуты фактически не содержат данных каждого конкретного узла. Напротив, они выступают подобно шаблону организации данных, которым будет пользоваться каждый узел. С технической точки зрения, данные атрибута узла содержатся в `MDataBlock`, а их считывание и запись выполняется объектом `MPlug`. В контексте интерфейса C++ API, атрибут – это всего лишь подробное описание, необходимое для организации атрибута узла. Для создания и редактирование атрибутов в C++ API служит `MAttribute` и производные от него классы.

Maya поддерживает широкий спектр атрибутов, включая простые числовые типы, такие, как `bool`, `float` и `int`, а также более сложные, составные атрибуты. Последние пользуются простыми типами и образуют их более сложные сочетания. Узел `RollingNode` требует в качестве входов лишь несколько простых значений с плавающей запятой. В следующем фрагменте кода описан атрибут `distance`. Для его создания вызвана функция `create`. Атрибут сохраняется как одно значение типа `double`. Эта информация передается классу

MFnNumericAttribute, для чего служит индикатор **MFnNumericData::kDouble**. Атрибут имеет полное и краткое имя – "distance" и "dist", соответственно. Значение по умолчанию равно 0.0. Полученный в результате объект атрибута записывается в статическом элементе данных с именем distance. Атрибут **distance** должен создаваться с использованием класса **MFnUnitAttribute**, однако в этом примере достаточно типа double.

```
MFnNumericAttribute nAttr;  
distance = nAttr.create( "distance", "dist",  
                         MFnNumericData::kDouble, 0.0 );
```

Атрибут **radius** создается аналогичным образом.

```
radius = nAttr.create( "radius", "rad", MFnNumericData::kDouble, 0.0 );
```

Последним атрибутом является атрибут **rotation**. Он отличается от входных атрибутов тем, что содержит угол. Существует немало способов представления углов в Maya. К примеру, их можно задавать в градусах или радианах. Действительно, угол не считается безразмерной величиной. Ввиду того что он может иметь разнообразные представления, его нельзя сохранять как обычное число типа **double**. Вместо этого для описания угла используется объект **MFnUnitAttribute**. Этот класс предназначен для работы с разными типами единиц измерения Maya, в том числе временем, углами и т. д.

```
MFnUnitAttribute uAttr;  
rotation = uAttr.create( "rotation", "rot",  
                         MFnUnitAttribute::kAngle, 0, 0 );
```

Теперь, после создания всех атрибутов, их можно добавлять в узел, вызывая функцию **addAttribute**, входящую в класс **MPxNode**.

```
addAttribute( distance );  
addAttribute( radius );  
addAttribute( rotation );
```

Чтобы сообщить Maya о том, как атрибуты воздействуют друг на друга, вы должны явно заявить об этом, воспользовавшись функцией **attributeAffects** из состава **MPxNode**. Эта функция устанавливает зависимость между атрибутами, тем самым определяя, какие из них считаются входами, а какие – выходами. Изменение входного атрибута вызовет немедленную реакцию тех атрибутов, которые от него зависят. При изменении значения атрибута **distance** или **radius** атрибут **rotation** потребует повторного вычисления. Формально, с помощью функции **attributeAffects** эта связь описывается так:

```
attributeAffects( distance, rotation );
attributeAffects( radius, rotation );

    Инициализация прошла успешно.

return MS::kSuccess;
}
```

Обратите внимание на отсутствие проверки ошибок. Это сделано для того, чтобы данный пример кода был простым и коротким. В реальном модуле вы должны выполнять проверку наличия ошибок при каждом вызове функции и, если один из них закончится неудачно, вернуть значение **MStatus**, указывающее на наличие сбоя.

Итак, команда и узел уже готовы, и вам нужно сообщить Maya об их существовании. После регистрации в системе вы сможете использовать их в своих сценах.

Модуль: GoRolling

Файл: pluginMain.cpp

Чтобы Maya могла загружать и выгружать подключаемый модуль, в нем, как всегда, должны присутствовать функции **initializePlugin** и **uninitializePlugin**. Мы расширили эти функции, и теперь они лучше справляются с проверкой ошибок. Процесс регистрации происходит следующим образом.

```
MStatus initializePlugin( MObject obj )
{
    MStatus stat;
    MString errStr;
    MFnPlugin pluginFn( obj, "David Gould", "1.0", "Any" );
```

Команда по-прежнему регистрируется при помощи функции **registerCommand** класса **MFnPlugin**.

```
stat = pluginFn.registerCommand( "goRolling",
                                  GoRollingCmd::creator );
if ( !stat )
{
    errStr = "registerCommand failed";
    goto error;
}
```

Регистрация узла выполняется немного иначе. Вы должны указать имя типа и уникальный идентификатор. Кроме того, нужно привести функцию создания узла. Наконец, следует задать функцию его инициализации.

```
stat = plugin.registerNode( "RollingNode",
                            RollingNode::id,
                            RollingNode::creator,
                            RollingNode::initialize );

if ( !stat )
{
    errStr = "registerNode failed",
    goto error;
}

return stat;
```

Функция `initialize` вызывается лишь однажды с целью настройки шаблонов всех атрибутов узла. Эта операция сопровождает его первоначальную регистрацию.

При возникновении тех или иных ошибок на экран выводится сообщение.

`error:`

```
    stat.perror( errStr );
    return stat;
```

Для отмены регистрации команды и узла служит функция `uninitializePlugin`.

```
MStatus uninitializePlugin( MObject obj )
{
    MStatus stat;
    MString errStr;
    MFnPlugin pluginFn( obj );
```

Как и в случае команд из предыдущих примеров, отмена регистрации этой команды выполняется с использованием функции `deregisterCommand` класса `MFnPlugin`. Функция принимает на вход имя команды, регистрацию которой предстоит аннулировать.

```
stat = pluginFn.deregisterCommand( "goRolling" );
if ( !stat )
{
    errStr = "deregisterCommand failed";
    goto error;
}
```

Для отмены регистрации узла его идентификатор передается входящей в класс **MFnPlugin** функции `deregisterNode`. Так как идентификатор каждого узла уникален и использовался ранее при регистрации, Maya в состоянии **отменить** регистрацию узла с этим идентификатором.

```
stat = pluginFn.deregisterNode( RollingNode::id );
if( !stat )
{
    errStr = "deregisterNode failed";
    goto error;
}

return stat;
```

Если по ходу функции `uninitializePlugin` возникает ошибка, на экран выводится сообщение.

```
error:

    stat.perror( errStr );
    return stat;
}
```

Этот пример показывает, что создание нового нестандартного **узла – сравни**тельно простое занятие. Как только узел **зарегистрирован**, его можно создавать и внедрять в граф **Dependency Graph**, подобно любому другому узлу. Все операции по считыванию и сохранению данных узла выполняет Maya. Кроме того, она поддерживает все соединения атрибута и выполняет другие операции. Фактически, узел должен лишь гарантировать перерасчет своих выходных атрибутов, когда Maya будет запрашивать их значения.

4.5.2. Модуль Melt

Теперь мы представим чуть более сложный пример. Этот узел имитирует эффект таяния объекта. Точнее говоря, он деформирует объект так, что создается впечатление, будто объект разогревается снизу. Тогда он начинает медленно таять и растекаться.

Исходные объекты показаны на рис. 4.20. Результат таяния изображен на рис. 4.21.

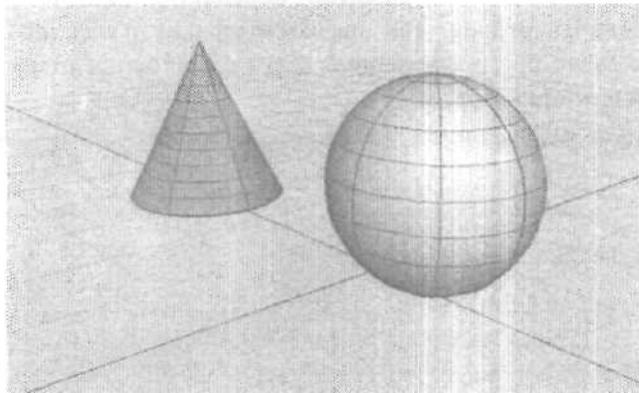


Рис. 4.20. Исходные объекты

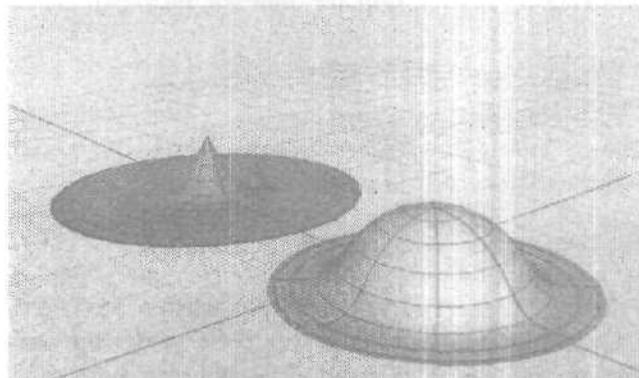


Рис. 4.21. Объекты послетаяния

В этом проекте создаются команда и узел с именем `melt`. Команда реализует цикл по всем выделенным NURBS-поверхностям. Узел `melt` внедряется в историю построения формы. Понятие истории построения мы поясним чуть позже.

Задача узла **melt** - деформировать исходную поверхность так, чтобы она выглядела, как после таяния. Степень таяния определяется атрибутом **amount** узла **melt**. Команда **melt** автоматически анимирует атрибут **amount** узла **melt**, поэтому таяние объекта происходит на протяжении текущего временного диапазона анимации.

1. Откройте рабочую среду **Melt**.
2. Скомпилируйте ее и загрузите полученный модуль **Melt.mll** в среде Maya.
3. Откройте сцену **Melt.ma**.
4. Выделите объекты **nurbsCone1** и **nurbsSphere1**.
5. В редакторе **Script Editor** выполните следующую команду:

```
melt;
```

6. Щелкните по кнопке **Play**.

Оба объекта медленно тают книзу, а затем начинают растекаться наружу.

Команда **melt** создает узел **melt** и внедряет его в историю построения каждого объекта. Чтобы понять происходящее, подробно рассмотрим объект **sphere**. До применения команды **melt** сфера выглядит так, как показано на рис. 4.22.

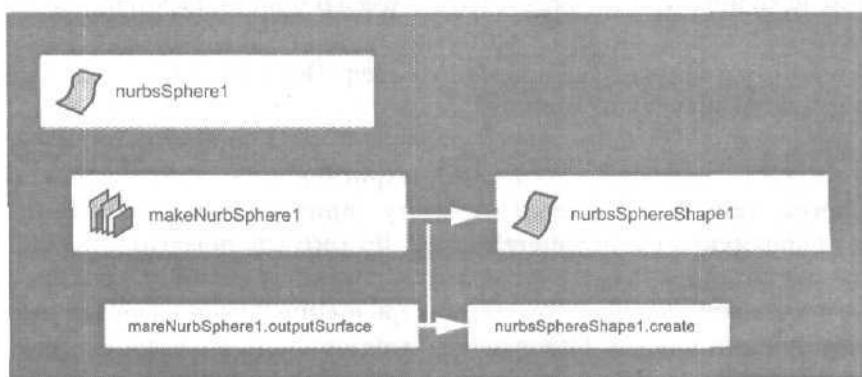


Рис. 4.22. Обычная сфера

Сфера состоит из узла **nurbSphereShape1** категории **shape** и узла **nurbSphere1** категории **transform**. Фактическая NURBS-поверхность появляется благодаря наличию узла **makeNurbSphere1**. Этот узел генерирует NURBS-поверхность сферической формы. Та же самая методика применяется и для других NURBS-примитивов, таких, как цилиндр и тор, которые пользуются, соответственно, узлами **makeNurbCylinder** и **makeNurbTorus**.

Поверхность выводится через атрибут `outputSurface` узла `makeNurbSphere1`. Далее она передается атрибуту `create` узла `nurbsSphereShape1`. Таким образом, узел `nurbsSphereShape1` содержит окончательный вариант NURBS-поверхности. Кроме того, на узел `shape` возлагается ответственность за вывод формы на экран. По существу, именно узел `nurbsSphereShape1` отображает полученную в результате NURBS-поверхность в составе сцены.

В предыдущих примерах атрибуты узлов были сравнительно простыми. Многие из них представляли собой всего лишь отдельные значения типа `double`. Этот узел содержит более сложный атрибут, в котором хранится целая NURBS-поверхность. Данный атрибут с NURBS-поверхностью может храниться в одном узле и соединяться с другими узлами. На практике NURBS-поверхность передается от одного узла другому при посредстве соединения.

В ходе выполнения команды `melt` создается одноименный узел. Он внедряется в историю построения сферы. Так называют последовательность узлов, которые совместно описывают готовую сферу. Для этого каждый из них по очереди генерирует те или иные данные, а затем обрабатывает их; результатом этих действий становится итоговая форма, получаемая по завершении процесса. В истории построения исходной сферы присутствовал только узел `makeNurbSphere1`. При добавлении узла таяния он занял место между `makeNurbSphere1` и конечным узлом `nurbsSphereShape1`. Обновленный граф DG показан на рис. 4.23. Обратите внимание на новый узел `melting2`.

История построения NURBS-сферы включает узел `makeNurbSphere1`, за которым следует `melting2`. Атрибут `outputSurface`, принадлежащий `makeNurbSphere1`, теперь передается атрибуту `inputSurface` узла `melting2`, а не поступает напрямую в `nurbsSphereShape1`. До того как поверхность будет передана узлу `nurbsSphereShape1`, ее может модифицировать узел `melting2`. Единственное изменение состоит в том, что теперь `melting2` расположился между двумя предыдущими узлами. Поверхность передается этому узлу, а значит, он может изменить ее, прежде чем, как и раньше, передать, наконец, узлу `shape`.

Узел `melt` содержит атрибут `amount`, определяющий степень таяния. Его автоматически анимирует команда `melt`. Для этого создается узел `animCurve`, который затем соединяется с атрибутом `amount` узла `melting2`. К счастью, этот узел анимационной кривой не придется создавать вручную, Maya построит его автоматически.

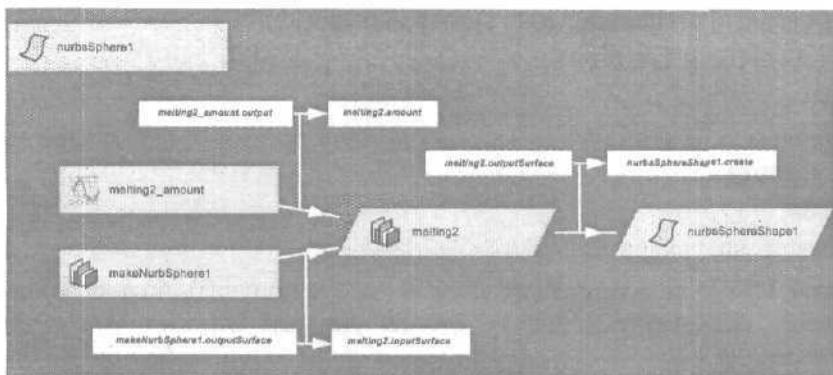


Рис. 4.23. После добавления узла **melt**

Модуль: Melt

Файл: MeltCmd.cpp

Команда **melt** отвечает за создание нужных узлов и последующее соединение с ними. К тому же она реализует анимацию атрибута **amount**. На функцию **doIt** опять-таки возложена задача настройки команды, тогда как фактически работу будет выполнять функция **redoIt**.

```
MStatus MeltCmd::doIt ( const MArgList &args )
{
    MStatus stat;
```

```
    MSelectionList selection;
    MGlobal::getActiveSelectionList( selection );
```

Для анимации атрибута **amount** нужно знать начальную и конечную границу текущего диапазона анимации. Соответствующие этим моментам времени ключевые кадры будут созданы позже.

```
    MTime startTime = MAnimControl::minTime();
    MTime endTime = MAnimControl::maxTime();
```

Организуем цикл по всем NURBS-поверхностям, выделенным в настоящее время. Это позволит применять команду **melt** в отношении любого числа таких форм. Чтобы сообщить классу **MIteSelectionList** о необходимости выполнения цикла исключительно по NURBS-поверхностям, используется фильтр **MFn::kNurbsSurface**.

```
MIItSelectionList iter( selection, MFn::kNurbsSurface );
for ( ; !iter.isDone(); iter.next() )
{
}
```

Среди выделенных объектов нам нужны лишь узлы тех форм, что являются **NURBS**-поверхностями.

```
MObject shapeNode;
iter.getDependNode( shapeNode );
```

Набор функций **MFnDependencyNode** служит, чтобы получить подключение к принадлежащему **shape**-узлу атрибуту **create**. Данный атрибут содержит **NURBS**-поверхность.

```
MFnDependencyNode shapeFn( shapeNode );
MPlug createPlug = shapeFn.findPlug( "create" );
```

Команда **melt** добавляет новый узел **melt** между текущим узлом **makeNurbsSphere1** и узлом **nurbSphereShape1**, и вам необходимо определить, где начинается входящее соединение с атрибутом **create**. Не обнаружив источник этого соединения, в дальнейшем вы не сможете связать его с новым узлом **melt**. Для получения источника соединения служит функция **connectedTo** класса **MPlug**. Она генерирует массив подключений - **источников** соединений. Но зачем нужен массив, если любой атрибут может иметь лишь одно входное соединение? Помимо прочего, функция **connectedTo** используется для запроса всех **целевых** подключений атрибута. Потребность в массиве возникает потому, что любой атрибут может пересыпаться множеству целевых атрибутов.

```
MPlugArray srcPlugs;
createPlug.connectedTo( srcPlugs, true, false );
```

Известно, что атрибут **create** может обладать только одним входным соединением, поэтому оно должно быть первым элементом массива. По-настоящему надежный подключаемый модуль должен содержать проверку **наличия** элементов в массиве. **Массив** может оказаться пустым, и этот случай должен быть выявлен, хотя в **данном** примере такой проверки не будет.

```
MPlug srcPlug = srcPlugs[0];
```

После этого создается новый узел **melt**. Для построения узла можно использовать его уникальный идентификатор. Это более **надежный** и устойчивый к ошибкам метод **задания** конкретного типа узла. Можно воспользоваться и имеющим тип **узла**, правда, это не так надежно.

```
MObject meltNode = dgMod.createNode( MeltMode::ld );
```

Далее организуем подключения к атриутам узла с именами **inputSurface** и **outputSurface**.

```
MFnDependencyNode meltFn( meltNode );
MPlug outputSurfacePlug = meltFn.findPlug("outputSurface");
MPlug inputSurfacePlug = meltFn.findPlug("inputSurface");
```

Теперь новый узел **melt** можно соединять с имеющимися узлами. Однако прежде чем устанавливать новые соединения, нужно разорвать существующее. Иначе говоря, необходимо устранить соединение между узлами **makeNurbsSphere1** и **nurbSphereShape1**. Для этого воспользуемся функцией **disconnect** класса **MDGModifier**. Ее первым аргументом является исходное подключение, вторым - целевое. Совместно оба подключения однозначно определяют конкретное соединение.

```
dgMod.disconnect( srcPlug, createPlug );
```

Теперь проложим соединения между различными подключениями.

```
dgMod.connect( srcPlug, inputSurfacePlug );
dgMod.connect( outputSurfacePlug, createPlug );
```

Новый узел **melt** успешно добавлен. Неприятным следствием применения **MDGModifier** оказалось то, что реально этот класс не создает узлы до тех пор, пока не будет вызвана его функция **doIt**. До вызова этой функции операции над DG лишь регистрировались, но не выполнялись. А значит, мы не можем заранее узнать о том, каким будет точное имя нового узла. Оно станет известно только при его создании. Каждый узел должен носить уникальное имя, поэтому если узел с данным именем уже есть, Maya автоматически присвоит новому узлу уникальное обозначение. Для его составления к заданному имени будет приписано некое число. Однако в следующем фрагменте кода требуется знать точное имя нового узла заранее. Для этого узлу надо присвоить уникальное имя. Используемое кодирование имен, вероятно, не вполне надежно, однако оно удовлетворяет целям данного примера. Узел **melt** меняет свое имя на предварительно составленное обозначение.

```
static i = 0;
MString name = MString("melting") + i++;
dgMod.renameNode( meltNode, name );
```

Следующий шаг заключается в установке ключевых кадров для атрибута **amount** узла **melt**. В начале временного диапазона значение атрибута приравнивается к 0.0, а в конце диапазона - к 1.0. В результате объект, который в первом кадре выглядит как твердое тело, полностью тает к концу последнего кадра.

Узел **animCurve** можно создать вручную и соединить с атрибутом **amount**. После этого ключевые кадры на анимационной кривой можно установить при помощи функционального набора **MFnAnimCurve**. Есть, однако более простой способ. Всю тяжелую работу может выполнить команда MEL **setKeyframe**. При этом операторы MEL готовятся заранее, а затем передаются функции **commandToExecute** класса **MDGModifier**. Первый вызов функции **setKeyframe** устанавливает первый ключевой кадр для анимации атрибута **amount**.

```
MString cmd;
cmd = MString( "setKeyframe -at amount -t " )
    + startTime.value()
    + " -v " + 0.0 + " " + name;
dgMod.commandToExecute( cmd );
```

Аналогичная команда служит для генерации ключа в последнем кадре.

```
cmd = MString("setKeyframe -at amount -t ")
    + endTime.value()
    + " -v " + 1.0 + " " + name;
dgMod.commandToExecute( cmd );
}
```

Наконец, для выполнения реальной работы вызывается функция **redoIt**.

```
return redoIt();
}
```

В теле функции **redoIt** все действия этой функции, зарегистрированные объектом **MDGModifier**, реализуются путем вызова функции **doIt** того же класса.

```
MStatus MeltCmd::redoIt()
{
    return dgMod.doIt();
}
```

По аналогии отменой всех операций управляет функция **undoIt** класса **MDGModifier**.

```
MStatus MeltCmd::undoIt()
{
    return dgMod.undoIt();
}
```

Узнав о том, как команда создает узел **melt** и соединяет его с другими узлами, вы готовы к рассмотрению самого нового узла. Вспомните, что именно узел **melt** занимается фактическим «плавлением» исходной поверхности.

Модуль: Melt

Файл: MeltNode.cpp

Класс **MeltNode** является производным от **MPxNode**, как и все нестандартные узлы DG. Класс узла имеет временный идентификатор.

```
const MTypeId MeltNode::id( 0x00334 );
```

Узел содержит всего три атрибута. Атрибуты **inputSurface** и **outputSurface** представляют собой NURBS-поверхности, атрибут **amount** - число с плавающей запятой.

```
MObject MeltNode::inputSurface;  
MObject MeltNode::outputSurface;  
MObject MeltNode::amount;
```

Функция **compute** - это тот фрагмент кода, где на самом деле происходит плавление объекта. Узел содержит всего один выходной атрибут **outputSurface**, поэтому проверка должна касаться только его.

```
MStatus MeltNode::compute( const MPlug& plug, MDataBlock& data )
```

```
{
```

```
    MStatus stat;
```

```
    if( plug == outputSurface )  
    {
```

Получим описатели всех атрибутов.

```
MDataHandle amountHnd = data.inputValue( amount );  
MDataHandle inputSurfaceHnd = data.inputValue( inputSurface );  
MDataHandle outputSurfaceHnd = data.outputValue( outputSurface );
```

Как известно, атрибут **amount** имеет тип **double**. Поэтому для его считывания служит функция **asDouble** класса **MDataHandle**.

```
double amt = amountHnd.asDouble();
```

Очень важно обратиться к правильной функции **as...**, поскольку вызов неверного метода может привести к краху системы Maya. Важно, по сути, заранее знать тип данных, которые хранятся в заданном атрибуте.

Атрибут **inputSurface** является NURBS-поверхностью. Чтобы прочитать его значение, применяется функция **asNurbsSurface**, описанная в классе **MDataHandle**. Она возвращает объект **MObject**, позволяющий получить данные о NURBS-поверхности.

```
MObject inputSurfaceObj = inputSurfaceHnd.asNurbsSurface();
```

Более сложные типы данных наподобие NURBS-поверхностей Maya хранит как особые геометрические данные. Чтобы получить возможность создавать и сохранять этот вид данных, нужно использовать соответствующий набор функций **MFnGeometryData**. В этом случае подходящим функциональным набором оказывается класс **MFnNurbsSurfaceData**. Пользуясь этим набором функций, вы сможете создать экземпляр данных NURBS-поверхности.

```
MFnNurbsSurfaceData surfaceDataFn;  
MObject newSurfaceData = surfaceDataFn.create();
```

Вслед за выделением места для хранения новой выходной поверхности ее можно получить, скопировав первоначальную входную поверхность.

```
MFnNurbsSurface surfaceFn;  
surfaceFn.copy( inputSurfaceObj, newSurfaceData );
```

Теперь выходная поверхность является точной копией исходной входной поверхности. Первую из них можно деформировать, не оказывая влияния на вторую. Начальный шаг состоит в присоединении к данным функционального набора **MFnNurbsSurface**. Этот функциональный набор можно использовать для доступа к данным о NURBS-поверхности и для их изменения.

```
surfaceFn.setObject( newSurfaceData );
```

С помощью функции **getCVs** класса **MFnNurbsSurface** получим все управляющие вершины (**CV, control vertices**) поверхности и занесем их в массив.

```
MPointArray pts;  
surfaceFn.getCVs( pts );
```

Пользуясь функциями **getCV** и **setCV** класса **MFnNurbsSurface**, можно получить и установить отдельные управляющие вершины поверхности, однако зачастую проще поместить все эти вершины в один массив, а затем производить операции с этим массивом. Такой прием сокращает количество функциональных вызовов и потому работает быстрее.

Вслед за этим рассчитываются вертикальные размеры NURBS-поверхности. По завершении этого шага переменные **minHeight** и **maxHeight** будут содер-

жать соответственно нижнюю (основание) и верхнюю (вершина) границу поверхности по оси *y*.

```
double minHeight = DBL_MAX, maxHeight = DBL_MIN, y;
unsigned int i;
for( i=0; i < pts.length(); i++ )
{
    y = pts[i].y;
    if( y < minHeight )
        minHeight = y;
    if( y > maxHeight )
        maxHeight = y;
}
```

Затем определяется расстояние, на которое переместится каждая управляющая вершина. Оно рассчитывается как относительная часть размера NUR3S-поверхности по вертикали. Значение **amount**, равное 1, соответствует всему этому размеру, значение 0.5 - его половине. Иначе говоря, если **amount** имеет значение 1, то все управляющие вершины, находящиеся на верхней границе объекта, переместятся вниз, к его основанию. Если же это значение равно 0.5, вершины преодолеют половину этого расстояния.

```
double dist = amt * (maxHeight - minHeight);
```

Далее организуется цикл по всем управляющим вершинам NURBS-поверхностей.

```
MVector vec;
double d;
for( i=0; i < pts.length(); i++ )
{
    MPoint &p = pts[i];
```

Каждая точка передвигается на требуемое расстояние по вертикали вниз.

```
p.y -= dist;
```

Те точки, что оказались теперь ниже исходного основания объекта, обрабатываются особо.

```
if( p.y < minHeight )
{
```

Вычислим расстояние от такой точки до основания **объекта**.

```
d = minHeight - p.y;
```

Исходя из *x*- и *z*-координат управляющей вершины, построим двумерный вектор. Этот вектор, если смотреть на него из верхней части **окна**, начинается в центре объекта и заканчивается в соответствующей управляющей **вершине**.

```
vec = MVector( p.x, 0.0, p.z );
```

Найдем длину этого **вектора**.

```
double len = vec.length();
```

Полученная длина позволяет определить, насколько далеко должна «уплыть» **управляющая** вершина. Если расстояние от нее до **центра** объекта очень и очень мало, такая вершина не должна никуда перемещаться. Подобное поведение точек гарантирует, что расчет растекания не приведет к числовым погрешностям и ошибкам деления на ноль. Кроме того, ввиду неподвижности таких точек это предотвращает образование «дыр» в нижней части таких **объектов**, как сфера, тор и др.

```
if( len > 1.0e-3 )
```

```
{
```

Взяв вертикальное расстояние, отложенное вниз от основания объекта, переместим управляющую вершину на это же расстояние по горизонтали и рассчитаем, исходя из **этого**, новую длину вектора. Такое действие имитирует эффект спуска управляющих вершин к основанию и их дальнейшего растекания. С учетом вновь полученной длины найдем новый вектор.

```
double newLen = len + d;
```

```
vec *= newLen / len;
```

Теперь *x*- и *z*-координаты управляющей вершины принимают новое значение.

```
p.x = vec.x;
```

```
p.z = vec.z;
```

```
}
```

Вертикальная координата управляющей вершины никогда не должна опускаться ниже исходного основания объекта.

```
p.y = minHeight;
```

```
}
```

```
}
```

Теперь, когда новое положение всех управляющих вершин найдено, обновим **NURBS**-поверхность.

```
surfaceFn.setCVs( pts );
```

Коль скоро NURBS-поверхность была изменена, очень важно **уведомить** об этом Maya. После любых изменений поверхности следует всегда вызывать функцию `updateSurface` класса `MFnNurbsSurface`.

```
surfaceFn.updateSurface();
```

Передадим новые данные о NURBS-поверхности выходному атрибуту, который содержит сведения о ней.

```
outputSurfaceHnd.set( newSurfaceData );
```

Коль скоро атрибут был рассчитан заново, пометим подключение как дос-троверное.

```
    data.setClean( plug );  
}
```

При поступлении запроса на пересчет иного атрибута, не описанного в дан-ном узле, вернем значение `MS::kUnknownParameter`, с тем чтобы его мог обрабо-тать базовый класс.

```
else  
    stat = MS::kUnknownParameter;
```

Наконец, вернем признак успешного или аварийного завершения функции.

```
    return stat;  
}
```

Атрибуты узла устанавливаются в принадлежащей узлу функции `initialize`.

```
MStatus MeltNode::initialize()  
{
```

```
    MFnNumericAttribute nAttr;  
    MFnTypedAttribute tAttr;
```

Атрибут **amount** описывается как единственное значение типа `double`.

```
    amount = nAttr.create( "amount", "amt",  
                           MFnNumericData::kDouble, 0.0 );
```

Атрибут **amount** подлежит анимации, а потому должен быть сделан **кадри-руемым**. В случае разрешения кадрирования атрибута вы можете устанавливать для него ключевые кадры. По умолчанию кадрирование атрибута не допускает-ся. Придание атрибуту статуса **кадрируемого** также позволяет увидеть его в окне **Channel Control** (Управление каналами).

```
nAttr.setKeyableC(true);
```

Для создания атрибутов `inputSurface` и `outputSurface` используйте класс `MFnTypedAttribute`. Он применяется при построении таких сложных атрибутов, как NURBS-поверхности.

```
inputSurface = tAttr.create( "inputSurface", "is",
                            MFnNurbsSurfaceData::kNurbsSurface );
```

По умолчанию, любой атрибут является видимым. Однако входной атрибут поверхность вручную, по команде melt соединен с другими узлами, поэтому его вывода на экран лучше всего избегать. Даже если атрибут скрыт, соединения с этим атрибутом можно по-прежнему разрывать и вновь устанавливать, для чего используется **Connection Editor**.

```
tAttr.setHidden( true );
```

Атрибут `outputSurface` создается примерно так же, как и атрибут `inputSurface`.

```
outputSurface = tAttr.create( "outputSurface", "os",
                            MFnNurbsSurfaceData::kNurbsSurface );
```

Атрибут `outputSurface` является непосредственным результатом действий над `inputSurface` и `amount`, т. е. результатом расчета новой поверхности. Новая поверхность является производной от входных атрибутов, а потому для ее повторного получения не нужно ничего, кроме подачи входной информации. На основе входных атрибутов новую поверхность всегда можно рассчитать заново, а значит, ее не нужно хранить в файле сцен Maya. Отказ от сохранения этого атрибута экономит дисковое пространство, которое может иметь большое значение для сцен с множеством сложных поверхностей.

```
tAttr.setStorable( false );
```

Как и атрибут `inputSurface`, атрибут `outputSurface` не должен отображаться ни в одном из основных окон редактирования.

```
tAttr.setHidden( true );
```

Затем все атрибуты включаются в состав узла.

```
addAttribute( amount );
addAttribute( inputSurface );
addAttribute( outputSurface );
```

И атрибут `amount`, и атрибут `inputSurface` оказывают прямое влияние на поверхность `outputSurface`. Эта зависимость описывается функцией `attributeAffects` класса `MPxNode`.

```
attributeAffects( amount, outputSurface );
attributeAffects( inputSurface, outputSurface );

return MS::kSuccess;
}
```

Хотя этот пример был сложнее предыдущего, базовые принципы не изменились. Узел по-прежнему определял, какие атрибуты он содержит и какой тип имеют эти атрибуты. Он содержал указание на то, что одни атрибуты влияют на другие. Даже при использовании более сложного типа данных (NURBS-поверхности) никаких существенных изменений для его поддержки нам не понадобилось.

4.5.3. Модуль Ground Shadow

В этом проекте рассматривается создание команды и узла **groundShadow**, которые синтезируют согласованные тени на поверхности земли. При наличии точечного источника света и известного набора объектов подключаемый модуль строит тени, отбрасываемые объектами и спроектированные на земную **поверхность**. Он будет работать в качестве узла, динамически создающего выходную картину, поэтому положение источника света может меняться, а полученные в результате тени будут автоматически перестраиваться. Кроме того, при изменениях геометрии объектов проекции их теней станут автоматически обновляться.

На рис. 4.24 представлены объекты, которые будут отбрасывать тень. На рис. 4.25 показаны тени, полученные в результате работы модуля **groundShadow**.

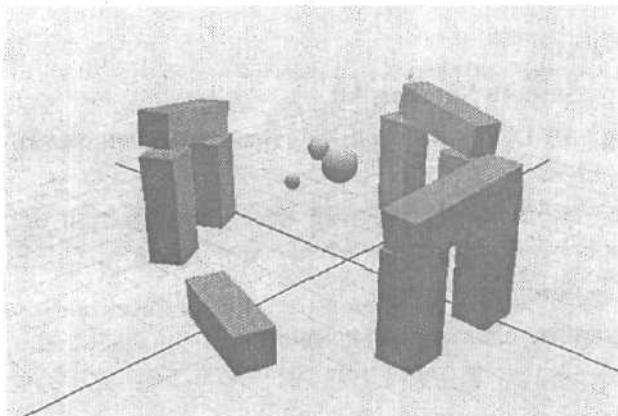


Рис. 4.24. Исходные объекты

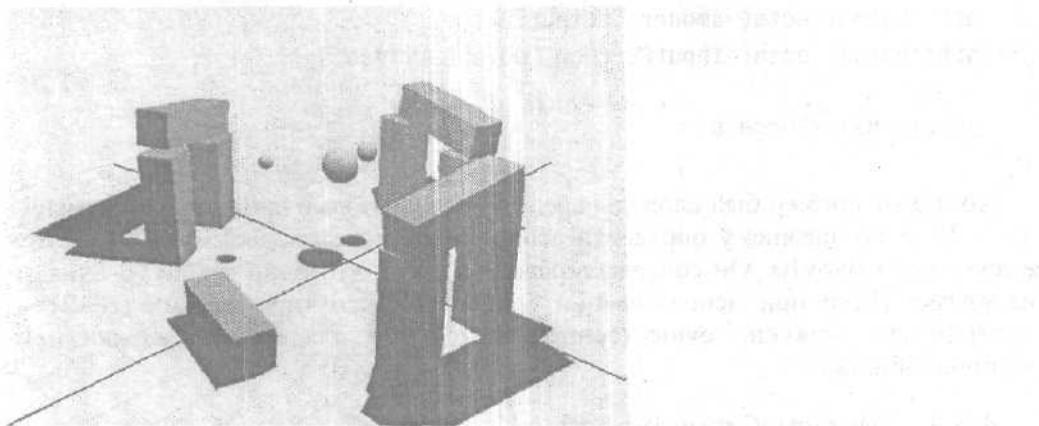


Рис. 4.25. Тени, отбрасываемые на поверхность земли

Важным дополнением к этому примеру является использование атрибута, способного работать с разными типами геометрических объектов. В предыдущем примере мы могли работать лишь с одним типом геометрии, NURBS-поверхностью. Способ настройки атрибута позволял, к примеру, использовать полигональную сетку. Данный проект продемонстрирует принципы настройки атрибута, принимающего геометрические объекты различных типов.

1. Откройте рабочую среду **GroundShadow**.
2. Скомпилируйте ее и загрузите полученный модуль **GroundShadow.mll** в среде **Maya**.
3. Откройте сцену **GroundShadow.ma**.
4. Выберите пункт меню **Shading | Smooth Shading All**.
5. Выберите пункт меню **Lighting | All Lights** (Освещение Все источники света).
6. Щелкните по кнопке **Play**.

По сцене начнет двигаться единственный точечный источник света, а небольшие сферические объекты станут обращаться вокруг друг друга.

7. Выделите все объекты, включая источник света.
8. В редакторе **Script Editor** выполните следующую команду:

```
groundShadow;
```

На поверхности земли вокруг всех выделенных объектов появятся тени.

9. Снимите выделение с каждого объекта,
10. Щелкните по кнопке **Play**.
По мере движения источника света тени согласованно изменяются.

Команда **groundShadow** отвечает за создание различных узлов геометрии, узлов **groundShadow** и соединение их атрибутов.

Команда **groundShadow** просматривает все выделенные объекты и определяет, какие из них являются узлами геометрии. Такие узлы содержат те или иные виды геометрических **объектов**, в том числе **NURBS**-поверхности, кривые, полигональные сетки и т. д. Для каждого выделенного геометрического объекта создается теневой узел. Он является точной копией исходного объекта геометрии. Единственное отличие состоит в том, что геометрия становится гладкой и ровной и искажается с целью создания плоского объекта, принимающего форму тени. Ответственность за реализацию этого искажения лежит на узле **groundShadow**.

Модуль: GroundShadow

Файл: GroundShadowCmd.h

Класс **GroundShadowCmd** порожден от класса **MPxCommand**.

```
class GroundShadowCmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& );
    virtual MStatus undoIt();
    virtual MStatus redoIt();
    virtual bool isUndoable() const { return true; }

    static void *creator() { return new GroundShadowCmd; }
    static MSyntax newSyntax();
```

Основное отличие от предыдущих команд состоит в том, что данный класс содержит элемент типа **MDagModifier**, а не привычный **MDGModifier**. Первый из них является расширением второго, предназначенным для работы с узлами **ОАГ**. Класс **MDagModifier** понимает иерархию **ОАГ** и потому позволяет преобразовывать отношения «родитель- потомок», действующие между узлами **ОАГ**. Данная команда будет заниматься созданием узлов **ОАГ** и изменением отношений между родительскими и дочерними элементами, поэтому вместо **MDGModifier** используется класс **MDagModifier**.

private:

```
    MDagModifier dagMod;  
};
```

Модуль: GroundShadow

Файл: GroundShadowCmd.cpp

Функция `doIt` в составе `groundShadow` выполняет всю работу по подготовке команды.

```
MStatus GroundShadowCmd::doIt ( const MArgList &args )  
{  
    MStatus stat;  
    MSelectionList selection;
```

Важно отметить, что при создании нового узла формы при помощи C++ API не происходит его автоматического присоединения к группе тонировки по умолчанию `initialShadingGroup`. Новую форму нужно, по сути, явно ввести в состав группы. Если этого не сделать, новая форма не пройдет тонировку, и для ее вывода будет использован случайный, часто пурпурный, цвет.

Соответственно, первый шаг заключается в получении узла `initialShadingGroup`.

```
MObject shadingGroupObj;  
selection.clear();  
MGlobal::getSelectionListByName( "initialShadingGroup", selection );  
selection.getDependNode( 0, shadingGroupObj );
```

Объект `MFnSet` присоединяется к группе тонировки, поскольку та, в действительности, является обычным набором.

```
MFnSet shadingGroupFn( shadingGroupObj );
```

Текущие выделенные объекты помещаются в список `selection`.

```
selection.clear();  
MGlobal::getActiveSelectionList( selection );  
MItSelectionList iter( selection );
```

Чтобы узнать, откуда приходит свет, должен быть выбран, по крайней мере, один точечный источник. Следующий шаг - организовать цикл по списку выделенных объектов и выяснить, выбран ли точечный источник света.

```
MDagPath lightTransformPath;  
MPlug pointLightTranslatePlug;  
iter.setFilter( MFn::kPointLight );
```

```
for ( iter.reset(); !iter.isDone(); iter.next() )  
    <  
        iter.getDagPath( lightTransformPath );
```

Результатом запроса пути к выделенному объекту является полный путь, ведущий к нему по ОАГ и заканчивающийся узлом **shape**. Поскольку нам необходимо лишь принадлежащий форме узел **transform**, последний фрагмент пути по ОАГ отбрасывается. Таким образом, полученный в итоге путь содержит все узлы **transform**, находящиеся в иерархии выше узла **shape**.

```
lightTransformPath.pop();
```

Для получения атрибута **translate** узла преобразования создается объект **MPlug**. Он служит для определения позиции источника света.

```
MFnDagNode dagNodeFn( lightTransformPath );  
pointLightTranslatePlug = dagNodeFn.findPlug( "translate" );
```

Коль скоро во внимание принимается только первый выделенный источник света, цикл можно остановить.

```
break;  
}
```

Далее выясним, является ли путь к узлу преобразования источника верным. Даже притом что **MIItSelectionList** организует цикл исключительно по точечным источникам света, путь **lightTransformPath** останется неопределенным, если ни один источник не выделен. По сути, здесь и проверяется этот факт. Если путь некорректен, то ни один точечный источник, должно быть, не выбран и вы можете завершить работу, отобразив сообщение об ошибке.

```
if( !lightTransformPath.isValid() )  
{  
    MGlobal::displayError( "\nSelect a point light." );  
    // "Выберите точечный источник."  
    return MS::kFailure;  
}
```

Теперь необходимая информация об источнике получена. Следующий шаг - организовать цикл по всем выбранным геометрическим объектам и создать для них теневые формы.

```
iter.setFilter( MFn::kGeometric );  
for ( iter.reset(), count=0; !iter.isDone(); iter.next(), count++ )
```

```
{
MDagPath geomShapePath;
iter.getDagPath( geomShapePath );

```

Полный путь к форме получен. Из него удаляется последний элемент – узел **shape**, поэтому в результате путь содержит только родительские узлы **transform**.

```
MDagPath geomTransformPath( geomShapePath );
geomTransformPath.pop();
```

Создаваемый теневой узел **shape** является простой копией исходной формы. Функция копирования вызывается с аргументом тиражирования, равным значению **false**, а значит, создается именно копия, а не экземпляр формы. Поскольку результат копирования позднее будет искажен для создания плоской теневой формы, исходная форма должна оставаться нетронутой. Если бы мы создали экземпляр исходного узла **shape**, то теневой узел **shape** повредил бы форму, используемую ими совместно, а это не входило в наши намерения.

```
MFnDagNode geomShapeFn( geomShapePath );
MObject newGeomTransformObj = geomShapeFn.duplicate( false, false );
```

К новому узлу **shape** присоединяется функциональный набор **newGeomShapeFn** типа **MFnDagNode**. Узел **shape** является первым потомком родительского узла **transform**, поэтому вызывается функция **child** класса **MFnDagNode**.

```
MFnDagNode newGeomShapeFn( newGeomTransformObj );
newGeomShapeFn.setObject( newGeomShapeFn.child(0) );
```

, При создании новой формы она занимает место ниже нового узла **transform**. Однако хотелось бы сделать так, чтобы новый узел **shape** находился под родительским узлом **transform** исходной формы. Такое расположение гарантирует, что при смещении исходной формы теневая форма также придет в движение, поскольку они обе используют одну и ту же иерархию родительских преобразований.

```
dagMod.reparentNode( newGeomShapeFn.object(), geomTransformPath.node() );
```

Как ранее упоминалось, автоматически Maya не помещает новый узел **shape** в группу тонировки по умолчанию, так что это требуется сделать вручную. Ввести узел **shape** в группу **тонировки** – значит просто добавить его в набор с помощью функции **addMember** класса **MFnSet**.

```
shadingGroupFn.addMember( newGeomShapeFn.object() );
```

Следом создается узел **groundShadow**. Он отвечает за деформацию теневой формы и ее превращение в тень.

```
MObject shadowNode = dagMod.MDGModifier::createNode(
    GroundShadowMode::id );
assert( !shadowNode.isNull() );
MFnDependencyNode shadowNodeFn( shadowNode );
```

Далее задаются подключения различных атрибутов узла **groundShadow**.

```
MPlug castingSurfacePlug = shadowNodeFn.findPlug( "castingSurface" );
MPlug shadowSurfacePlug = shadowNodeFn.findPlug( "shadowSurface" );
MPlug lightPositionPlug = shadowNodeFn.findPlug( "lightPosition" );
```

Узел и команда **groundShadow** предназначены для поддержки двух разных типов геометрических объектов: полигональных сеток и NURBS-поверхностей. Каждый тип геометрии имеет различные входные и выходные подключения. В случае с узлом сетки входная геометрическая информация передается узлу через атрибут **inMesh**, тогда как входная информация о геометрии NURBS-поверхности приходит по атрибуту **create**. Чтобы получить правильный атрибут в соответствии с типом геометрического объекта, задайте значение строк **outGeomPlugName** и **inGeomPlugName**. Чтобы определить тип узла **shape**, воспользуйтесь функцией **apiType** класса **MDagPath**.

```
MString outGeomPlugName, inGeomPlugName;
switch( geomShapePath.apiType() )
{
    case MFn::kMesh:
        outGeomPlugName = "worldMesh";
        inGeomPlugName = "inMesh";
        break;

    case MFn::kNurbsSurface:
        outGeomPlugName = "worldSpace";
        inGeomPlugName = "create";
        break;
}
```

Подключение сопоставляется с выходным атрибутом, связанным с геометрической формой,

```
MPlug outGeomPlug = geomShapeFn.findPlug( outGeomPlugName );
```

Важно запомнить, что выходной атрибут геометрического объекта представлен не единственным выходом, а массивом выходных данных. Каждый экземпляр формы имеет отдельный выход, поэтому в выходном массиве геометрий формы, экземпляры которой создавались три раза, содержатся три элемента. Чтобы определить, какому из экземпляров соответствует текущий выделенный объект, вызовите функцию `instanceNumber` класса `MDagPath`. Она возвратит вам номер экземпляра выделенного объекта. Этот номер напрямую соответствует индексу экземпляра в выходном массиве геометрий.

```
unsigned int instanceNum = geomShapePath.instanceNumber();
```

В данный момент подключение `outGeomPlug` указывает на родительский атрибут выходного геометрического объекта, иначе говоря, на массив. Хотелось бы переадресовать его на конкретный элемент массива. Для этого служит функция `selectAncestorLogicalIndex` класса `MPlug`. Она направляет подключение на элемент массива, имеющий заданный индекс.

```
outGeomPlug.selectAncestorLogicalIndex( instanceNum );
```

Нацелим подключение `inGeomPlug` на входной атрибут узла геометрии.

```
MPlug inGeomPlug = newGeomShapeFn.findPlug( inGeomPlugName );
```

Произведем настройку элемента данных `MDagModifier` и свяжем его с различными атрибутами. Положение точечного источника света передается атрибуту узла `shadow`, содержащему позицию освещения. Конкретные атрибуты `GroundShadowNode` будут описаны в следующем разделе. Выходной геометрический объект передается тому атрибуту узла `shadow`, где хранится отбрасывающая тень поверхность. Это именно та поверхность, которая образует тень. Результатом работы узла `shadow` является другая поверхность, путем деформации преобразованная в плоскую тень. Итоговая поверхность передается атрибуту формы-копии, где хранится входная геометрия. Это и есть теневая форма, которая выводится на экран в составе сцены.

```
dagMod.connect( pointLightTranslatePlug, lightPositionPlug );
dagMod.connect( outGeomPlug, castingSurfacePlug );
dagMod.connect( shadowSurfacePlug, inGeomPlug );
}
```

Если ни один из объектов не является сеткой или NURBS-поверхностью, функция завершает свою работу с ошибкой.

```
if( count == 0)
{
    MGlobal::displayError( "\nSelect one or more geometric objects.");
    // "Выделите один или несколько геометрических объектов."
    return MS::kFailure;
}
```

Теперь, когда объект **dagMod** настроен, для выполнения реальных действий можно вызвать его функцию **redoIt**.

```
return redoIt();
}
```

Функции **redoIt** и **undoIt** похожи по своей форме на аналогичные функции предыдущих команд. Единственное отличие состоит в том, что вместо привычного класса **MDGModifier** используется класс **MDagModifier**.

```
MStatus GroundShadowCmd::redoIt()
{
    return dagMod.doIt();
}
```

```
MStatus GroundShadowCmd::undoIt()
{
    return dagMod.undoIt();
}
```

Узел GroundShadow принимает один геометрический объект и создает на выходе другой. Входом объекта также является положение источника света. На основе позиции освещения и входного геометрического объекта узел рассчитывает место, где должна находиться тень этого объекта, отбрасываемая на плоскость земной поверхности. Полученный в результате геометрический **объект** – это всего лишь плоский, искаженный вариант исходного входного объекта. Такая плоская геометрическая фигура выводится на экран в качестве тени объекта. Еще одно дополнение, которого мы не коснулись в своем обсуждении, – атрибут **groundHeight** узла **groundShadow**. Вы можете изменять его для переноса плоскости поверхности на новую высоту.

Модуль: GroundShadow**Файл: GroundShadowNode.h**

Класс **GroundShadowNode** - это стандартный узел DG, поэтому его прямым предком является класс **MPxNode**. Он содержит все обычные функции-члены. Его атрибуты описаны в следующем разделе.

```
class GroundShadowNode : public MPxNode
{
public:
    virtual MStatus compute( const MPlug& plug, MDataBlock& data );

    static void *creator();
    static MStatus initialize();

    static const MTypeId id;

public:
    static MObject lightPosition;
    static MObject castingSurface;
    static HObject shadowSurface;
    static MObject groundHeight;
};
```

Модуль: GroundShadow**Файл: GroundShadowNode.cpp**

Опишем, прежде всего, атрибуты узла. Атрибут **lightPosition** задает положение источника света, входящего в состав сцены. Атрибут **castingSurface** представляет собой поверхность исходного объекта. В атрибуте **shadowSurface** хранится результат- теневая поверхность. Атрибут **groundHeight** содержит единственное вещественное значение, определяющее уровень плоскости земной поверхности.

```
MObject GroundShadowNode::lightPosition;
MObject GroundShadowNode::castingSurface;
MObject GroundShadowNode::shadowSurface;
HObject GroundShadowNode::groundHeight;
```

Функция `compute` отвечает за прием входных атрибутов (`lightPosition`, `castingSurface` и `groundHeight`) и синтез теневой поверхности, записываемой в атрибут `shadowSurface`.

```
MStatus GroundShadowNode::compute(const MPlug& plug, MDataBlock& data)
{
    MStatus stat;
```

Единственным выходным атрибутом этого узла является атрибут **shadow Surface**, который и подлежит проверке.

```
if( plug == shadowSurface )
{
}
```

Для всех входных и выходных атрибутов подготавливаются объекты **MDataHandle**.

```
MDataHandle groundHeightHnd = data.inputValue( groundHeight );
MDataHandle lightPositionHnd = data.inputValue( lightPosition );
MDataHandle castingSurfaceHnd = data.inputValue(castingSurface);
MDataHandle shadowSurfaceHnd = data.outputValue( shadowSurface );
```

Исходная поверхность, отбрасывающая тень, копируется в описатель теневой поверхности `shadowSurfaceHnd`. В данный момент теневая поверхность представляет собой точную копию исходной.

```
shadowSurfaceHnd.copy( castingSurfaceHnd );
```

В следующем разделе кода инициализируются те переменные, которые в дальнейшем будут использоваться при вычислении окончательной геометрической формы тени. Первой определяется высота плоскости поверхности.

```
double gHeight = groundHeightHnd.asDouble();
```

В переменной `lightPoint` типа **MVector** сохраняется положение источника света.

```
MVector lightPointC lightPositionHnd.asDouble3() );
```

Нормаль к плоскости земной поверхности направлена строго вверх, т. е. по направлению оси *y*.

```
MVector planeNormal( 0.0, 1.0, 0.0 );
```

Построим точку, которая гарантированно лежит на поверхности.

```
MVector planePoint( 0.0, gHeight, 0.0);
```

Рассчитаем скалярное произведение нормали и данной точки.

```
double c = planeNormal * planePoint;
```

Деформируем теперь каждую точку геометрической фигуры, определяющей поверхность тени. Деформация является результатом такой проекции точек на плоскость земной поверхности, центром которой является месторасположение источника света.

```
MPoint surfPoint;
double denom, t;
MItGeometry iter( shadowSurfaceHnd, false );
for( ; !iter.isDone(); iter.next() )
{
```

Найдем положение геометрической фигуры в мировом пространстве координат.

```
surfPoint = iter.position( MSpace::kWorld );
```

Проведем воображаемую линию между точкой поверхности и источником света. Новое положение точки поверхности определяется пересечением этой линии с плоскостью земли.

```
denom = planeNormal * (surfPoint - lightPoint);
if( denom != 0.0 )
{
    t = (c - (planeNormal * lightPoint)) / denom;
    surfPoint = lightPoint + t * (surfPoint - lightPoint);
}
```

Сменим положение точки поверхности на результат проекции.

```
iter.setPosition( surfPoint, MSpace::kWorld );
}
```

Затем обновим атрибут, содержащий выходную поверхность.

```
data.setClean( plug );
}
```

В случае запроса Maya на пересчет неизвестного узлу атрибута вернем код надлежащего признака.

```
else
    stat = MS::kUnknownParameter;

return stat;
}
```

Функция **initialize** создает шаблоны атрибутов узла.

```
MStatus GroundShadowNode::initialize()
```

```
{
```

Атрибут **lightPosition** представляет собой обычную точку. Для ее хранения достаточно трех вещественных чисел двойной точности.

```
MFnNumericAttribute nAttr;
lightPosition = nAttr.create( "lightPosition", "lpos",
                             MFnNumericData::k3Double, 0.0 );
nAttr.setKeyable( true );
```

Атрибут **groundHeight** - это лишь значение расстояния. Оно измеряется в единицах длины, поэтому для его хранения используется класс **MFnUnitAttribute**.

```
groundColor = u MFnUnitAttribute uAttr;
Attr.create( "groundColor", "grnd",
            MFnUnitAttribute::kDistance, 0.0 );
uAttr.setKeyable( true );
```

Атрибут **castingSurface** имеет тип **MFnGenericAttribute**. Он позволяет описать ряд разнообразных типов данных, которые будут подаваться на его вход. В данном случае атрибут станет принимать и сетки, и NURBS-поверхности.

```
MFnGenericAttribute gAttr;
castingSurface = gAttr.create( "castingSurface", "csrf" );
gAttr.addAccept( MFnData::kMesh );
gAttr.addAccept( MFnData::kNurbsSurface );
gAttr.setHidden( true );
```

Так как атрибут **shadowSurface** является результатом деформации копии **castingSurface**, он должен поддерживать те же типы данных, что и последняя. Соответственно, этот атрибут имеет тот же тип, что и поверхность **castingSurface**.

```
shadowSurface = gAttr.create( "shadowSurface", "ssrf" );
gAttr.addAccept( MFnData::kMesh );
gAttr.addAccept( MFnData::kNurbsSurface );
gAttr.setHidden( true );
```

Этот атрибут является выходным, поэтому его можно регенерировать на основе входов узла, а значит, он не требует сохранения в файле сцены Maya.

```
gAttr.setStorable( false );
```

Теперь все атрибуты добавляются в узел.

```
addAttribute( groundHeight );
addAttribute( lightPosition );
addAttribute( castingSurface );
addAttributeC shadowSurface );
```

Изменение любого из атрибутов **groundHeight**, **lightPosition** или **castingSurface** влияет на атрибут **shadowSurface**. Далее следует описание этой зависимости.

```
attributeAffects( groundHeight, shadowSurface );
attributeAffects( lightPosition, shadowSurface );
attributeAffects( castingSurface, shadowSurface );
```

```
return MS::kSuccess;
}
```

4.5.4. Атрибуты

В описании любого атрибута узла участвуют классы, порожденные от **MFnAttribute**. В зависимости от того, атрибут какого типа вы хотите создать, используется тот или иной класс-потомок **MFnAttribute**.

К сожалению, одно и то же слово *атрибут* служит в Maya для обозначения двух отчасти различных понятий. Как следствие, возникают две точки зрения. С точки зрения пользователя, каждый узел включает в себя несколько атрибутов. Они изменяются пользователем или участвуют в анимации. На взгляд программиста, атрибуты - это нечто иное. В этом разделе понятие *атрибут* интерпретируется так, как оно понимается именно с позиции программиста.

В интерфейсе C++ API класс **MFnAttribute** и его потомки могут интуитивно трактоваться как шаблоны или проекты, которые предписывают, как следует создавать фрагменты данных, расположенные в узлах. Самое важное отличие от пользовательского взгляда на атрибуты состоит в том, что те фактически не содержат данных. Они лишь предоставляют их детальное описание. Само подобное описание служит для создания реальных данных.

К примеру, класс, производный от **MFnAttribute**, описывает имя, а также тип данных узла. Кроме того, он определяет, должны ли данные сохраняться

на диске и может ли изменяться их значение (чтение/запись). Атрибут включает описание отдельного фрагмента данных. Скажем, он может указывать что в узле будет содержаться элемент типа `float` с именем `amplitude`. При создании узла выделяется область памяти для хранения `float`, при этом ей назначается имя `amplitude`. Каждый узел имеет свой собственный, уникальный элемент данных `amplitude`. Если вы поменяете значение `amplitude` одного узла, это не повлияет на значения `amplitude` в остальных.

Maya считывает информацию, содержащуюся в атрибуте и использует ее для фактического создания данных, присутствующих в каждом узле. Коль скоро атрибут - это шаблон, отсюда следует, что он описывается лишь однажды. По существующему соглашению, атрибуты описываются в статической функции-члене узла с именем `initialize`, хотя вы вправе использовать для этих целей любую другую функцию.

Создание атрибутов

Атрибут создается в теле функции инициализации. Затем он регистрируется узлом, для чего служит функция `addAttribute` класса `MPxNode`. В следующем примере создается и регистрируется узлом атрибут `days`.

```
class MyNode : public MPxNode
{
    ...
    static MStatus initialize();
    static MObject daysAttr;
};

MObject MyNode::daysAttr;

MStatus MyNode::initialize()
{
    MFnNumericAttribute numFn;
    daysAttr = numFn.create( "days", "d", MFnNumericData::kLong );

    addAttribute( daysAttr );

    return MS::kSuccess;
}
```

Тот же базовый формат применяется и для остальных атрибутов. Главное отличие состоит в использовании в зависимости от типа атрибута разных классов **MFn...Attribute**.

Составные атрибуты

Составные атрибуты служат для группировки других атрибутов. Атрибуты, объединенные в группу, считаются *потомками* составного атрибута, который, в свою очередь, рассматривается как *их родитель*. В данном примере составной атрибут player создан на основе двух дочерних атрибутов: age и homeruns.

```
class MyNode : public MPxNode
{
    ...
    static MStatus initialize();
    static MObject playerAttr;
    static HObject ageAttr;
    static HObject homeRunsAttr;
};

MObject MyNode::playerAttr;
MObject MyNode::ageAttr;
MObject MyNode::homeRunsAttr;
```

```
MStatus MyNode::initialize()
{
```

Дочерние атрибуты описываются как обычно.

```
MFnNumericAttribute numFn;
ageAttr = numFn.create( "age", "a", MFnNumericData::kLong );
homeRunsAttr = numFn.create( "homeruns", "hr", MFnNumericData::kLong );
```

Составной атрибут использует класс **MFnCompoundAttribute**. Его создание не отличается от создания других атрибутов.

```
MFnCompoundAttribute compFn;
playerAttr = compFn.create( "player", "ply" );
```

Затем дочерние атрибуты добавляются к составному.

```
compFn.addChild( ageAttr );
compFn.addChild( homeRunsAttr );
```

Наконец, все атрибуты, как дочерние, так и родительский, вводятся в состав узла. Немаловажно заметить, что при включении в узел составного атрибута ни один из его потомков не добавляется автоматически. Дочерние атрибуты нужно добавить явно.

```
addAttribute( ageAttr );
addAttribute( homeRunsAttr );
addAttribute( playerAttr );

return MS::kSuccess;
}
```

Значения по умолчанию

Каждый атрибут имеет значение по умолчанию. Это именно то значение, которое присваивается атрибуту при его инициализации. В зависимости от конкретного класса данных вы можете установить значение по умолчанию, воспользовавшись входящей в функциональный набор функцией `create` или соответствующей функцией установки умолчания.

К примеру, значение по умолчанию атрибутов перечислимого типа (`MFnEnumAttribute`) устанавливается в функции `create` следующим образом. Приведенный код задает значение атрибута по умолчанию, равное 0.

```
MFnEnumAttribute enumFn;
attr = enumFn.create( "days", "d", 0 );
...

```

Класс `MFnNumericAttribute` служит для создания числовых атрибутов. Его значение по умолчанию можно установить при помощи функции `create` или функции-члена этого класса с именем `setDefault`. Следующие строки демонстрируют применение следующего метода:

```
MFnNumericAttribute numFn;
attr = numFn.create( "active", "act", MFnNumericData::kBoolean );
numFn.setDefault( false );
```

Если пользователь ни разу не изменил значение атрибута, то оно совпадает с установкой по умолчанию. Если же значение атрибута не отличается от заданного по умолчанию, оно не будет записываться на диск. Коль скоро такое значение равно умолчанию, в его сохранении нет никакой необходимости. Атрибут можно попросту инициализировать значением по умолчанию, а не значением с диска. Это сокращает размер помещаемых на диск файлов сцены.

Важным следствием этого является то, что атрибуты, пользовавшиеся предыдущими значениями по умолчанию, станут при дальнейшей смене умолчаний атрибутов автоматически пользоваться новыми установками. Такое поведение не всегда может привести к планируемому вами результату. Оно способно «разрушить» существующие сцены, поскольку те атрибуты, которые использовали установки по умолчанию, теперь имеют иные значения. Лучший способ избежать этого— на старте разработки узла решить, каким будет умалчиваемое значение атрибута. Зафиксировав значение атрибута по умолчанию до начала создания сцен Maya с участием этого узла, вы избежите подобной проблемы.

Свойства атрибутов

Атрибуты наделены различными свойствами. Эти свойства, к примеру, определяют, имеете ли вы право изменять значение атрибута или устанавливать с ним соединения. В табл. 4.2 перечислены те общие свойства, которые имеются у всех атрибутов.

В следующем разделе более подробно описываются некоторые наиболее важные свойства.

Readable и Writable

Атрибут имеет флаги *readable* и *writable*. По умолчанию оба этих флага принимают значение *true*. Достаточно интересно, что значение атрибута можно прочитать всегда, независимо от того, установлен ли флаг разрешения считывания. Флаг возможности считывания определяет, может ли атрибут использоваться как источник соединения. Если он имеет значение *false*, вам не удастся создать соединение, берущее начало в этом атрибуте и заканчивающееся в каком-то другом. Для получения и установки флага разрешения считывания применяются, соответственно, функции *isReadable* и *setReadable*.

Вот пример с использованием этих функций.

```
MFnMessageAttribute msgFn;  
attr = msgFn.create( "controller", "Ctrl" );  
bool isread = msgFn.isReadable(); // Возвращает true  
msgFn.setReadable( false );
```

Если флаг разрешения записи атрибута не установлен в *true*, вы не сможете изменить его значение по команде *setAttr*. Кроме того, им нельзя воспользоваться и как целевым атрибутом соединения, а значит, вам не удастся установить соединение, берущее начало в другом атрибуте и заканчивающееся в этом. В следующем примере показаны различные функции чтения и установки флага разрешения записи.

ТАБЛИЦА 4.2. ОБЩИЕ СВОЙСТВА АТРИБУТОВ

Свойство	Описание	Значение по умолчанию
Readable	Может быть источником соединения	true
Writable	Может быть целевым атрибутом соединения	true
Connectable	Может участвовать в соединении	true
Storable	Сохраняется в файле сцены	true
Keyable	Может быть анимированным	false
Hidden	Является скрытым	false
UsedAsColor	Обрабатывать значения как цвета	false
Cached	Значение кэшируется	true
Array	Является массивом	false
IndexMatter	Индекс не должен изменяться	true
ArrayDataBuilder	Для установки значения использовать форматор данных массива	false
Indeterminant	Определяет, может ли атрибут участвовать в вычислениях	false
DisconnectBehavior	Действия после разрыва соединения	kNothing
Internal	Является внутренним атрибутом узла	false
RenderSource	Перекрывает информацию об образцах рендеринга	false

```
MFnMessageAttribute msgFn;
attr = msgFn.create( "controller", "Ctrl" );
bool iswrite = msgFn.isWritable(); // Возвращает true
msgFn.setWritable( false );
```

Connectable

Как упоминалось ранее, флаги разрешения считывания и записи определяют, можете ли вы использовать атрибут, соответственно, в качестве источника и целевого атрибута соединения. Тем не менее в конечном итоге возможность установки любого соединения определяет флаг разрешения соединений. Если флаг, разрешающий считывание, имеет значение `true`, а флаг, разрешающий соединения, - значение `false`, вам не удастся использовать этот атрибут в качестве источника соединения. Для того чтобы это стало возможным, оба флага должны быть установлены в `true`. Флаг разрешения соединений как таковой использу-

стся в сочетании с флагами разрешения считывания и записи и определяет, какие типы соединений являются допустимыми.

Следующий пример служит иллюстрацией того, как считывается и устанавливается флаг разрешения соединений.

```
MFnMatrixAttribute mtxFn;
attr = mtxFn.create( "xform", "xfm", MFnMatrixAttribute::kDouble );
bool iscon = mtxFn.isConnectable(); // Возвращает true
mtx.setConnectable( false );
```

Keyable

Если атрибут допускает кадрирование, вы можете анимировать его через установку ключевых кадров. По умолчанию кадрирование атрибута не допускается. Как следствие, атрибут не виден в окне **Channel Box**. Для считывания и установки флага разрешения кадрирования используются, соответственно, функции `isKeyable` и `setKeyable`.

```
MFnMatrixAttribute mtxFn;
attr = mtxFn.create( "xform", "xfm", MFnMatrixAttribute::kDouble );
bool canKey = mtxFn.isKeyable(); // Возвращает false
mtx.setKeyable( true );
```

Storable

Флаг, разрешающий сохранение, определяет, сохраняются ли значения атрибута на диске. Установив значение флага равным `false`, вы запретите сохранение атрибута. По умолчанию атрибут является сохраняемым.

Некоторые атрибуты не обязательно должны помещаться на диск. К их числу относятся те, значения которых выражаются через другие атрибуты, иначе говоря, **выходы**. Их значения можно определить исходя из иных, входных атрибутов. Коль скоро выходные атрибуты поддаются расчету, их можно заново найти на основе входных. По сути, выходные атрибуты не нужно сохранять на диске ввиду возможности их восстановления. Хотя сохранение таких атрибутов не запрещено, они будут без необходимости занимать дисковое пространство. Поэтому если вы знаете, что атрибут является выходным, т. е. зависимым атрибутом в вызове функции `attributeAffects`, на диске его сохранять не нужно. Это демонстрирует следующий пример.

```
MFnNumericAttribute numFn;
widthAttr = numFn.create( "width", "w", MFnNumericData::kDouble );

sizeAttr = numFn.create( "size", "s", MFnNumericData::kDouble );
```

```
numFn.setStorable( false ); // Не нужно сохранять
```

```
attributeAffects( widthAttr, sizeAttr );
```

Array

По умолчанию в атрибуте хранится единственный элемент данных. Превратив атрибут в массив, можно заставить его запоминать целую серию элементов. Атрибуты-массивы называют также составными элементами. По умолчанию атрибут не является массивом. Для получения и установки данного свойства служат, соответственно, функции isArray и setArray. В следующем примере создается атрибут для хранения массива вещественных чисел двойной точности.

```
MFnNumericAttribute numFn;
```

```
sizesAttr = numFn.create( "sizes", "s", MFnNumericData::kDouble );
```

```
numFn.setArray( true ); // Теперь в нем можно хранить массив double
```

Методы добавления элементов в массив, а также доступа к ним обсуждаются в одном из следующих разделов.

Cached

По умолчанию атрибуты подвергаются кэшированию. Это значит, что копия данных атрибута располагается в блоке данных узла. Если значение атрибута кэшируется, функция compute не станет вызываться всякий раз, когда это значение будет запрашиваться. Кэширование ускоряет считывание значений атрибутов узла. Его недостатком является потребность в большем объеме памяти. Кэширование можно включить или отключить при помощи функции setCache.

```
MFnNumericAttribute numFn;
```

```
sizesAttr = numFn.create( "sizes", "s", MFnNumericData::kDouble );
```

```
numFn.setCached( false ); // Кэширования больше не будет
```

Динамические атрибуты

Функция initialize является местом описания разных атрибутов узла. Каждый экземпляр узла пользуется подобными атрибутами. Они называются **статическими**, поскольку всегда имеются в любом экземпляре узла. Удалить их **оттуда** вам не удастся.

Наряду с ними, можно описывать **динамические атрибуты**. Это атрибуты, добавляемые позднее, по мере необходимости. Динамические атрибуты могут совместно использоваться всеми узлами данного типа или быть уникальными для каждого из узлов.

Чтобы создать динамический атрибут, его **нужно**, прежде **всего**, описать. Описывается он так же, как и статические атрибуты.

```
MFnNumericAttribute numFn;
MObject attr = numFn.create( "size", "s", MFnNumericData::kDouble );
```

Затем, при помощи функции `addAttribute` класса `MFnDependencyNode` атрибут вводится в состав узла. В нашем случае функции `addAttribute` передается значение `MFnDependencyNode::kLocalDynamicAttr`. Оно означает, что атрибут должен быть добавлен к данному конкретному узлу, а не ко всем узлам этого типа.

```
MFnDependencyNode nodeFn( nodeObj );
nodeFn.addAttribute( attr, MFnDependencyNode::kLocalDynamicAttr );
```

Чтобы добавить атрибут ко всем экземплярам узлов данного типа, как существующим, так и будущим, нужно при вызове функции `addAttribute` использовать значение `MFnDependencyNode::kGlobalDynamicAttr`. Важно при этом, чтобы имя атрибута было уникальным. Если узел уже содержит атрибут с таким именем, динамический атрибут добавлен не будет. В дальнейшем динамический атрибут можно позднее удалить, для чего служит функция `removeAttribute`.

```
nodeFn.removeAttribute( attr, MFnDependencyNode::kLocalDynamicAttr );
```

4.5.5. Функция compute

Функция `compute` представляет собой тот фрагмент кода, где на основе входных атрибутов рассчитывается выходная информация. Функция `compute` принимает ссылку на объект `MDataBlock`, который используется для извлечения и установки атрибутов разных узлов. Чрезвычайно важна, чтобы для получения всей информации, необходимой узлу для расчета выходных значений, применялся исключительно `MDataBlock`. В ходе вычислений вы не должны использовать никакие данные, взятые из источников за пределами этого объекта.

Также чрезвычайно важно, чтобы в теле функции `compute` не вызывались команды языка MEL `setAttr` и `getAttr`. Обе команды могут стать косвенной причиной вычислений над графом `DG`. Если во время этих вычислений будет явно или неявно запрошено значение того же самого подключения, это приведет к образованию бесконечного цикла.

Рассмотрим следующий пример. Даже не отличаясь особой ясностью, он на деле демонстрирует опасность применения `getAttr` в тексте функции `compute`. Мы создадим экземпляр класса `MyNode` с именем `myNode1`. Предположим, что узел имеет два атрибута: `scores` - массив результатов игрока, а также `average` - его средний результат. В случае запроса атрибута `average` узел должен вычислить это значение. Выполнение `MGlobals::executeCommand` в вызванной функции `compute` порождает бесконечный цикл.

```
MStatus MyNode::compute( const MPlug& plug, MDataBlock& data )
{
    MStatus stat;

    if( plug == avgAttr )
    {
        MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
        MDataHandle avgHnd = data.outputValue( avgAttr );
        MGlobal::executeCommand( "getAttr myNode1.average" );
        data.setClean( plug );
    }
    else
        stat = MS::kUnknownParameter;

    return stat;
}
```

Этот пример подводит нас к очень важному правилу, являющемуся ключом к предотвращению любых случаев неявных вычислений над графом DG в теле функции compute.

Для получения значений атрибутов пользуйтесь только объектом MDataBlock.

Обратите внимание: несмотря на то что считывать значения подключений при их пересчете не запрещено, в этом часто нет никакого смысла. Вызов функций getValue класса **MPlug** и asDouble класса **MDataHandle** не приведет к возникновению бесконечного цикла.

```
MStatus MyNode::compute( const MPlug& plug, MDataBlock& data )
{
    MStatus stat;

    if( plug == avgAttr )
    {
        MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
        MDataHandle avgHnd = data.outputValue( avgAttr );
        double value;
```

```

    plug.getValue( value ); // Считывание значения
    MDataHandle resInHnd = data.inputValue( resAttr );
    value = resInHnd.asDouble(); // Считывание значения
    data.setClean( plug );
}
else
    stat = MS::kUnknownParameter;
}

return stat;
}

```

Для вычисления текущего значения подключения никогда не следует полагаться на значение, которое имелось в нем ранее. На самом деле, вы не вправе делать никаких предположений относительно текущего значения подключений. Коль скоро они могут вычисляться в разное время, нет и гарантии того, что подключение будет, к примеру, сохранять значение, оставшееся в нем от предыдущего кадра.

4.5.6. Подключения

В контексте интерфейса C++ API атрибут - это лишь шаблон организации данных в рамках узла. Атрибут не содержит никаких данных, а только представляет подробное описание того, как эти данные должны быть созданы. При наличии конкретного экземпляра узла для доступа к его данным фактически служит *подключение*. Оно предусматривает механизм доступа к реальным данным того или иного узла. Чтобы создать подключение, надо указать требуемый узел и его атрибут. Пользуясь этим сочетанием, подключение обращается к реальным данным конкретного узла. Для создания подключений и доступа к ним служит класс **MPlug**.

Следующий пример иллюстрирует создание объекта MPlug, связанного с атрибутом translateX данного узла transform с именем **ballObj**. Функция **findPlug** класса **MFnDependencyNode** предназначена для организации подключения к заданному атрибуту. Имя указанного атрибута может иметь полную или краткую форму.

```

MFnDependencyNode nodeFn( ballObj );
MPlug transxPlg = nodeFn.findPlug( "translateX" );

```

Кроме того, функция `findPlug` перегружена для подачи на ее вход объекта-атрибута, зарегистрированного в принадлежащей узлу функции `initialize`. Однако непосредственная ссылка на объект-атрибут зачастую доступна лишь в файле реализации узла, а потому такой прием используется достаточно редко.

Теперь, когда подключение создано, можно получить данные атрибута, воспользовавшись функцией `getValue` класса `MPlug`.

```
double tx;  
transxPlg.getValue( tx );
```

Функция `getValue` перегружена для возвращения данных многих различных типов. Поэтому важно гарантировать соответствие типов считываемых данных и данных атрибута узла. Так как атрибут `translateX` имеет тип `double`, он и помещается в переменную этого типа. Однако если вы попытаетесь считать данные атрибута, указав неверный тип, результат будет непредсказуемым. Ваши действия могут даже стать причиной сбоя системы Maya. В следующем примере показана попытка извлечения того же атрибута, но как элемента данных `short`.

```
short tx;  
transxPlg.getValue( tx ); // Непредсказуемый результат
```

Это самая распространенная ошибка, связанная с использованием подключений, поэтому убедитесь в том, что тип данных, которые вы пытаетесь прочитать, соответствует типу данных атрибута.

Для установки значения подключения используйте функцию `setValue`. Следующий пример служит иллюстрацией того, как установить значение подключения, равное 2.3. Обратите внимание на явное приведение `типа`, призванное обеспечить вызов надлежащей версии перегруженной функции `setValue`.

```
transxPlg.setValue( double(2.3) );
```

По умолчанию происходит считывание текущего значения атрибута. Однако вы можете прочитать и значение, соответствующее другому моменту времени. Для этого служит класс `MDGContext`. Сначала установите требуемое время вычислений, а затем вызовите функцию `getValue`. В следующем примере переменная `t` типа `MTime` принимает значение времени, равное 1.5 секундам. Далее она применяется для инициализации переменной `ctx` класса `MDGContext`. Переменная `ctx` передается функции `getValue` класса `MPlug`, что позволяет найти значение атрибута в указанное время.

```
MTime t(1.5, MTime::kSeconds);  
MDGContext ctx(t);  
transxPlg.getValue( tx, ctx );
```

Функция `setValue` не может использоваться в отношении другого момента времени. Она устанавливает только текущее значение атрибута. Для смены текущего времени служит функция `viewFrame` класса `MGlobal`. Заметьте, что эта функция часто требует обновления DG, а потому ее использованием не стоит злоупотреблять.

```
MTIME t(1.5, MTIME::kSeconds);  
MGlobal::viewFrame(t);
```

Для создания анимированных значений атрибута обратитесь к команде `keyframe` языка MEL.

Составные подключения

Помимо прочего, класс `MPlug` применяется для перемещения по составным атрибутам. В следующем примере создано подключение к атрибуту `translate` узла `transform`.

```
MFnDependencyNode nodeFn(transformNodeObj);  
MPlug transPlg = nodeFn.findPlug("translate");
```

Теперь переменная `transPlg` ссылается на атрибут `translate`. Это составной атрибут, содержащий три дочерних атрибута: `translateX`, `translateY` и `translateZ`. Для доступа к дочернему атрибуту воспользуйтесь функцией `child` класса `MPlug`.

```
MObject transxAttr = nodeFn.attribute("translateX");  
MPlug transxPlg = transPlg.child(transxAttr);
```

Найти родителя дочернего подключения можно при помощи функции `parent` того же класса `MPlug`.

```
MPlug parent = transxPlg.parent();
```

Если вы не знаете точного количества и имен дочерних атрибутов, можете использовать функции `numChildren` и `child` класса `MPlug`. Функция `child` является перегруженной, что позволяет ей принимать индекс *n*-го потомка. К примеру, для организации цикла по всем потомкам составного атрибута выполните следующие операторы:

```
const unsigned int nChilds = transPlg.numChildren();  
unsigned int i;  
for( i=0; i < nChilds; i++ )  
{  
    MPlug child = transPlg.child(i);  
    ... используйте дочернее подключение  
}
```

Подключения-массивы

Атрибут-массив содержит ряд других атрибутов. Эти атрибуты носят название элементов массива. Подключение, ссылающееся на массив, именуется подключением-массивом; подключение, ссылающееся на элемент, - подключением-элементом.

В следующем примере создается нестандартный узел, содержащий атрибут-массив scores. Данный атрибут включает в себя последовательность целочисленных значений типа long.

```
class MyNode : public MPxNode
{
    ...
    static MStatus initialize();
    static MObject scoresAttr;
};

MObject MyNode::scoresAttr;
```

```
MStatus MyNode::initialize()
```

```
{
```

С целью создания атрибута **scores** используется класс **MFnNumericAttribute**.

```
MFnNumericAttribute numFn;
scoresAttr = numFn.create( "scores", "scrs", MFnNumericData::kLong );
```

Чтобы указать, что атрибут является массивом, вызовем функцию **setArray**.
numFn.setArray(true);

```
addAttribute( scoresAttr );
```

```
return MS::kSuccess;
```

```
}
```

Чтобы увидеть, какие элементы находятся в массиве, опишем следующую служебную функцию **printArray**. Она выводит содержимое массива в окно редактора Script **Editor**.

```

void printArray( MPlug &arrayPlug )
{
    MString txt( "\nArray..."); // "Массив..."

    unsigned int nElems = arrayPlug.numElements();
    unsigned int i=0;
    for( i=0; i < nElems; i++ )
    {
        MPlug elemPlg = arrayPlug.elementByPhysicalIndex(i);

        txt += "\nElement #"; // "Элемент №"
        txt += (int)i;
        txt += ": ";
        double value = 0.0;
        elemPlg.getValue( value );
        txt += value;
        txt += " (";
        txt += (int)elemPlg.logicalIndex();
        txt += ")";
    }

    MGlobal::displayInfo( txt );
}

```

Рассмотрим самые важные фрагменты этой функции. Первый шаг ее работы связан с подсчетом количества элементов массива. Для этого применяется функция `numElements` класса `MPlug`.

```
unsigned int nElems = arrayPlug.numElements();
```

Организуется цикл по элементам.

```
for( i=0; i < nElems; i++ )
{
```

Организуется подключение к каждому элементу, для чего вызывается функция `elementByPhysicalIndex`, входящая в класс `MPlug`. Ее результат – подключение к *i*-му элементу массива.

```
MPlug elemPlg = arrayPlug.elementByPhysicalIndex(i);
```

Значение подключения элемента определяется при помощи функции `getValue` класса **MPlug**.

```
double value = 0.0;  
elemPlg.getValue( value );
```

Наряду с отображением значения элемента, на экран выводится его логический индекс. Объяснение сути логических индексов последует чуть позже. Для нахождения логического индекса подключения вызывается функция `logicalIndex` класса **MPlug**.

```
txt += (int)elemPlg.logicalIndex();
```

Чтобы продемонстрировать добавление элементов в массив, воспользуемся следующими операторами. Они позволяют создать узел **MyNode** и добавлять элементы в массив `scores`.

```
MObject myNodeObj = dgMod.createNode( MyNode::id );  
MFnDependencyNode depFn( myNodeObj );
```

При помощи функции `findPlug` создадим подключение к атрибуту `scores`.

```
MPlug scoresPlg = depFn.findPlug( "scores" );
```

Элементы массивов имеют физические и логические индексы. Физический индекс элемента - это его номер в реально существующем массиве. Физические индексы массива лежат в диапазоне от 0 до `numElements()-1`. При удалении элемента массива физические индексы некоторых элементов могут изменяться. С другой стороны, логический индекс элемента не меняется никогда. Логический индекс — это средство назначения элементу абсолютного, неизменяемого индекса, не зависящего от любых вставок или удалений, происходящих в реальном массиве. Ссылаясь на элемент массива в языке MEL, вы используете его логический индекс. MEL не содержит средств получения физического индекса элемента.

Необходимость физических и логических индексов обусловлена тем, что соединения между атрибутами устанавливаются на основе их логических адресов. Ввиду постоянства этих индексов вы можете не сомневаться, что удаление одного элемента массива не изменит логического индекса другого. В этом состоит их отличие от физических индексов, которые могут меняться.

Следующие примеры помогут сделать еще более понятным различие между обоими вариантами индексирования. Подключение `scorePlg` связывается с элементом, логический индекс которого равен 0. Для этого используется функция `elementByLogicalIndex`.

```
MPlug scorePlg;  
scorePlg = scoresPlg.elementByLogicalIndex(0);
```

Затем этот элемент принимает значение 46.

```
scorePlg.setValue( 46 );
```

Далее содержимое массива выводится на экран.

```
printArray( scoresPlg );
```

Результаты имеют вид:

Array...

```
Element #0: 46 (0)
```

В массив добавлен один элемент. Для создания элементов с заданным индексом предназначена функция `elementByLogicalIndex`. Если элемент с указанным логическим индексом уже есть, выполняется доступ к существующему элементу. Физический и логический индексы первого элемента равны 0.

Теперь обратимся к элементу с логическим индексом 2. Поскольку его еще нет, он будет создан.

```
scorePlg = scoresPlg.elementByLogicalIndex(2);
```

Установим значение элемента равным 12.

```
scorePlg.setValue( 12 );
```

Выведем массив еще раз.

```
printArray( scoresPlg );
```

Результаты имеют вид:

Array...

```
Element ff0: 46 (0)
```

```
Element #1: 12 (2)
```

Физический массив теперь содержит два элемента. Как вы и могли ожидать, их физические индексы равны 0 и 1. Однако логические индексы элементов имеют значения 0 и 2. Коль скоро при обращении ко второму элементу использовался индекс 2, который был передан функции `elementByLogicalIndex`, элемент получил логический индекс, равный двум. Заметьте, что логический индекс 1 отсутствует. Поскольку обращений к нему пока не последовало, такой элемент не был создан. Логические индексы массива **могут** иметь «пропуски», поэтому массив можно считать разреженным. Несмотря на то что логические индексы являются разреженными, физические индексы отличаются **непрерывностью**.

На следующем шаге обратимся к элементу с логическим индексом 1.

```
scorePlg = scoresPlg.elementByLogicalIndex(1);
```

Установим значение элемента равным 93.

```
scorePlg.setValue( 93 );
```

Затем выведем содержимое массива на экран.

```
printArray( scoresPlg );
```

Результаты имеют вид:

Array...

Element #0: 46 (0)

Element #1: 93 (1)

Element #2: 12 (2)

В массив добавлен элемент с физическим индексом 1. Используемый им логический индекс - это индекс, переданный функции `elementByLogicalIndex`. Обратите внимание на то, что элемент с логическим индексом 2 (значение 12) теперь имеет иной физический индекс. Раньше он был равен 1, а теперь - 2. Это еще раз показывает, что физический индекс элемента может изменяться, в то время как логический – никогда.

Чтобы снова подчеркнуть тот факт, что логический индекс не обязательно соответствует физическому, добавим еще один элемент. На этот раз он будет иметь логический индекс, намного превышающий любой другой. Для обращения к этому элементу воспользуемся логическим индексом 25.

```
scorePlg = scoresPlg.elementByLogicalIndex(25);
```

Элемент принимает значение 57.

```
scorePlg.setValue( 57 );
```

Массив выводится на экран.

```
printArray( scoresPlg );
```

Результаты имеют вид:

Array...

Element #0: 46 (0)

Element #1: 93 (1)

Element #2: 12 (2)

Element #3: 57 (25)

Теперь массив состоит из четырех элементов. Физические индексы лежат в диапазоне от 0 до `numElements()-1`. Логические индексы в точности совпадают с выбранными для них значениями. В данный момент в массиве нет элементов с логическими индексами от 3 до 24. Поскольку логические индексы не меняются, их ис-

пользование является самым безопасным способом организации ссылок на элементы массива.

Считывание элементов по их логическим индексам производится при помощи функции `elementByLogicalIndex` класса `MPlug`. Если, как упоминалось ранее, элемент с данным логическим индексом существует, он считывается. Если элемент не существует, он создается. Следующий пример позволяет узнать значение элемента с логическим индексом 1.

```
int value;
scorePlg = scoresPlg.elementByLogicalIndex(1);
scorePlg.getValue(value); // Значение 93
```

Чтобы получить перечень всех логических индексов массива, воспользуйтесь функцией `getExistingArrayAttributeIndices` класса `MPlug`. Она возвращает массив логических индексов всех существующих элементов (которые участвуют в соединениях либо имеют установленные значения).

```
MIntArray logIndices;
scoresPlg.getExistingArrayAttributeIndices( logIndices );
```

Теперь массив `logIndices` содержит значения [0, 1, 2, 25]. Что произойдет, если попытаться прочитать значение несуществующего элемента? Следующий код содержит обращение к значению элемента с логическим индексом 10.

```
MStatus stat;
scorePlg = scoresPlg.elementByLogicalIndex( 10 );
scorePlg.getValue( value );
MGlobal::displayInfo( MString("value: ") + value ); // "значение: "
```

Результат его выполнения будет таков:

```
// value: 0
```

Если элемента, имеющего указанный логический индекс, нет, вы получите значение, принятое по умолчанию. Это значит, что **все логические** индексы являются допустимыми. Результатом обращения к несуществующему элементу станет **умалчиваемое** значение. Если же элемент существует, результатом станет текущее значение этого элемента.

Вам может потребоваться определить родительское подключение-массив заданного подключения-элемента. С этой целью воспользуйтесь функцией `array` класса `MPlug`.

```
MPlug arrayPlg = scorePlg.array();
```

Подключение `arrayPlg` теперь содержит ссылку на атрибут-массив `scores`.

4.5.7. Блоки данных

Блоки данных являются местом хранения реальных данных узла. Данные не содержатся внутри узла, а находятся в одном или нескольких блоках данных. Вам необязательно знать о том, каково их внутреннее устройство и каким образом они связаны между собой. Взамен для считывания и установки данных в узлах применяются классы **MDataBlock**, **MDataHandle** и **MArrayDataHandle**.

Класс **MDataBlock** предусматривает удобный механизм группировки воедино всех данных узла. Для доступа к одному из атрибутов используйте функции класса **MDataBlock** `inputValue` или `outputValue`. Если в процессе вычислений атрибут играет роль входа, пользуйтесь функцией `inputValue`. Если в процессе вычислений он играет роль выхода, пользуйтесь функцией `outputValue`. Обе эти функции возвращают экземпляр класса **MDataHandle**, предназначенный для дальнейшего доступа к данным. Если вы хотите обратиться к атрибуту-массиву, вам, по аналогии, нужно вызывать функции `inputArrayValue` или `outputArrayValue` класса **MDataBlock**. Обе они возвращают объект **MArrayDataHandle**, позволяющий получить доступ к отдельным элементам массива.

Функции `input...Value` возвращают описатель, который можно применять лишь при операциях чтения данных. Такие данные доступны только для чтения. Вы не сможете переписать их, воспользовавшись описателем, который возвращает эти функции. Аналогично, функции `output...Value` возвращают описатель, который может применяться исключительно для записи данных. Эти данные доступны только для записи. Вы не сможете прочитать их, воспользовавшись описателем, который возвращает названные функции. Поскольку атрибуты делятся на входные (только для чтения) и выходные (только для записи), подобные ограничения обычно не влекут за собой никаких проблем.

В следующем примере класс **MyNode** имеет два атрибута: **scores**, массив целых чисел, и **average**, среднее значение массива. Соответственно, функции `compute` необходим доступ как к атрибуту-массиву **scores**, так и к единичному атрибуту **average**.

```
MStatus MyNode::compute( const MPlug& plug, MDataBlock &data )
{
    MStatus stat;

    if( plug == avgAttr )
    {
        MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
```

```

MDataHandle avgHnd = data.outputValue( avgAttr );
const unsigned int nElems = scoresHnd.elementCount();
double sum = 0.0;
unsigned int i;
for( i=0; i < nElems; i++ )
{
    scoresHnd.jumpToElement( i );
    MDataHandle elemHnd = scoresHnd.inputValue();
    sum += elemHnd.toInt();
}
sum /= nElems;
avgHnd.set( sum );
data.setClean( plug );
}
else
    stat = MS::kUnknownParameter;

return stat;
}

```

В тексте функции `compute` атрибут **scores** используется как входная информация для вычисления атрибута **average**. Доступ к данным, хранящимся в **scores**, осуществляется при помощи функции `inputArrayValue` класса **MDataBlock**. Она возвращает описатель **MArrayDataHandle**, который затем может использоваться для обращения к отдельным элементам массива **scores**.

```
MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
```

Затем, путем вызова функции `outputValue` класса **MDataBlock**, создается описатель среднего значения.

```
MDataHandle avgHnd = data.outputValue( avgAttr );
```

Количество элементов в массиве **scores** определяется при помощи функции `elementCount` класса **MArrayDataHandle**,

```
const unsigned int nElems = scoresHnd.elementCount();
```

Далее организуется цикл по отдельным элементам.

```
double sum = 0.0;
unsigned int i;
for( i=0; i < nElems; i++ )
{
```

Для доступа к конкретному элементу применяется функция `jumpToElement`, описанная в классе **MArrayDataHandle**. В качестве аргумента она принимает индекс элемента массива.

```
scoresHnd.jumpToElement( i );
```

Вслед за этим создается описатель отдельного элемента.

```
MDataHandle elemHnd = scoresHnd.inputValue();
```

Значение элемента считывается и прибавляется к текущей сумме.

```
sum += elemHnd.toInt();
}
```

После этого рассчитывается среднее значение, и найденная величина сохраняется. Наконец, подключение помечается как достоверное.

```
sum /= nElems;
avgHnd.set( sum );
data.setClean( plug );
```

Отдельные потомки составного атрибута могут быть получены с использованием функции `child` класса **MDataHandle**. Поскольку атрибут способен содержать произвольные массивы других атрибутов, а каждый из этих атрибутов сам может представлять из себя массив, класс **MArrayDataHandle** может возвращать описатели этих подмассивов, для чего предназначены функции этого класса `inputArrayHandle` и `outputArrayHandle`.

Классы **MDataHandle** и **MArrayDataHandle** являются достаточно компактными, поэтому их экземпляры можно без труда создавать и удалять по мере необходимости. Они лишь обеспечивают удобный интерфейс с данными нижнего уровня, которые располагаются в разных блоках данных.

4.5.8. Рекомендации по проектированию узлов

Рассмотрев вопросы создания нестандартных узлов и их внедрения в граф DG, обсудим теперь некоторые общие принципы проектирования узлов собственной разработки. Важнейшие правила проектирования узлов таковы.

- ◆ Воздерживайтесь от сложных узлов.
- ◆ Узлы никогда не должны располагать сведениями об окружающей среде или контексте.
- ◆ Узлы никогда не должны использовать данные, являющиеся для них внешними.

Простота

В процессе проектирования и реализации узлов **совсем** несложно начать добавлять к ним все новые и новые возможности. В результате узел может неизвестно вырасти за счет тех **функций**, которые включены в его состав. Помня о том, что в действительности узлы - это **строительные блоки**, используемые в более крупных системах, и избегая создания сложных многоцелевых узлов, вы легче добьетесь повторного использования их функциональности. Если один и тот же узел выполняет две несмежные задачи, следует разбить его на два отдельных узла. Это понизит сложность узлов и максимально расширит возможности их вторичного применения.

Другое важное соображение заключается в том, что коль скоро узел - это просто класс **C++**, можно легко допустить ошибку, нагружив его большим набором функциональных возможностей, чем необходимо на самом деле. В системе Maya узлы **DG** решают точно поставленные и очень специфичные задачи. Они принимают входные и выдают выходные данные. Ничем иным узлы заниматься не должны. Помните, что ваш узел - это не обычный класс **C++**, способный выполнять те или иные действия, а особый фрагмент конкретной системы, графа зависимости Maya.

Контекст

Узел никогда не должен знать о том, в каком месте **DG** он находится. Фактически, он не должен знать даже о **том**, что используется графом **DG**. Узел должен располагать информацией лишь о своих входных и выходных **атрибутах**, а также о том, как выдавать их, отвечая на запрос.

К сожалению, узел может начать «осматриваться **вокруг**». Интерфейс **C++ API** предлагает множество функций, которые позволяют отслеживать соединения, берущие начало в данном **узле** и заканчивающиеся в другом месте. Эти функции дают возможность проследить как **восходящие**, так и **нисходящие** соединения. Таким образом, узел способен определить свое место в более крупной сети **DG**. Однако подобные действия крайне опасны.

Если разработанный вами узел будет знать о том, что является, например, частью истории построения объекта он станет делать определенные предположения. Узел может предположить, что он применяется в конкретной конфигурации узлов. Это чрезвычайно опасно, коль скоро пользователь может создать экземпляр вашего узла и использовать его в каком-то ином месте сети графа зависимости. Ваш узел, по всей вероятности, будет выдавать неправильные результаты или даже может дать сбой, поскольку предполагает наличие определенных условий, которые в данном случае выполняться не будут.

Узел никогда не должен проверять, связаны ли его атрибуты с другими узлами. Правильно разработанный узел не должен делать этого ни при каких обстоятельствах. Maya обрабатывает все соединения узлов так, что вам как разработчику узлов не придется об этом заботиться. Атрибуты представлены в C++ API одинаковым образом независимо от того, участвуют они в соединениях или нет. Распространенной ошибкой является создание узлов, которые знают о том, что они соединены друг с другом. Тогда при помощи указателей узлы пытаются общаться между собой или совместно использовать данные. На практике это равнозначно попытке обойти механизмы передачи данных в Maya, что крайне опасно. Если один из узлов будет удален или разорвет соединение, он может стать причиной разыменования другим узлом указателя, ставшего некорректным. Эта операция может привести к аварийному завершению работы Maya.

Узлы никогда не должны выяснять момент времени вычислений над ними. Если вы разработаете узел, который кэширует информацию с учетом момента времени, высока вероятность того, что в некоторых контекстах ваш узел будет работать неправильно. К примеру, атрибуты узла можно рассчитать в любое указанное время. Но на деле вычисления над узлом могут производиться применительно к разным моментам времени. Если вы разработаете свой узел, предполагая, что обращение к узлу происходит лишь в конкретное время, он не будет работать при использовании других временных значений.

Не высказывая никаких предположений о том, когда и как происходит вычисление узла, вы, по сути, гарантируете правильность функционирования своих узлов.

Локальность

Узлы никогда, ни при каких обстоятельствах не должны изучать свои окрестности. Это особенно важно при расчете узлом своих выходных атрибутов. В процессе вычислений узел никогда не должен пользоваться внешними данными. Он обязан видеть лишь собственные входные атрибуты. При наличии таких данных, которые нужны узлу для расчетов и которые не являются его входными атрибутами, их следует добавить как входной атрибут. Вы вправе выполнять вычисления так, как пожелаете сами.

Все данные, которые приходят в узел, должны передаваться строго посредством графа DG. Они не должны поступать из каких-либо внешних источников. Применение внешних данных внутри узла нарушает его способность функционировать вне контекста. В этом случае вычисления над DG, скорее всего, не приведут правильному результату.

Это ключевое правило имеет такое большое значение, что мы заострим на нем внимание еще раз, хотя и этого будет недостаточно.

Узлы никогда не должны видеть ничего, кроме своих собственных атрибутов!

Сети узлов

Если узел должен иметь столь близорукий взгляд на мир, как же спроектировать систему, в которой нужно задействовать множество взаимосвязанных компонентов? Узлы ничего не знают о своем контексте, поэтому создание и поддержка сети узлов должны стать задачей некоей нестандартной команды. Эта команда должна обладать всей информацией о контексте, в котором создаются узлы и в котором они участвуют в соединениях. Таким образом, узлы должны оставаться простыми и нацеленными на решение единственной задачи. Контекст использования узлов должен стать заботой команд, являющихся составной частью системы.

4.6. Локаторы

Локаторы предоставляют пользователям наглядные трехмерные вспомогательные элементы, позволяющие перемещать их и управлять ими. Они выводятся в окне просмотра Maya, но не отображаются в окончательном варианте рендеринга. Локаторы могут служить для создания трехмерных «рукояток», применяемых пользователями для управления другими процессами. К примеру, локаторы формы отпечатков ног могут использоваться для описания следов персонажа. Пользователь может просто передвигать локаторы отпечатка и изменять направление, в котором пошел персонаж.

В поставку Maya входит множество измерительных инструментов, реализованных как локаторы. Инструмент Distance Tool (Расстояние) создает два локатора, а затем показывает расстояние между ними. Связав каждую из конечных точек локатора с отдельным объектом, вы сразу же увидите расстояние между ними даже тогда, когда объекты анимированы. К дополнительным инструментам измерений относятся Parameter Tool (Параметр) и Arc Length Tool (Длина дуги), которые выводят на экран, соответственно, значение параметра точки и расстояние вдоль кривой.

Создание нестандартного локатора представляет собой достаточно простую задачу, коль скоро для добавления своих новых возможностей вам нужно лишь переписать несколько функций-членов. Основные функции-члены, требующие изменений, – это draw, isBounded и boundingBox. Если вы хотите изобразить ло-

катор собственным цветом, нужно, кроме того, реализовать функции `color` и `colorRGB`. В тексте функции `dDraw` вы вправе рисовать локатор так, как вам будет угодно. Для этого доступны почти все функции OpenGL. Наконец, функции работы с ограничивающим прямоугольником `isBounded` и `boundingBox` важны в том случае, если вы хотите, чтобы ваш локатор корректно работал с разнообразными инструментами выделения и изменения масштаба изображения, которые имеются в Maya.

4.6.1. Модуль BasicLocator

Этот пример подключаемого модуля демонстрирует создание элементарного локатора. Локатор может выводить себя на экран несколькими различными способами. Наряду с этим, он дает пользователю возможность растягивать себя в длину и в ширину. Пример элементарного локатора приведен на рис. 4.26.

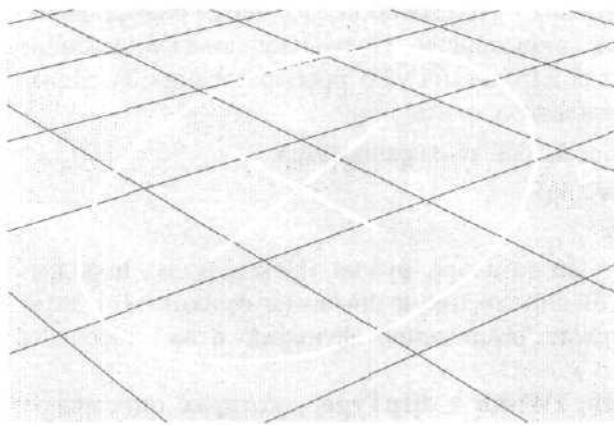


Рис. 4.26. Элементарный локатор

1. Откройте рабочую среду **BasicLocator**.
2. Скомпилируйте ее и загрузите полученный модуль `BasicLocator.mll` в среде Maya.
3. Выберите пункт меню **File | New Scene**.
4. В редакторе **Script Editor** выполните следующую команду:

`createNode basicLocator;`

Узел `basicLocator` будет создан и выведен на экран. Локатор не нуждается в соединении с другими узлами или какой-то иной инициализации, как

это было в предыдущих примерах, поэтому команда `createNode` является быстрым и простым методом создания экземпляра узла.

5. Увеличьте изображение локатора или нажмите клавишу **a**, чтобы вы могли рассмотреть его более отчетливо.

6. Откройте редактор **Attribute Editor**.

На экране вы увидите допускающие кадрирование атрибуты узла **basicLocator1**.

7. Установите значение **X Width** (Ширина по X) равным 2.

8. Установите значение **Disp Type** (Тип изображения) равным 1.

Локатор примет ромбовидную форму.

9. Установите значение **Disp Type** равным 2.

Локатор будет изображен как эллипс, так как атрибут **X Width** вдвое превышает атрибут **Z Width** (Ширина по Z). Придав обоим размерам равное значение, вы получите изображение окружности. Поскольку локатор, подобно всем другим узлам **OAG**, имеет родительский узел преобразования, вы можете перемещать его, вращать и масштабировать.

10. Поэкспериментируйте с локатором, двигая и вращая узел.

Модуль: BasicLocator

Файл: BasicLocator.h

Чтобы создать нестандартный узел локатора, нужно, прежде всего, породить его от класса **MPxLocatorNode**. В дополнение к обычным функциям `creator` и `initialize` класс также содержит реализацию функций `draw`, `isBounded` и `boundingBox`.

Он имеет три атрибута: **xWidth**, **zWidth** и **dispType**, - которые описывают, соответственно, масштаб локатора в длину и в ширину, а также способ его отображения.

```
class BasicLocator : public MPxLocatorNode
{
public:
    virtual void draw( M3dView & view, const MDagPath & path,
                        M3dView::DisplayStyle style,
                        M3dView::DisplayStatus status );

    virtual bool isBounded() const;
    virtual HBoundingBox boundingBox() const;
```

```
static void *creator();
static MStatus initialize();

static const MTypeId typeId;
static const MString typeName;

// Атрибуты
static MObject xWidth;
static MObject zWidth;
static MObject dispType;

private;
    bool getCirclePoints( MPointArray &pts ) const;
};
```

Модуль: BasicLocator**Файл: BasicLocator.cpp**

Узел отображает себя одним из трех способов; как треугольник, ромб или окружность. Каждую из этих форм можно описать последовательностью взаимосвязанных линий. Вместо того чтобы создавать отдельные функции рисования для каждого типа изображения, нами организован массив точек. Если изображение примет вид треугольника, на экран будут выведены только три точки. Когда оно примет вид ромба – четыре точки и т. д. Кроме того, сюда же входит дополнительная точка, совпадающая с первой и тем самым замыкающая кривую. Этот массив точек генерируется функцией `getCirclePoints`. При вычислении окончательного положения точек функция учитывает атрибуты `xWidth`, `zWidth` и `dispType`.

```
const double M_2PI = M_PI * 2.0;

bool BasicLocator::getCirclePoints( MPointArray &pts ) const
{
    MStatus stat;
    MObject thisNode = thisMObject();
    MFnDagNode dagFn( thisNode );
```

```
MPPlug xWidthPlug = dagFn.findPlug( xWidth, &stat );
float xWidthValue;
xWidthPlug.getValue( xWidthValue );

MPPlug zWidthPlug = dagFn.findPlug( zWidth, &stat );
float zWidthValue;
zWidthPlug.getValue( zWidthValue );

MPPlug typePlug = dagFn.findPlug( dispType, &stat );
short typeValue;
typePlug.getValue( typeValue );

unsigned int nCirclePts;

switch( typeValue )
{
    case 0:
        nCirclePts = 4;
        break;
    case 1:
        nCirclePts = 5;
        break;
    case 2:
        nCirclePts = 20;
        break;
}

pts.clear();
pts.setSizeIncrement( nCirclePts );

MPoint pt;
pt.y = 0.0;
```

```
const double angleIncr = M_2PI / (nCirclePts - 1);
double angle = 0.0;
unsigned int i=0;
for( ; i < nCirclePts; i++, angle+=angleIncr )
{
    pt.x = xWidthValue * cos( angle );
    pt.z = zWidthValue * sin( angle );
    pts.append( pt );
}

return true;
}
```

Нередко самой сложной частью локатора является его функция `draw`. В Maya эта функция служит двум целям. Она вызывается для рисования узла в текущем окне просмотра и применяется для обнаружения выделения узла. Когда Maya выясняет, является ли объект с областью выделения, либо определяет, где был произведен щелчок мышью, вызывается принадлежащая объекту функция `draw`. Далее, Maya использует результат рисования для определения признака выделения объекта. Вы не должны беспокоиться о втором варианте применения функции `draw`, поскольку Maya обрабатывает его для вас автоматически.

По вашему желанию локатор может быть сколь простым, столь и сложным. Фактически, в процессе рисования вам доступны почти все функциональные вызовы OpenGL. Есть, однако и некоторые ограничения. Важно, чтобы функция `draw` оставила OpenGL в том же состоянии, как и до своего вызова. Решение этой задачи может облегчить функция OpenGL `glPushAttrib`. Она служит для сохранения и восстановления текущего состояния графической подсистемы.

На момент вызова принадлежащей локатору функции `draw` текущие преобразования уже в силе. Узел как таковой не должен заниматься позиционированием, вращением или масштабированием самого себя. Все это делается автоматически. Узел просто рисует себя в локальном пространстве объекта. Текущий цвет также устанавливается автоматически исходя из текущего состояния узла (выделен, активен, бездействует и т. д.). Составив функции `color` и `colorRGB`, вы можете описать собственный цвет для любого из этих состояний узла.

Функция `draw` вызывается с несколькими аргументами. Первый из них - текущее окно просмотра Maya, в котором будет изображен локатор. Оно задается классом `M3dView`. Также функции передается полный путь `MDagPath`,

ведущий к узлу-локатору по ОАГ. Аргумент `M3dView::DisplayStyle` описывает режим рисования, в котором локатор должен быть выведен. Среди стилей - ограничивающий прямоугольник, затененное изображение, плоское затенение и т. д. Аргумент `M3dView::DisplayStatus` определяет текущее состояние узла в окне просмотра. Локатор может быть активным, «живым», бездейственным, невидимым и т. д.

```
void BasicLocator::draw( M3dView &view, const MDagPath &path,
                        M3dView::DisplayStyle style,
                        M3dView::DisplayStatus status )
{
```

Прежде чем вызывать любую из функций рисования OpenGL, нужно вызвать функцию `beginGL` и сохранить текущее состояние OpenGL в памяти.

```
view.beginGL();
glPushAttrib( GL_CURRENT_BIT );
```

Следом создаются вершины текущей отображаемой формы.

```
MPointArray pts;
getCirclePoints( pts );
```

Далее в виде отрезков прямых изображается последовательность вершин локатора.

```
glBegin(GL_LINE_STRIP);
for( unsigned int i=0; i < pts.length(); i++ )
    glVertex3f( float(pts[i].x), float(pts[i].y), float(pts[i].z) );
glEnd();
```

Кроме того, в центре локатора изображается небольшой крестик.

```
glBegin(GL_LINES);
    glVertex3f( -0.5f, 0.0f, 0.0f );
    glVertex3f( 0.5f, 0.0f, 0.0f );

    glVertex3f( 0.0f, 0.0f, -0.5f );
    glVertex3f( 0.0f, 0.0f, 0.5f );
glEnd();
```

Теперь, когда рисование завершено, текущее состояние OpenGL может быть заменено восстановленным из памяти предыдущим состоянием. Также следует вызвать функцию `endGL`, для того чтобы показать, что рисование окончено.

```
glPopAttrib();
view.endGL();
}
```

Функция `isBounded` вызывается в том случае, когда Maya требуется определить, знает ли узел о размерах своих границ. Вызов функции `boundingBox` позволяет получить реальные границы формы локатора. Эти функции работы с границами строго рекомендуется реализовать. Без них Maya будет испытывать трудности в определении точного размера локатора, поэтому операции **Frame All** (Показать все) и **Frame Selection** (Показать выделенные объекты) приведут к некорректному масштабированию.

```
bool BasicLocator::isBounded() const
{
    return true;
}
```

Вершины локатора являются, как известно, самыми протяженными в длину и в ширину фрагментами узла, поэтому найти ограничивающий прямоугольник для них - значит найти ограничивающий прямоугольник для узла в целом. Этот процесс существенно упрощается функцией `getCirclePoints`, которая предоставляет полный список вершин локатора.

```
MBoundingBox BasicLocator::boundingBox() const
{
    MPointArray pts;
    getCirclePoints( pts );
}
```

Конструктор по умолчанию объекта `bbox` инициализирует его как прямоугольник нулевого размера.

```
MBoundingBox bbox;
```

С помощью функции `expand` класса `MBoundingBox` в него добавляются точки. Ограничивающий прямоугольник увеличивается, если это необходимо для вставки очередной из них.

```
for( unsigned int i=0; i < pts.length(); i++ )
    bbox.expand( pts[i] );
return bbox;
}
```

Функция `initialize` создает все три дополнительных атрибута и добавляет их в узел.

```
MStatus BasicLocator::initialize()
```

```
{  
    MFnUnitAttribute unitFn;  
    MFnNumericAttribute numFn;  
    MStatus stat;
```

Как атрибут **xWidth**, так и атрибут **zWidth** определяют значение расстояния и потому должны быть организованы при помощи класса **MFnUnitAttribute**, а не простого типа **double**. Кроме того, необходимо указать их минимальное значение, а также значение по умолчанию.

```
xWidth = unitFn.create( "xWidth", "xw", MFnUnitAttribute::kDistance );  
unitFn.setDefault( MDistance(1.0, MDistance::uiUnit()) );  
unitFn.setMin( MDistance(0.0, MDistance::uiUnit()) );  
unitFn.setKeyable( true );  
stat = addAttribute( xWidth );  
if(!stat)  
{  
    stat.perror( "Unable to add \"xWidth\" attribute" );  
    // "Невозможно добавить атрибут xWidth"  
    return stat;  
}
```

```
zWidth = unitFn.create( "zWidth", "zw", MFnUnitAttribute::kDistance );  
unitFn.setDefault( MDistance(1.0, MDistance::uiUnit()) );  
unitFn.setMin( MDistance(0.0, MDistance::uiUnit()) );  
unitFn.setKeyable( true );  
stat = addAttribute( zWidth );  
if(!stat)  
{  
    stat.perror( "Unable to add \"zWidth\" attribute" );  
    return stat;  
}
```

Атрибут **dispType** содержит значение **short**, которое указывает текущий стиль рисования. Его минимальное значение равно 0, а максимальное - 2.

```
dispType = numFn.create( "dispType", "att", MFnNumericData::kShort );
numFn.setDefaultValue( 0 );
numFn.setMin( 0 );
numFn.setMax( 2 );
numFn.setKeyable( true );
stat = addAttribute( dispType );
if(!stat)
{
    stat.error( "Unable to add \"dispType\" attribute" );
    return stat;
}

return MS::kSuccess;
}
```

Модуль: BasicLocator

Файл: PluginMain.cpp

Узел локатора похож на любой другой узел, поэтому его нужно зарегистрировать во время инициализации подключаемого модуля. Регистрация узла выполняется как обычно, за исключением того, что функции registerNode нужно передать тип узла `MPxNode::kLocatorNode`. Если не указать его явно, по умолчанию будет использован тип `MPxNode::kDependNode`.

```
stat = plugin.registerNode(
    BasicLocator::typeName,
    BasicLocator:: typeId,
    &BasicLocator::creator,
    &BasicLocator::initialize,
    MPxNode::kLocatorNode );
```

Функция deregisterNode вызывается как обычно – с указанием идентификатора узла.

```
stat = plugin.deregisterNode( BasicLocator::typeId );
```

47. Манипуляторы

Maya предлагает немало различных способов настройки и изменения атрибутов заданного узла. Наиболее широко используется **такое** средство установки значения **атрибута**, как редактор Attribute Editor. Но для некоторых узлов есть и более наглядные инструменты решения этой задачи. Благодаря использованию манипуляторов пользователи могут просто модифицировать конкретное множество атрибутов. В зависимости от типа атрибута такой подход может быть интуитивно более понятным, нежели использование Attribute Editor или иных методов.

4.7.1. Модуль BasicLocator2

Чтобы на деле увидеть простой пример работы манипуляторов, скомпилируйте и загрузите подключаемый модуль BasicLocator2. Данный модуль расширяет возможности узла-локатора, созданного в предыдущем [примере](#), и содержит манипуляторы для всех своих собственных атрибутов: **xWidth**, **zWidth** и **dispType**. Эти манипуляторы показаны на рис. 4.27.

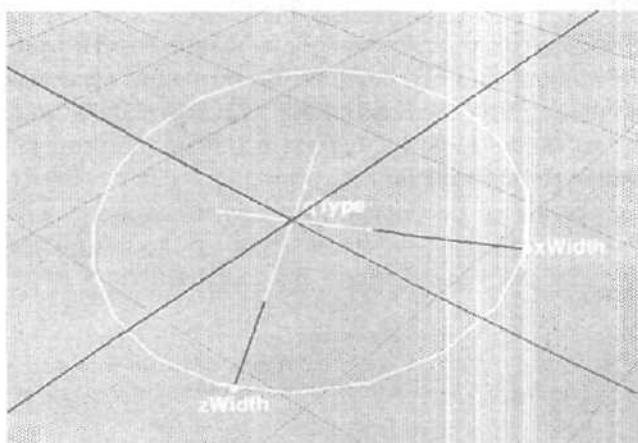


Рис. 4.27. Элементарный локатор с выведенными на экран манипуляторами

1. Откройте рабочую среду **BasicLocator2**.
2. Скомпилируйте ее и загрузите полученный модуль **BasicLocator2.mll** в среде Maya.
3. Выберите пункт меню **File | New Scene**.

4. В редакторе **Script Editor** выполните следующую команду:

```
createNode basicLocator;
```

Узел **basicLocator** будет создан и выведен на экран.

5. Выберите на панели окна просмотра пункт **View | Frame Selection** (Вид | Показать выделенные объекты) или нажмите клавишу **f**.

6. Откройте редактор **Attribute Editor**.

Редактор откроется, и вы увидите, как изменяются значения атрибутов.

7. Выберите пункт меню **Modify | Transformation Tools | Show Manipulator Tool** (Изменить Инструменты преобразования Показать манипуляторы) или нажмите клавишу **t**.

Манипуляторы, которыми обладает локатор, будут показаны на экране.

Если манипуляторы не видны, убедитесь в том, что вы выбрали узел **basicLocator1**, а не его родительский узел преобразования **transform1**.

8. Щелкните по значку-окружности рядом со словом **Type** (Тип).

Атрибут локатора **dispType** будет увеличен на единицу.

9. Щелкните по этому значку еще раз.

Теперь локатор изображается в виде окружности.

10. Перетащите точку рядом с меткой **X Width**.

Атрибут локатора **xWidth** обновится. Обратите внимание, что, когда вы управляете манипулятором, значение атрибута в Attribute Editor изменяется.

Узел-манипулятор можно описать и для любого другого узла **Dependency Graph**. В данном примере нами был описан узел-локатор **basicLocator** и соответствующий узел-манипулятор **basicLocatorManip**. Узел-манипулятор – это особый узел, создаваемый, когда пользователь выделяет узел и, выбрав режим **Show Manipulator Tool**, запрашивает инструмент управления им. В этот момент Maya порождает экземпляр узла-манипулятора. Затем этот узел создает ряд базовых манипуляторов, с которыми взаимодействует пользователь. Узел-манипулятор прикрепляет эти базовые манипуляторы к редактируемому узлу, поэтому изменение одного из манипуляторов, к примеру, буксировка точки, приводит к смене значения атрибута узла. Когда пользователь покидает режим **Show Manipulator Tool**, узел-манипулятор автоматически удаляется. Узел-манипулятор не может отображаться ни в одном из окон Maya (**Hypergraph**, **Outliner** и т. д.) и не позволяет редактировать собственные атрибуты. В сущности, он представляет собой временный рабочий узел, который существует лишь в период выделения узла, редактируемого с помощью манипулятора.

Создать узел-манипулятор вне инструмента **Show Manipulator Tool** или пользовательского контекста невозможно.

Узел-манипулятор можно считать «контейнером» базовых манипуляторов. Базовый манипулятор - это один из методов изменения атрибутов, предварительно описанных в среде Maya. В данном примере манипулятор **baseLocatorManip** содержит три базовых манипулятора: манипуляторы расстояния для атрибутов **xWidth** и **zWidth** и манипулятор состояния для атрибута **dispType**. В настоящее время Maya поддерживает **десять** разных базовых манипуляторов.

1. FreePointManip
2. DirectionManip
3. DistanceManip
4. PointOnCurveManip
5. PointOnSurfaceManip
6. DiscManip
7. CircleSweepManip
8. ToggleManip
9. StateManip
10. CurveSegmentManip

Полное описание каждого базового манипулятора приведено в справочной документации **Maya**, посвященной классам. Описывать свои собственные базовые манипуляторы запрещено, поэтому создавайте более сложные манипуляторы на основе серии базовых. Такие базовые манипуляторы будут именоваться **потомками** главного узла-манипулятора.

В числе основных этапов создания **узла-манипулятора** – порождение нового класса от **MPxManipContainer**. Далее в узел добавляется ряд дочерних базовых манипуляторов. Эти потомки связываются с конкретным атрибутом целевого узла, которым будет оперировать манипулятор. Можно описать как простую связь между дочерним манипулятором и связанным с ним атрибутом, так и более сложную. Для описания подобной связи предназначены функции преобразования. Они описывают механизм передачи информации от манипулятора к атрибуту и обратно.

Модуль: BasicLocator2

Файл: BasicLocator.cpp

Прежде чем переходить к рассмотрению деталей организации узла-манипулятора, нужно слегка изменить первоначальный узел-локатор, чтобы сообщить Maya о том, что теперь с этим локатором связан узел-манипулятор. Maya поддерживает внутреннюю таблицу всех узлов и узлов-манипуляторов, которые с ними связаны. В конце функции `initialize` вызовите статическую функцию `addToManipConnectTable` с идентификатором `того` узла, который теперь располагает манипулятором.

```
MStatus BasicLocator::initialize()
{
    ...
    MPxManipContainer::addToManipConnectTable( const_cast<MTypeId &>
                                                ( typeId ) );
    ...
}
```

Это единственное изменение, которое необходимо внести в узел `baseLocator` для работы с манипулятором.

Модуль: **BasicLocator2**

Файл: **BasicLocatorManip.h**

Узел-манипулятор является производным от класса `MPxManipContainer`.

```
class BasicLocatorManip ; public MPxManipContainer
{
public:
    virtual MStatus createChildren();
    virtual MStatus connectToDependNode(const MObject & node);

    virtual void draw( M3dView & view, const MDagPath & path,
                      M3dView::DisplayStyle style,
                      M3dView::DisplayStatus status);

    static void * creator();

    MManipData startPointCallback(unsigned index) const;
    MManipData sideDirectionCallback(unsigned index) const;
    MManipData backDirectionCallback(unsigned index) const;
```

```
MVector nodeTranslation() const;
MVector worldOffset(MVector vect) const;

static const MTypeId typeId;
static const MString typeName;

MManipData centerPointCallback(unsigned index) const;

// Пути к дочерним манипуляторам
MDagPath xWidthDagPath;
MDagPath zWidthDagPath;
MDagPath typeDagPath;

// Объект, с которым будет работать манипулятор
MObject targetObj;
};
```

Наряду с описанием множества функций-членов, речь о которых пойдет чуть позднее, узел содержит описание **путей** по ОАГ к используемым базовым манипуляторам. Это дочерние манипуляторы, с которыми взаимодействует **пользователь**.

```
MDagPath xWidthDagPath;
MDagPath zWidthDagPath;
MDagPath typeDagPath;
```

Кроме того, узлу-манипулятору нужно помнить о том, каким узлом он управляет. Это тот узел **baseLocator**, который был выделен в **момент** активации **Show Manipulator Tool**. Ссылка на управляемый объект хранится в элементе данных **targetObj**. Заметьте, что он имеет тип **MObject**, а не **MDagPath**.

```
MObject targetObj;
```

Манипуляторы могут применяться по отношению к любому узлу **зависимости**, а не только к узлам **ОАГ**. Поэтому вы можете **пользоваться** манипуляторами при работе с любым узлом, который вами описан.

Модуль: BasicLocator2

Файл: BasicLocatorManip.cpp

Как и все другие узлы, узел **BasicLocatorManip** должен содержать описание уникального идентификатора. В этом примере мы воспользуемся идентификатором, имеющим временное значение.

```
const MTypeId BasicLocalManip:: typeId( 0x00338 );
```

Очень важно назвать узел-манипулятор с учетом имени того узла, с которым он будет работать. Имя узла-манипулятора должно быть образовано из названия узла и следующей за ним строки **Manip**, а значит, данный узел должен называться **baseLocatorManip**. Очень важно, чтобы имя узла точно соответствовало этому соглашению, иначе манипулятор не будет связан с ним корректным образом. Обратите внимание, что имена узлов чувствительны к регистру.

```
const MString BasicLocatorManip:: typeName( "basicLocatorManip" );
```

Интересно отметить тот факт, что **MPxManipContainer** имеет свою статическую функцию **initialize**. В предыдущих примерах описание собственной функции имел класс. Поскольку этот манипулятор не выполняет никаких специальных действий по инициализации помимо того, что было сделано базовым классом, узел-манипулятор не требует перегрузки этой функции. Заметьте, однако что если вы и в самом деле опишете свою собственную функцию **initialize**, вы должны гарантировать вызов функции **initialize** базового класса. Следующий код служит примером того, как это можно сделать.

```
MStatus BasicLocatorManip:: initialize()
{
MStatus stat;
stat = MPxManipContainer:: initialize();
... //здесь выполните любую дополнительную инициализацию
return stat;
}
```

Поскольку узел-манипулятор в действительности является лишь контейнером для хранения базовых манипуляторов, функция **createChildren** очень важна. Она предназначена для создания базовых манипуляторов и включения их в состав узла.

```
MStatus BasicLocatorManip:: createChildren()
{
MStatus stat = MStatus:: kSuccess;
```

Чтобы организовать новый узел-манипулятор расстояния, вызовите функцию `addDistanceManip`. Для хранения пути к вновь созданному узлу служит путь по ОАГ `xWidthDagPath`.

```
xWidthDagPath = addDistanceManip( "xWidthManip", "xW" );
```

К узлу-манипулятору расстояния присоедините набор функций `MFnDistanceMap`, что позволит вам устанавливать различные свойства манипулятора.

```
MFnDistanceManip xWidthFn( xWidthDagPath );
```

Манипулятор расстояния вычерчивается в определенном направлении в виде начальной и конечной точек. Пользователь передвигает конечную точку и тем самым задает расстояние. Начальная точка манипулятора `xWidth` должна располагаться в центре узла, а конечная - пробегать вдоль оси `x`.

```
xWidthFn.setStartPoint( MVector(0.0, 0.0, 0.0) );
```

```
xWidthFn.setDirection( MVector(1.0, 0.0, 0.0) );
```

Создайте еще один узел-манипулятор расстояния, на сей раз для атрибута `zWidth`. Его начальная точка также располагается в центре узла. Однако конечная точка этого манипулятора пробегает по оси `z`.

```
zWidthDagPath = addDistanceManip( "zWidthManip", "zW" );
```

```
MFnDistanceManip zWidthFn( zWidthDagPath );
```

```
zWidthFn.setStartPoint( MVector(0.0, 0.0, 0.0) );
```

```
zWidthFn.setDirection( MVector(0.0, 0.0, 1.0) );
```

Для взаимодействия с атрибутом `dispType` создайте манипулятор состояния. При его инициализации укажите, что он имеет максимум три состояния, т. е. столько же состояний, что и атрибут `dispType`.

```
typeDagPath = addStateManip( "typeManip", "tM" );
```

```
MFnStateManip typeFn( typeDagPath );
```

```
typeFn.setMaxStates( 3 );
```

Теперь все базовые манипуляторы созданы. Заметьте, что связи между конкретным базовым манипулятором и атрибутом `узла`, на который он будет влиять, не установлены. С этой целью используется функция `connectToDependNode`. Она вызывается при выделении узла сцены и запросе его манипулятора обычно посредством Show Manipulator Tool и принимает на вход данный узел. Она связывает базовые манипуляторы с подключениями узла. Кроме того, ею реализуются все вызовы, необходимые для размещения и отображения **отдельных** манипуляторов.

```
MStatus BasicLocatorManip::connectToDependNode( const MObject &node )
```

```
{
```

Элемент данных `targetObj` принимает значение управляемого узла.

```
targetObj = node;
MFnDependencyNode nodeFn(node);
```

К манипулятору расстояния присоединяется функциональный набор `MFnDistanceManip`. Создается подключение к атрибуту узла `xWidth`.

```
MFnDistanceManip xWidthFn( xWidthDagPath );
MPlug xWidthPlug = nodeFn.findPlug( "xWidth", &stat );
```

Организуется связь между манипулятором расстояния и подключением к атрибуту `xWidth`. Любые изменения, происходящие с манипулятором расстояния, теперь будут отражаться на значении атрибута `xWidth` и наоборот.

```
xWidthFn.connectToDistancePlug( xWidthPlug );
```

При рисовании манипулятора он выводится на экран относительно мирового начала координат. Если узел перемещается, нужно сообщить об этом манипулятору, с тем чтобы тот смог перерисовать себя относительно нового положения узла. Этот процесс уведомления осуществляется посредством функции обратного вызова. Подобная функция вызывается всякий раз, когда происходит перемещение узла. Она может точно определить местоположение его центра. Центр узла - это то место, где выводится начальная точка манипулятора расстояния.

Чтобы осуществить этот процесс, требуется обратный вызов, реализующий преобразование «подключение - манипулятор». Этот обратный вызов конвертирует значение подключения в значение связанного с ним манипулятора.

```
addPlugToManipConversionCallback( xWidthFn.startPointIndex(),
                                  (plugToManipConversionCallback)centerPointCallback );
```

Аналогично, направление манипулятора расстояния должно следовать за направлением оси `x` узла. Организуется другой обратный вызов, который реализует преобразование «подключение - манипулятор» и определяет точное направление оси `x` узла.

```
addPlugToManipConversionCallback( xWidthFn.directionIndex(),
                                  (plugToManipConversionCallback)sideDirectionCallback
);
```

Та же последовательность шагов выполняется и в отношении подключения `zWidth`. Начальная точка его манипулятора расстояния ставится в центр узла, а его направление приравнивается к направлению оси `z` последнего.

```
MFnDistanceManip zWidthFn( zWidthDagPath );
MPlug zWidthPlug = nodeFn.findPlug( "zWidth" );
zWidthFn.connectToDistancePlug( zWidthPlug );

addPlugToManipConversionCallback( zWidthFn.startPointIndex(),
                                 (plugToManipConversionCallback)centerPointCallback );

addPlugToManipConversionCallback( zWidthFn.directionIndex(),
                                 (plugToManipConversionCallback)backDirectionCallback
);

Манипулятор состояния связывается с принадлежащим узлу подключением
dispType. Положение манипулятора состояния определяется центром узла.

MFnStateManip typeFn( typeDagPath );
MPlug typePlug = nodeFn.findPlug( "dispType" );
typeFn.connectToStatePlug( typePlug );

addPlugToManipConversionCallback( typeFn.positionIndex(),
                                 (plugToManipConversionCallback)centerPointCallback );
```

Крайне важно, чтобы в конце этой функции вызывалась функция `finishAddingMaps`. Ее можно вызвать только один раз.

```
finishAddingManips();
```

Вслед за вызовом функции `finishAddingMaps` должен произойти вызов функции `connectToDependNode` класса `MPxManipContainer`. Это позволит базовому классу добавлять любые связи с манипуляторами, которые будут необходимы.

```
MPxManipContainer::connectToDependNode(node);
```

```
return MS::kSuccess;
```

```
}
```

Если вы не намерены дописывать собственные элементы рисования, тогда вам не нужно добавлять нестандартную функцию `draw`. Без нее базовые манипуляторы будут по-прежнему рисовать `себя` самостоятельно. Если же вы хотите добавить к базовым манипуляторам какие-то дополнительные элементы изображения или любую другую информацию, `предназначенную` для вывода на экран, создайте свою собственную функцию `d raw`. В этом примере рядом с базо-

выми манипуляторами будут выведены текстовые метки, позволяющие пользователю легче распознавать эти манипуляторы.

Как и функция `draw` локатора, функция `draw` узла-манипулятора принимает на вход объект `M3dView`, который содержит текущее изображение. Параметр `MDagPath` представляет собой путь к отображаемому узлу-манипулятору. Аргументы `M3dView::DisplayStyle` и `M3dView::DisplayStatus` описывают внешний вид и режим, в котором следует нарисовать узел.

```
void BasicLocatorManip::draw( M3dView &view, const MDagPath &path,
                               M3dView::DisplayStyle style,
                               M3dView::DisplayStatus status )
```

Прежде чем приступить к рисованию, нужно вызвать функцию `draw` класса-предка. Она нарисует все, что нужно этому классу, включая, в большинстве случаев, базовые манипуляторы.

```
MPxManipContainer::draw(view, path, style, status);
```

Далее считываются значения атрибутов узлов.

```
MFnDependencyNode nodeFn( targetObj );
MPlug xWidthPlug = nodeFn.findPlug( "xWidth" );
float xWidth;
xWidthPlug.getValue( xWidth );
```

```
MPlug zWidthPlug = nodeFn.findPlug( "zWidth" );
float zWidth;
zWidthPlug.getValue( zWidth );
```

Производится подготовка к рисованию средствами OpenGL.

```
view.beginGL();
glPushAttrib( GL_CURRENT_BIT );
```

Создается текст для маркировки манипулятора.

```
char str[100];
MVector TextVector;
MString distanceText;
```

Задайте текст для маркировки манипулятора.

```
strcpy(str, "XWidth");
distanceText = str;
```

Следующий шаг состоит в определении того места, в котором метка должна быть выведена на экран. Это будет вектор в мировых координатах, который указывает, откуда нужно выводить метку, если узел не был перемещен. Далее от центра узла делается отступ на величину этого вектора.

```
MVector xWidthTrans = nodeTranslation();
TextVector = xWidthTrans;
TextVector += worldOffsetC MVector(xWidth , 0, 0) );
view.drawText(distanceText, TextVector, M3dView::kLeft);
```

В найденной позиции метка выводится на экран.

Текст метки и позиция вывода определяются для атрибутов zWidth и dispType. Их метки тоже отображаются на экране.

```
strcpy(str, "ZWidth");
distanceText = str;
MVector zWidthTrans = nodeTranslation();
TextVector = zWidthTrans;
TextVector += worldOffsetC MVector( 0, 0, zWidth ) );
view.drawText(distanceText, TextVector, M3dView::kLeft);

strcpy(str, "Type");
distanceText = str;
TextVector = nodeTranslation();
TextVector += worldOffsetC MVector( 0, 0.1, 0 ) );
view.drawText( distanceText, TextVector, M3dView::kLeft);
```

Рисование в OpenGL заканчивается.

```
glPopAttrib();
view.endGL();
}
```

Остальные функции класса - это функции, которые отвечают за определение положения различных точек, расположенных вокруг узла. Это функции обратного вызова, реализующие преобразования «подключение - манипулятор». Функция centerPointCallback возвращает текущую позицию центра узла в мировых координатах.

```
MManipData BasicLocatorManip::centerPointCallback(unsigned index) const
{
```

Для сохранения этой позиции нужен объект, работающий с числовыми данными. Позиция сохраняется в виде трех вещественных чисел двойной точности - `k3Double`.

```
MFnNumericData numData;
MObject numDataObj = numData.create( MFnNumericData::k3Double );
```

Функция `nodeTranslation` возвращает смещение центра узла относительно мирового начала координат.

```
MVector vec = nodeTranslation();
```

Позиция устанавливается равной этому смещению.

```
numData.setData( vec.x, vec.y, vec.z );
```

Каждая функция обратного вызова возвращает объект типа `MManipData`. Этот объект предназначен для хранения всего множества разнообразных типов данных, которые манипулятор способен потенциально изменить.

```
return MManipData( numDataObj );
}
```

Функция `sideDirectionCallback` возвращает позицию вектора (1, 0, 0) в мировом пространстве, рассчитанную относительно узла.

```
MManipData BasicLocatorManip::sideDirectionCallback( unsigned index )
{
    MFnNumericData numData;
    MObject numDataObj = numData.create(MFnNumericData::k3Double);
    MVector vec = worldOffset( MVector(1, 0, 0) );
    numData.setData( vec.x, vec.y, vec.z );
    return MManipData( numDataObj );
}
```

Функция `nodeTranslation` является служебной функцией, которая возвращает текущее положение центра узла в мировых координатах.

```
MVector BasicLocatorManip::nodeTranslation() const
{
    MFnDagNode dagFn( targetObj );
    MDagPath path;
    dagFn.getPath(path);
```

```
path.pop();

MFnTransform transformFn( path );
return transformFn.translation( MSpace::kWorld );
}
```

Функция `worldOffset` является служебной функцией, которая возвращает вектор, равный `смещению` между указанным вектором и его позицией в мировом пространстве.

```
MVector BasicLocatorManip::worldOffset(MVector vect) const
```

```
{
```

```
    MVector axis;
```

```
    MFnDagNode transform( targetObj );
```

```
    MDagPath path;
```

```
    transform.getPath(path);
```

```
    MVector pos( path.inclusiveMatrix() * MVector(0, 0, 0) );
```

```
    axis = vect * path.inclusiveMatrix();
```

```
    axis = axis - pos;
```

```
    return axis;
```

```
}
```

4.8. Деформаторы

Деформаторы не только просты в концептуальном понимании, их также легко реализовать. Деформатором называется узел, который принимает на вход серию точек и переставляет их на новое место. Деформатор не может добавить или удалить точку, он вправе лишь изменить ее положение в пространстве. Для этого может использоваться любой метод передвижения точек. Такой метод может быть простым, как, например, сдвиг на заданное расстояние, или сложным, как определение нового положения точек с привлечением имитационного моделирования из гидродинамики. Деформатор может `оперировать` широким спектром геометрических примитивов. На элементарном уровне он способен модифицировать точки решеток, управляющие вершины и вершины многоугольников.

4.8.1. Модуль SwirlDeformer

Рассмотрим подробно пример деформатора **SwirlDeformer**. Подключаемый модуль создает узел-деформатор скручивания, который изменяет форму объекта, вращая его вершины с учетом их удаленности от центра воронки. Также вы сможете задавать начальную и конечную границу **вращения**, между которыми происходит скручивание. На рис. 4.28 показано, что произойдет с объектом после работы над ним деформатора **SwirlDeformer**.

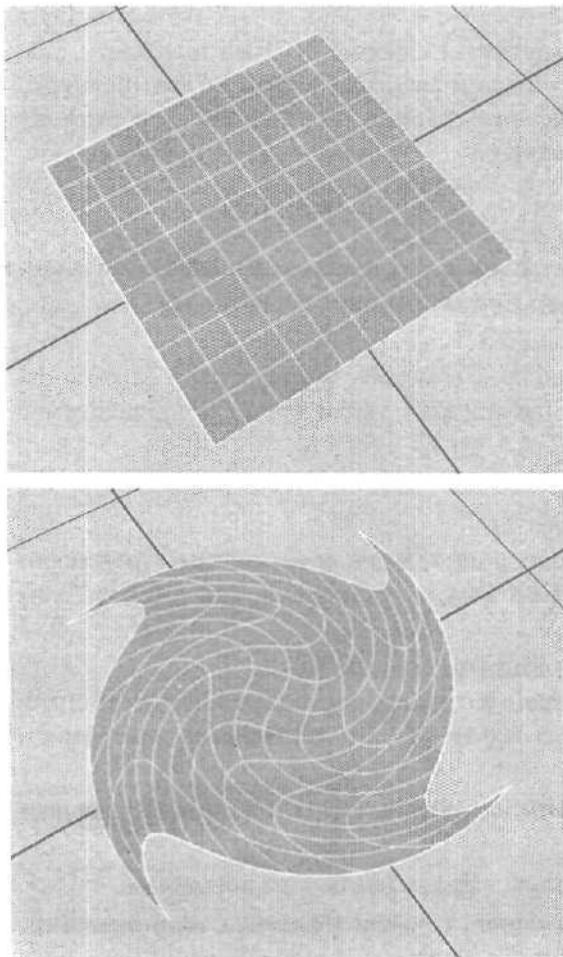


Рис. 4.28. SwirlDeformer

1. Откройте рабочую среду **SwirlDeformer**.
2. Скомпилируйте ее и загрузите полученный модуль **SwirlDeformer.mll** в среде Maya.
3. Откройте сцену **Swirl.ma**.
4. Выделите объект **nurbsPlane1**.
5. В редакторе **Script Editor** выполните следующую команду:
`deformer -type swirl;`

Под действием деформатора скручивания форма NURBS-плоскости станет иной. Обратите внимание: скручивание не деформирует объект целиком. Это происходит потому, что конечная граница, переданная деформатору скручивания, меньше ширины объекта,
6. Отобразите окно **Channel Box**.
7. Щелкните по элементу **swirl1** в разделе **INPUTS**.

На экране будут показаны все три основных параметра: **Envelope** (Огибающая), **Start Dist** (Начальная граница), **End Dist** (Конечная граница).
8. Установите атрибут **Envelope** равным 0.5.

Вращение уже выражено не так сильно. Вы можете в диалоговом режиме изменять значение **Envelope** и наблюдать увеличение и уменьшение интенсивности скручивания.
9. Верните атрибут **Envelope** в 1.0.
10. Установите атрибут **End Dist** равным 2.0.

Теперь скручивание затрагивает меньшую область поверхности. Поэкспериментируйте, изменения **Start Dist** и **End Dist**, и посмотрите, как эти атрибуты влияют на характер скручивания.
11. Установите **Start Dist равным 0.0** и **End Dist равным 3.0**.

Кроме общих параметров огибающей и границ, вы можете задать степень влияния деформатора на отдельные вершины. Это достигается изменением весовых коэффициентов.
12. Нажмите клавишу F8, чтобы перейти на уровень **Control Vertex** (Управляющие вершины) NURBS-плоскости.
13. Выделите несколько вершин, лежащих недалеко от центра плоскости.
14. Выберите пункт главного меню **Window | General Editors Component Editor...** (Окно 1 Универсальные редакторы Редактор компонентов...).
15. Щелкните по вкладке **Weighted Deformers** (Взвешенные деформаторы).

16. Перейдите к столбцу **swirl1**, чтобы выделить несколько вершин для редактирования.
17. Переместите бегунок значения в нижней части окна.

Веса выделенных вершин изменятся, в результате чего влияние деформатора скручивания уменьшится или увеличится в зависимости от установленных значений.

Создавая собственный деформатор, вы автоматически получаете от Maya некоторые атрибуты. Атрибут **envelope** - общий для всех деформаторов. Он описывает степень деформации, прикладываемой к объекту.

Модуль: SwirlDeformer

Файл: SwirlDeformer.h

Класс **SwirlDeformer** создается на основе класса **MPxDeformerNode**.

```
class SwirlDeformer : public MPxDeformerNode
{
public:
```

```
    static void *creator();
    static MStatus initializeC();
```

Главное отличие деформатора от прочих узлов заключается в наличии в его составе функции **deform** и отсутствии в нем функции **compute**. Функция **compute** по-прежнему существует, однако она реализована в порождающем классе. Если вам не требуется выполнять никаких специальных вычислений, лучше всего просто реализовать функцию **deform**, оставив функцию **compute** в сфере ответственности базового класса. Подробности, касающиеся функции **deform**, описаны в следующем разделе.

```
virtual MStatus deform( MDataBlock &block,
                        MItGeometry &iter,
                        const MMatrix &mat,
                        unsigned int multiIndex );
```

Опишем атрибуты начальной и конечной границы скручивания. Атрибут **envelope** наследуется от класса **MPxDeformerNode**.

```
private:
    // Атрибуты
    static MObject startDist;
    static MObject endDist;
```

```
};
```

Модуль: SwirlDeformer**Файл: SwirlDeformer.cpp**

Основная функция, требующая реализации в деформаторе, - это функция `deform`. Именно функция `deform` на деле выполняет деформацию геометрического объекта. Ей передается `MDataBlock`, который содержит блок данных узла деформации. Это тот самый блок данных, который обычно передается вам в функции `compute`. Второй параметр `MIteGeometry` - итератор, позволяющий обойти все точки геометрического объекта. Итератор может организовать цикл по точкам многих различных типов, включая **управляющие** вершины, точки **решеток**, вершины сеток и т. д. Он уже настроен на обход точек лишь того типа, которые должны быть модифицированы деформатором.

Третий параметр `localToWorld` типа `MMatrix` - матрица преобразования локального пространства в мировое. Точки при передаче их деформатору находятся в локальном пространстве узла геометрии. При необходимости деформации в мировом пространстве просто преобразуйте их при помощи этой матрицы. После деформации, однако важно вернуть эти точки в локальное пространство, воспользовавшись матрицей, обратной по отношению к `localToWorld`.

Последний параметр – это `geomIndex`. Деформатор способен изменять форму множества геометрических узлов, а также множества компонентов единственного узла геометрии. Посредством `geomIndex` Maya отслеживает деформируемые вами фрагменты геометрии.

```
MStatus SwirlDeformer::deform( MDataBlock& block,
                                MIteGeometry &iter,
                                const MMatrix &localToWorld,
                                unsigned int geomIndex )
{
    MStatus stat;
```

Из памяти считывается атрибут **envelope**.

```
MDataHandle envData = block.inputValue( envelope );
float env = envData.asFloat();
```

Если значение `envelope` равно 0, деформация **никак** не повлияет на геометрию, поэтому функция может сразу же заканчивать свою работу.

```
if( env == 0.0 )
    return MS::kSuccess;
```

Далее считаются атрибуты начальной и конечной границы скручивания.

```
MDataHandle startDistHnd = block.inputValue( startDist );
double startDist = startDistHnd.toDouble();
```

```
MDataHandle endDistHnd = block.inputValue( endDist );
double endDist = endDistHnd.toDouble();
```

Для обхода и деформации всех точек деформатор будет использовать итератор геометрии типа **MIteGeometry**.

```
for( iter.reset(); !iter.isDone(); iter.next() )  
{
```

Каждая точка может иметь свой собственный, связанный с ней весовой коэффициент. Этот вес определяется при помощи функции `weightValue` класса **MPxDeformerNode**.

```
weight = weightValue( block, geomIndex, iter.index() );
```

Если точка не имеет веса, то деформатор на нее не повлияет и эту точку можно пропустить.

```
if( weight == 0.0f )
    continue;
```

Производится считывание текущей точки.

```
pt = iter.position();
```

Рассчитывается длина перпендикуляра, опущенного из этой точки на ось *y*.

```
dist = sqrt( pt.x * pt.x + pt.z * pt.z );
```

Если точка лежит ближе к центру вращения, чем начальная граница, или дальше от центра вращения, чем конечная граница, деформатор не окажет на эту точку никакого влияния.

```
if( dist < startDist || dist > endDist )
    continue;
```

Чем ближе к центру деформатора лежит точка, тем сильнее она вращается. Результат вычисления силы этого вращения заносится в переменную `distFactor`. Ее значение лежит в диапазоне от 0 до 1.

```
distFactor = 1 - ((dist - startDist) / (endDist - startDist));
```

Затем определяется угол вращения точки. Он является результатом масштабирования `distFactor` с учетом значения `envelope` и веса `weight` конкретной точки. Полученное произведение умножается на **числовое** значение полного оборота, выраженное в радианах.

```
ang = distFactor * M_PI * 2.0 * env * weight;
```

Если вращение отсутствует, точка будет пропущена.

```
if( ang == 0.0 )
```

```
    continue;
```

Выполняется поворот точки вокруг оси `y` на угол `ang`.

```
cosAng = cos( ang );
```

```
sinAng = sin( ang );
```

```
x = pt.x * cosAng - pt.z * sinAng;
```

```
pt.z = pt.x * sinAng + pt.z * cosAng;
```

```
pt.x = x;
```

Наконец, точка обновляется и занимает новое положение после деформации.

```
iter.setPosition( pt );
```

```
}
```

```
return stat;
```

```
}
```

В дополнение к тем атрибутам, которые унаследованы от базового класса, узел-деформатор имеет и два новых: **`startDist`** и **`endDist`**. Функция `initialize` описывает эти атрибуты и вводит их в состав узла.

```
MStatus SwirlDeformer::initialize()
```

```
{
```

```
MFnUnitAttribute unitFn;
```

```
startDist = unitFn.create( "startDist", "sd", MFnUnitAttribute::kDistance );

```

```
unitFn.setDefault( MDistance( 0.0, MDistance::uiUnit() ) );

```

```
unitFn.setMin( MDistance( 0.0, MDistance::uiUnit() ) );

```

```
unitFn.setKeyable( true );
```

```
endDist = unitFn.create( "endDist", "ed", MFnUnitAttribute::kDistance );

```

```
unitFn.setDefault( MDistance( 3.0, MDistance::uiUnit() ) );

```

```
unitFn.setMin( MDistance( 0.0, MDistance::uiUnit() ) );
unitFn.setKeyable( true );

addAttribute( startDist );
addAttribute( endDist );

attributeAffects( startDist, outputGeom );
attributeAffects( endDist, outputGeom );

return MS::kSuccess;
|
```

Модуль: SwirlDeformer

Файл: PluginMain.cpp

Единственным несущественным изменением функции `initializePlugin` является то, что при регистрации узла-деформатора должен быть задан тип `MPxNode::kDeformerNode`.

```
stat = plugin.registerNode( SwirlDeformer::typeName,
                            SwirlDeformer:: typeId,
                            SwirlDeformer:: creator,
                            SwirlDeformer:: initialize,
                            MPxNode::kDeformerNode );
```

Процесс отмены регистрации в функции `uninitializePlugin` ничем не отличается от аналогичного процесса для других узлов.

4.8.2. Изменения в графе зависимости

Знать принципы применения узла-деформатора в графе зависимости Maya также важно, как и понимать принципы его написания. При отладке своего деформатора вам важно уяснить его место в грандиозной схеме общей деформации Maya.

Команда языка MEL `deformer` вносит ряд изменений в граф Dependency Graph. До деформации NURBS-плоскости DG выглядит так, как показано на рис. 4.29. Это стандартный вид истории построения NURBS-плоскости.

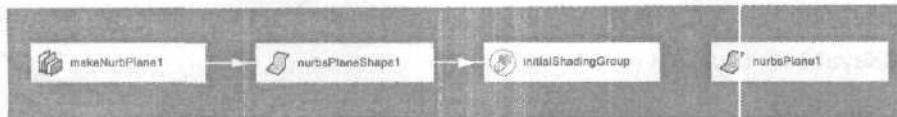


Рис. 4.29. NURBS-плоскость до деформации

После выполнения оператора `deformer -type swirl` плоскость NURBS станет выглядеть так, как на рис. 4.30. Ради простоты некоторые фрагменты были изъяты, однако основные узлы и их соединения сохранены.

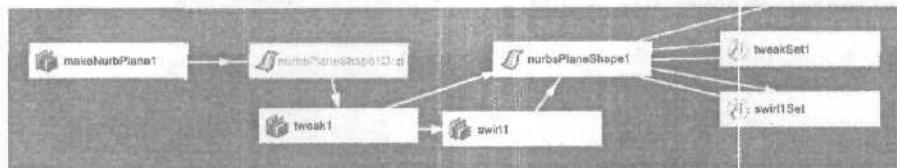


Рис. 4.30. NURBS-плоскость после деформации

Узел `makeNurbPlane1` теперь является входом узла `nurbsPlaneShapeOrig`. Этот узел – точная копия NURBS-формы `nurbsPlaneShape1` до деформации. Далее эта форма передается узлу доводки с именем `tweak1`. В ходе первой деформации объекта Maya создает его копию и связывает ее с новым узлом доводки. Это позволяет вам возвращаться назад и заниматься доводкой объекта. Любые доводки исходного объекта вступают в силу после всех деформаций. Выход узла доводки – это потенциально деформированный объект. Поскольку никакая доводка исходной формы NURBS-плоскости не производилась, узел доводки без изменений передает геометрический объект от узла формы к узлу скручивания `swirl1`. Узел `swirl1` является экземпляром узла-деформатора `SwirlDeformer`. Он деформирует переданную ему геометрию. Полученный в результате деформированый геометрический объект затем поступает на вход последней NURBS-формы `nurbsPlaneShape1`. Эта последняя форма содержит окончательный вариант геометрического объекта и выводит его на экран.

Всякий раз при создании узла-деформатора порождается новый узел категории `set`. Узел `set` содержит список тех объектов, а также, возможно, их компонентов, на которые должно распространяться действие деформатора. В данном случае для узла `tweak1` и узла деформации `swirl1` созданы узлы наборов `tweakSet1` и `swirl1Set`, соответственно.

По запросу пользователя Maya автоматически обрабатывает переупорядочение ваших узлов-деформаторов. Кроме того, при удалении узла-деформатора Maya автоматически удаляет все посторонние узлы и заново выстраивает цепочку

деформаций. Также Maya гарантирует отсутствие неоправданного дублирования геометрических объектов при их передаче от одного узла-деформатора к другому. С первого до последнего деформатора, по сути, последовательно передается одна копия геометрического объекта. Каждый деформатор вносит свой вклад в изменение формы объекта. В конечном итоге геометрия объекта предоставляет собой результат поочередного воздействия всех деформаторов на исходный объект.

4.8.3. Вспомогательные инструменты

Деформаторы многих типов зачастую полезно снабжать одним или несколькими вспомогательными инструментами. Вспомогательный инструмент – это узел, создаваемый деформатором либо для того, чтобы дать пользователю возможность лучше представить себе атрибуты деформатора, либо для того, чтобы позволить ему напрямую ими управлять. К примеру, деформатор **twist** включает в себя узел **twistHandle**. Он дает пользователю возможность наглядно представить различные параметры изгиба, включая начальный и конечный угол, а также верхнюю и нижнюю границу деформации.

4.8.4. Модуль SwirlDeformer2

Этот подключаемый модуль продемонстрирует принципы написания вспомогательного инструмента, позволяющего лучше управлять деформатором скручивания. В частности, мы создадим локатор для описания центра и направления вращения. Обычно применяемый таким образом вспомогательный локатор об разно называют «рукойatkой». Локатор как таковой станем именовать **swirlHandle**. На рис. 4.31 показан результат работы **SwirlDeformer** над NURBS-плоскостью. Последовавшее затем вращение **swirlHandle** вызвало скручивание вдоль направления поворота.

1. Откройте рабочую среду **SwirlDeformer2**.
2. Скомпилируйте ее и загрузите полученный модуль **SwirlDeformer2.mll** в среде Maya.
3. Убедитесь в том, что модуль **SwirlDeformer.mll** выгружен из памяти.
4. Откройте сцену **Swirl.ma**.
5. Выделите объект **nurbsPlane1**.

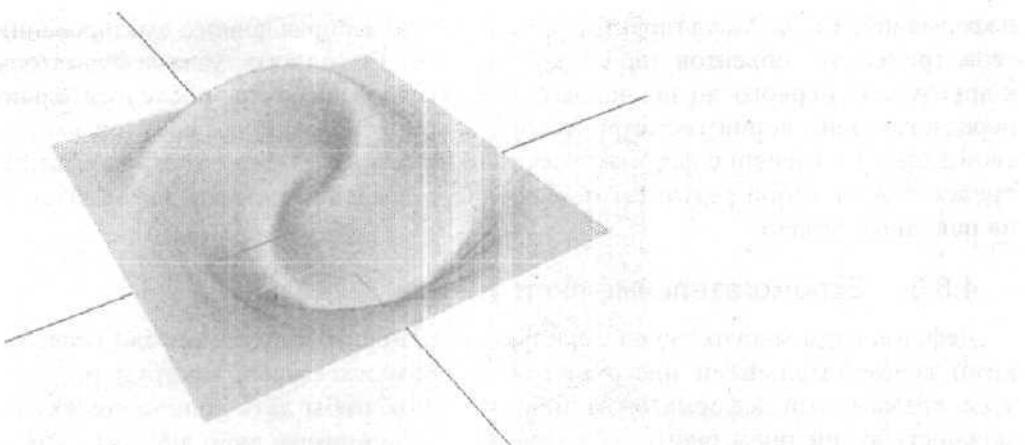


Рис. 4.31. SwirlDeformer с повернутой «рукояткой»

6. В редакторе **Script Editor** выполните следующую команду:


```
deformer -type swirl;
```

 Деформатор скручивания по-прежнему воздействует на плоскость.
7. Выберите **объект swirlHandle**.
8. В окне **Channel Box** установите значение атрибута **Rotate X** узла **swirlHandle** равным 25.
Выполняемое теперь скручивание характеризуется некоторым углом относительно «рукоятки». Поэкспериментируйте, изменяя атрибут **End Dist** узла **swirl1**, и внимательно **посмотрите**, как повлияет угол на степень скручивания.
Добавить один или несколько вспомогательных инструментов к имеющемуся узлу-деформатору можно сравнительно легко. Заметьте, что в этом примере использовался один из предопределенных локаторов Maya. Можно создать собственный, нестандартный локатор и использовать его в качестве «рукоятки». Также вы вправе добавлять к узлу-деформатору свои **собственные** манипуляторы. Все эти возможности демонстрируют многие стандартные деформаторы Maya.

Модуль: SwirlDeformer2

Файл: SwirlDeformer.h

Класс **SwirlDeformer2** описывается совершенно аналогично классу **SwirlDeformer**. Чтобы ввести в состав узла вспомогательные инструменты, первоначальный класс **SwirlDeformer** был **модифицирован** – в нем описан ряд но-

вых функций и новый атрибут. А именно, реализованы следующие функции, унаследованные от класса **MPxDeformerNode**. Речь о них пойдет чуть позднее.

```
virtual MObject &accessoryAttribute() const;  
virtual MStatus accessoryNodeSetup( MDagModifier &cmd );
```

Добавлен дополнительный матричный атрибут **deformSpace**, содержащий текущую матрицу преобразования объекта-«рукоятки».

```
static MObject deformSpace;
```

Модуль: SwirlDeformer2

Файл: SwirlDeformer.cpp

В функции `initialize` к узлу добавлен новый матричный атрибут **deformSpace**.

```
MStatus SwirlDeformer::initialize()  
{  
    MFnMatrixAttribute mAttr;  
    deformSpace = mAttr.create( "deformSpace", "dSp" );
```

Коль скоро эта матрица является матрицей преобразования «рукоятки», сохранять ее нет никакой необходимости.

```
mAttr.setStorable( false );
```

Изменение функции `deform` нацелено на добавление в нее нового матричного атрибута **deformSpace**.

```
MStatus SwirlDeformer::deform( MDataBlock& block,  
                               MITGeometry &iter,  
                               const MMatrix &localToWorld,  
                               unsigned int geomIndex )  
{
```

Этот атрибут считывается из блока данных так же, как и другие атрибуты. Здесь же рассчитывается и обратная матрица.

```
    MDataHandle matData = block.inputValue( deformSpace );  
    MMatrix mat = matData.asMatrix();  
    MMatrix invMat = mat.inverse();
```

Единственное существенное изменение в методе деформации состоит в том, что сначала точки переносятся в локальное пространство вспомогательной «рукоятки». Затем в этом пространстве выполняется деформация, после чего производится обратный переход к исходному пространству объекта. Перенос точки в пространство «рукоятки» реализуется как преобразование этой точки при помощи обратной матрицы «рукоятки».

```
    ...
pt = iter.position();
pt *= invMat;
```

По завершении деформации точки снова переносятся в исходное локальное пространство путем их преобразования с использованием матрицы «рукоятки».

```
    ...
pt *= mat;
iter.setPosition( pt );
```

Создавая узел-деформатор, вызовем функцию `accessoryNodeSetup`, чтобы создать те вспомогательные узлы, которые могут понадобиться деформатору. Этой функции передается объект `MDagModifier`, что позволяет создать новые узлы и добавить их в граф зависимости.

```
MStatus SwirlDeformer::accessoryNodeSetup( MDagModifier &dagMod )
{
    ...
}
```

Построим новый узел-локатор.

```
MObject locObj = dagMod.createNode( "locator", MObject::kNullObj, &stat );
if( !stat )
    return stat;
```

Присвоим локатору более понятное обозначение.

```
dagMod.renameNode( locObj, "swirlHandle" );
```

Принадлежащий узлу-деформатору матричный атрибут `deformSpace` управляет матрицей преобразования узла-локатора, для чего атрибут `matrix` узла преобразования локатора соединен с атрибутом `deformSpace` узла-деформатора. Всякий раз при трансформации локатора атрибут `deformSpace` будет автоматически обновляться.

```
MFnDependencyNode locFn( locObj );
MObject attrMat = locFn.attribute( "matrix" );
stat = dagMod.connect( locObj, attrMat, thisMObject(), deformSpace );

return stat;
}
```

Если бы пользователю пришлось удалять **объект-«рукоятку»** деформатора, следовало бы удалить и сам узел-деформатор, и наоборот. К счастью, Maya управляет этим автоматически, при условии что вы сообщили ей, на какие атрибуты узла-деформатора влияет вспомогательный инструмент. Нужно указать только один из затрагиваемых атрибутов. Когда **объект-«рукоятка»** деформатора удаляется, соединение с этим атрибутом разрывается, после чего Maya удаляет сам узел-деформатор. Аналогично, при удалении **узла-деформатора** **объект-«рукоятка»** деформатора удаляется тоже.

```
MObject &SwirlDeformer::accessoryAttribute() const
{
    return deformSpace;
}
```

4.9. Расширенные возможности C++ API

В этом разделе рассматриваются вопросы, связанные с расширенными возможностями интерфейса C++ API.

4.9.1. Общие вопросы

Обращение к узлам

Так как объект **MObject**, по сути, является лишь «пустым» указателем на некие внутренние данные Maya, очень важно не допускать, чтобы он удерживался слишком долго. **Фактически, MObject** остается действительным лишь тогда, когда еще существует фрагмент данных, на который он ссылается. Если по той или иной причине эти данные будут удалены, **MObject** не получит уведомления об этом и потому продолжит работу с указателем, который стал некорректным. Если в этом случае вы воспользуетесь объектом **MObject**, он, скорее всего, вызовет аварийное завершение Maya по причине разыменования недействительного указателя. Если же вам нужно сохранять ссылки на узлы **OAG**, обратитесь к **объекту MDagPath**. Работая с обычными узлами DG, пользуйтесь их именами. Имя узла

может измениться, поэтому важно поддерживать ссылку по имени в **актуальном состоянии**. Можно установить объект **MNodeMessage**, который будет информировать вас об изменении имени данного узла. Этой цели служит функция **addNameChangedCallback** класса **MNodeMessage**.

Заместители

При описании своих собственных узлов и команд вам может показаться, что это те самые узлы и команды, которые фактически используются графом DG. В действительности они являются лишь заместителями **Объекты-заместители** описываются всеми классами Maya, начинаяющимися с **MPx**.

Скажем, к примеру, вы описали **собственный** класс **MyNode**, производный от **MPxNode**. Создавая экземпляр **MyNode**, Maya **фактически** создаст два объекта. **Один** из **них** – это внутренний объект **Maya**, содержащий ваш экземпляр **MyNode**. Ваш узел не используется в DG напрямую. Там находится именно внутренний объект **графа**, который просто включает в себя ссылку на объект **MyNode**. **Вот** почему все классы, производные от **MPx**, называются заместителями. Реальным узлом является внутренний объект Maya.

На деле вы можете получить указатель на свой узел из узла графа DG, вызвав функцию **userNode** класса **MPxNode**. Она возвратит указатель на экземпляр вашего класса, который используется внутренним узлом Maya.

Так как описанные пользователем узлы образованы из **двух** частей, вы должны быть осторожны в отношении действий, выполняемых вами в конструкторе. Составляя конструктор класса **MyNode**, вы не вправе вызывать никакие функции-члены **MPxNode**. Это относится ко всем классам, которые явно или неявно порождены от **MPxNode**. **Причина** такого ограничения состоит в том, что внутренний объект-узел Maya и экземпляр класса **MyNode** не **связываются** до тех пор, пока экземпляр не сконструирован полностью. Поэтому во **время** построения **MyNode** функции **MPxNode** еще недоступны. Они **станут доступными** лишь после **того**, как **MyNode** будет **создан**, поскольку только тогда установится соединение **между** объектами.

Для упрощения соединения нестандартных узлов в **MPxNode** описана виртуальная функция **postConstructor**, которую вы можете реализовать в своем узле. Вызов этой функции происходит при создании соединения между двумя объектами, а значит, тогда вы сможете свободно вызвать любую из функций-членов **MPxNode**. Как таковой **конструктор MyNode** надлежит делать очень **коротким**, а большую часть работы по инициализации возложить на функцию **postConstructor**. Если конструктору **MyNode** не потребуются никакие функции **MPxNode**, класс, разумеется, может и не иметь реализации **postConstructor**, а выполнять всю инициализацию в своем конструкторе.

Сетевые и несетевые подключения

Одной из проблем, часто сбивающих с толку пользователей, являются сетевые и несетевые подключения Maya. На практике знать различие между ними совершенно не обязательно. Фактически их реализация различается в том, что касается хранения системой Maya информации о соединениях между атрибутами. По сути, разработчикам эта тема зачастую и не нужна. Подключения могут использоваться и без того, знают ли программисты о том, являются они сетевыми или нет. Однако даже притом что понимание этого вопроса может не повлиять на вашу деятельность как разработчика, вы можете получить более глубокое представление о внутренних механизмах Maya.

Подключение, ссылающееся на атрибут конкретного узла, должно знать только об узле и об атрибуте. Обладая этими сведениями, подключение способно найти данные конкретного атрибута узла. При работе с атрибутами-массивами нужна дополнительная информация - индекс элемента. Наличие индекса элемента позволяет подключению отыскать данные, хранящиеся в указанном элементе массива узла. Maya допускает произвольную вложенность атрибутов, а потому подключение фактически содержит полный путь к данному атрибуту. В него входят индексы всех подключений-массивов, начиная с корневого подключения и заканчивая конкретным атрибутом, на который указывает ссылка. Чтобы узнать полный путь к атрибуту данного подключения, воспользуйтесь функцией `info` класса `MPlug`.

Наряду с этим, подключение в Maya можно использовать и для другой цели. А именно, оно может применяться для хранения информации о соединениях между атрибутами. Кроме того, в нем можно хранить и другую статусную информацию. Для упрощения процесса соединения атрибутов и сохранения прочей статусной информации Maya поддерживает внутреннее дерево подключений каждого узла. В это дерево входят подключения, созданные для всех атрибутов узла, участвующих в соединениях. DG использует внутреннее дерево подключений для обхода соединений между атрибутами узлов. Подключение, присутствующее в этом внутреннем дереве, именуется сетевым. Подключение, которого нет в дереве подключений, именуется несетевым. Отсюда логически вытекает, что все атрибуты, участвующие в соединениях, имеют сетевые подключения, которые с ними связаны.

При попытке создать подключение к атрибуту Maya сначала проверяет, есть ли во внутреннем дереве подключений уже имеющееся подключение к данному атрибуту. Если есть, Maya возвращает существующее сетевое подключение. Если же подключения нет, создается новое несетевое подключение. В том и дру-

том случае вы можете использовать полученное в результате подключение совершенно одинаковым образом. Вы вправе вызывать одни и те же функции-члены **MPlug** независимо от того, с сетевым или несетевым подключением вы работаете.

Если вам нужно узнать, является ли подключение **сетевым**, можно воспользоваться функцией **isNetworked** класса **MPlug**. В следующем примере создается подключение к атрибуту **translate** узла **transform**. Функция **isNetworked** класса **MPlug** ~~возвращает~~ **true**, если подключение является **сетевым**, и **false**, если это не так.

```
MFnDependencyNode nodeFn( transformObj );
MPlug transPlg = nodeFn.findPlug( "translate" );
bool isNet = transPlg.isNetworked();
```

4.9.2. Граф зависимости

Контексты

Контекст указывает причину запуска вычислений над графом DG. Контекст вычислений может принимать множество различных состояний. Можно установить «нормальный» («normal») контекст как признак вычислений над графом в текущее время. Также контексту можно придать значение «в указанное время» («at specific time»), что означает, что вычисления над DG должны выполняться в конкретный момент времени. Контекст может быть переведен и в другие состояния, такие, как «для экземпляра» («for an instance») и «при инверсной кинематике» («during inverse kinematics»). Последние названные состояния являются внутренними состояниями Maya, поэтому вам не удастся установить или запросить их напрямую.

Классом, который служит для доступа к контексту, а также для его установки, является **MDGContext**. Его можно инициализировать, указав конкретное время или иной контекст. В следующем примере контекст инициализируется значением времени, соответствующим 12-му кадру.

```
MTime t( 12, MTime::kFilm );
MDGContext ctx( t );
```

Для получения значения атрибута можно использовать функцию **getValue** класса **MPlug**. По умолчанию она возвращает значение атрибута в данный момент времени. Контекст текущего времени определяется статическим членом **fsNormal** класса **MDGContext**. Прототип функции **getValue** класса **MPlug** для вещественных чисел имеет следующий вид:

```
MStatus getValue( float&, MDGContext &ctx=MDGContext::fsNormal ) const
```

Заметьте: ссылке на контекст присвоено в прототипе функции значение объекта `fsNormal`. Вы можете определить, имеет ли контекст значение текущего времени, воспользовавшись функцией `isNormal` класса `MDGContext`. До тех пор пока вы не зададите контекст явным образом, расчет подключений будет соответствовать текущему моменту времени. В следующем примере при расчете подключения используется дополнительный контекст.

```
MTIME t( 500, MTIME::kMilliseconds );
MDGContext ctx( t );
MFnDependencyNode depFn( transformObj );
MPlug transxPlg = depFn.findPlug( "translateX" );
double tx;
transxPlg.getValue( tx, ctx );
```

Блоки данных

Блоки данных являются местом хранения данных атрибута узла. Блок данных, как следует из его названия, есть не что иное, как блок памяти. Описание узла сопровождается описанием **всех** атрибутов, которые он содержит. Реальный объем памяти, необходимый для хранения тех или иных атрибутов, можно определить заранее. Требования к памяти **таких** простых атрибутов фиксированного размера, как `char`, `boolean`, `float`, `2long`, `short`, `3double` и т. д., поддаются расчету. Количество памяти, необходимое для всех этих атрибутов, суммируется. Для хранения всех данных блока выделяется один участок памяти **необходимого** размера. Это более эффективно, чем резервирование отдельных фрагментов памяти для каждого конкретного атрибута. Все атрибуты, не имеющие постоянного размера, такие, как массивы, сетки и т. д., получают отдельные участки памяти. Блок данных содержит указатель на эти участки, которые не включаются непосредственно в его состав.

Дальнейшей демонстрации сказанного поможет такой пример. Дан нестандартный узел `makePyramid`, который строит полигональную сетку в форме пирамиды. Он имеет четыре атрибута: `width`, `height`, `geom` и `capBase`. Узел и соответствующие типы данных каждого атрибута показаны на рис. 4.32.

На рис. 4.33 представлен созданный для этого узла блок данных. Он содержит контекст, речь о котором пойдет немного позднее. Все простые атрибуты: `width`, `height` и `capBase` - могут занимать непрерывный блок памяти. Атрибут `geom` имеет тип данных `mesh` переменного размера и потому хранится отдельно от блока данных. Указатель на эту сетку содержится в самом блоке.

Изначально, после своего создания, узел фактически не имеет блока данных. Блок данных узла автоматически размещается в памяти либо при создании соединения, либо при смене одного из атрибутов на значение, отличное от умалчиваемого.

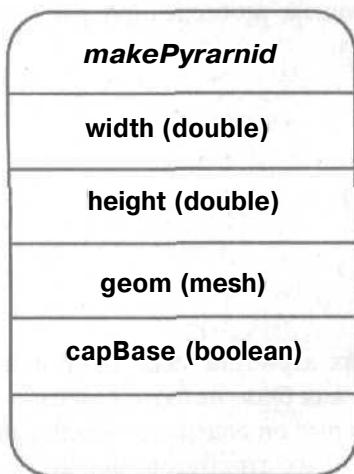


Рис. 4.32. Узел makePyramid

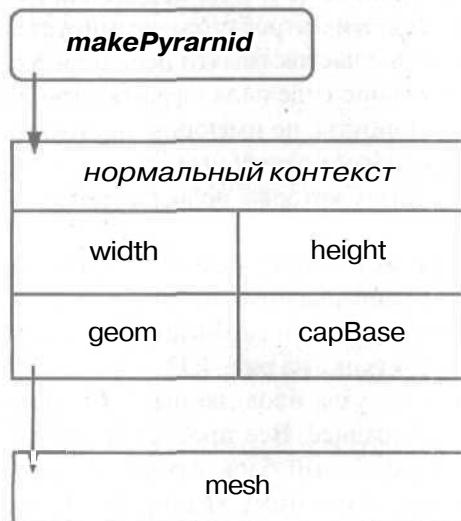


Рис. 4.33. Блок данных узла makePyramid

В действительности узел может содержать несколько блоков данных. Каждый из них имеет контекст, который с ним связан. Контекст описывает конкретный момент времени, когда производятся вычисления над узлом. Обычно это единственный контекст, именуемый *нормальным контекстом*. Этот контекст соответствует текущему времени. Вычисления над узлом могут выполняться в разные моменты времени и при различных обстоятельствах, поэтому узел может иметь ряд блоков данных для каждого отдельного контекста. На рис. 4.34 показан узел **makePyramid**, обладающий двумя блоками данных. Первый блок данных содержит данные узла, когда вычисления выполняются в текущий момент (нормальный контекст). Второй блок данных содержит данные узла, когда вычисления производятся в момент времени, равный 2.3 секунды.

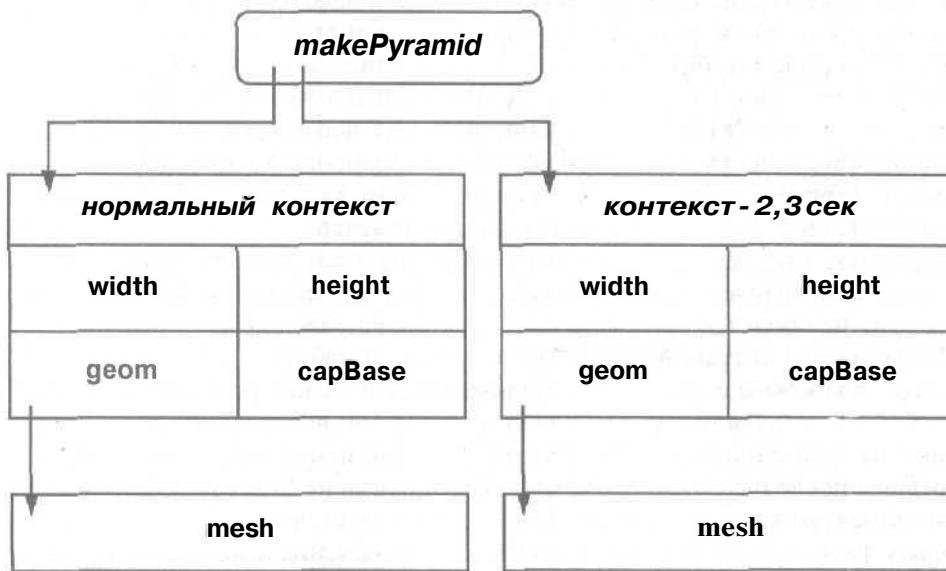


Рис. 4.34. Несколько блоков данных

Так как вычисления над большинством узлов выполняются в текущий момент времени, эти узлы часто содержат единственный блок данных. Дополнительные блоки данных создаются и удаляются по мере необходимости. Узел может инициировать создание блока данных в указанный момент времени при помощи функции **forceCache** класса **MPxNode**. Для получения контекста конкретного блока данных служит функция **context** класса **MDataBlock**.

Классы **MDataHandle** и **MArrayDataHandle** - это простые объекты, которые знают о распределении памяти в блоке данных. Работая с простыми типами,

пригодными для хранения непосредственно в блоке, они способны организовать эффективное обращение к памяти. При работе с информацией, расположенной вне блока, названные классы пользуются указателями для разыменования соответствующих данных.

Флаги распространения

Во введении упоминалось о том, что смена одного атрибута влечет за собой установку битов изменений всех других, зависимых от него атрибутов, включая выходные соединения. Бит изменений распространяется по всем атрибутам, которые прямо или косвенно затрагивает изменившийся атрибут. В сложной сети даже простое распространение признаков изменений может занять определенное время. Для сокращения этих издержек каждое подключение снабжено флагом *распространения*. Если флаг распространения равен `true`, то при получении сообщения, содержащего бит изменений, подключение установит внутренний бит изменений и передаст это сообщение другим подключениям. Если же флаг распространения равен `false`, подключение не будет пересылать сообщение с битом изменений далее. На практике флаг распространения не допускает передачи сообщений с битами изменений тем подключениям, флаги которых уже должны быть установлены. Если данное подключение помечено как измененное, можно предположить, что оно передало это сообщение всем другим подключениям, на которые само влияет. Таким образом, если вы пометите его как измененное еще раз, ему не придется распространять свое сообщение снова.

В большинстве ситуаций подобный метод будет работать просто замечательно. Однако возможны случаи, когда подключение не имеет пометки «изменено», в то время как подключения, соединенные с ним по восходящим соединениям, ее имеют. В этой ситуации их сообщения с битами изменений не были корректно переданы последнему подключению. Оно никогда не будет рассчитано вновь, так как помечено как достоверное. Оно никогда не узнает о том, что изменено, поскольку флаг распространения входящего подключения имеет значение `false`.

Даже в том случае, если такое произойдет, это, к счастью, можно исправить. Используйте команду `dgdirty` для установки всех подключений узла в положение «изменено» или «достоверно». Команда вызовет принудительную рассылку сообщения с битом изменений всем узлам, которых это касается, независимо от их текущих настроек флага распространения.

`dgdirty $nodeName;`

Чтобы пометить все подключения как достоверные, воспользуйтесь следующей командой:

`dgdirty -clean $nodeName;`

Обработка сквозного прохода

Все нестандартные узлы прямо или косвенно порождены от **MPxNode**. Этот узел содержит несколько атрибутов, однако самым важным для разработчика является атрибут **nodeState**. Он доступен через интерфейс Maya после выполнения следующих действий.

1. Выделите узел.
2. Откройте редактор **Attribute Editor**.
3. Раскройте шире элемент **Node Behavior** (Поведение узла).
4. Выберите новое значение из выпадающего списка рядом с приглашением **Node State** (Состояние узла).

На программном уровне атрибут **nodeState** представлен перечислимым типом с четырьмя значениями.

- * 0 (нормальное)
- * 1 (сквозной проход)
- ◆ 2 (блокировка)
- ◆ 3 (внутренне запрещен)

Атрибут **nodeState** является признаком того, должен ли узел вычислять свои выходные атрибуты. Обычно **nodeState** имеет значение 0 (нормальное), поэтому узел **их** действительно вычисляет. Если значение **nodeState** равно 1 (сквозной проход), входные атрибуты **узла** передаются на его выход без каких бы то ни было вычислений. Это состояние приведено в редакторе **Attribute Editor** как **hasNoEffect**.

При разработке узла вам нужно решить, будете ли вы поддерживать состояние сквозного прохода. На деле это определяется допустимостью понятия сквозного прохода для узлов вашего типа. Так, деформаторы должны поддерживать это состояние. Если атрибут **nodeState** имеет значение «сквозной проход», необходимо просто передать входной геометрический объект на выход, не производя никакой его деформации.

Следующий код показывает, как нужно изменить подключаемый модуль **SwirlDeformer** ради поддержки этого состояния узла. Прежде чем выполнять деформацию, функция проверяет, равен ли атрибут **nodeState** единице (сквозной проход). Если да, функция сразу же завершается, не деформируя геометрию.

```
MStatus SwirlDeformer::deform( MDataBlock& block, MItGeometry &iter,
                               const MMatrix &localToWorld,
                               unsigned int geomIndex )
{
    MStatus stat;

    MDataHandle stateHnd = data.inputValue( state );
    int state = stateHnd.asInt();
    if( state == 1 )// Сквозной проход
        return MS::kSuccess;

    MDataHandle envData = block.inputValue( envelope );
    float env = envData.asFloat();

    ...
}
```

Циклические зависимости

Циклические зависимости могут возникать, когда выход одного узла поступает на вход другого, а тот затем передается на вход первого. Между двумя узлами могут быть и иные промежуточные узлы, однако важнейшее свойство зависимости состоит в том, что выйдя из первого узла и начав обход его исходящих соединений, вы в конечном итоге вернетесь в тот самый узел.

Граф DG действительно обрабатывает циклические зависимости, хотя результаты не всегда могут совпадать с ожидаемыми. Чаще всего они зависят от того, какой из узлов вычисляется первым. Ввиду неопределенности результата циклических зависимостей лучше избегать вовсе.

Приложение А Дополнительные ресурсы

Продолжить изучение программирования в среде Maya вам поможет огромное число разнообразных доступных онлайновых и офлайновых ресурсов.

Сетевые ресурсы

В сети Интернет представлено обширное множество ресурсов, посвященных вопросам программирования задач компьютерной графики. Несмотря на то что вы можете найти информацию по этой теме, обратившись к своей любимой поисковой машине, есть несколько узлов, где содержатся специальные сведения о программировании пакета Maya.

Web - сайт книги

Официальный Web-сайт этой книги находится по адресу www.davidgould.com. Перед вами, хотя и неполный, перечень информации, доступной на этом сайте.

- ◆ Исходный код сценариев на языке MEL и программ на языке C++ для всех примеров из этой книги,
- * Дополнительные примеры сценариев на языке MEL.
- ◆ Дополнительные примеры исходного кода с использованием C++ API.
- * Список допущенных в книге опечаток⁷.
- * Постоянно обновляемый словарь терминов.
- ◆ Обновляемый список других Web-сайтов и сетевых ресурсов по данной тематике.

⁷ Все опечатки, обнаруженные на момент подготовки русского перевода книги в настоящем издании исправлены. – Примеч. перев.

Дополнительные Web - сайты

Особого внимания заслуживают [следующие Web-сайты](#), на которых приведена специальная информация о программировании Maya и размещены форумы по теме.

Alias \ Wavefront

www.aliaswavefront.com

На этом сайте находится полная справочная информация по системе Maya. Сайт содержит самую последнюю информацию о разработке продуктов, примеры сценариев и подключаемые модули. При желании заняться созданием коммерческих модулей обязательно ознакомьтесь с программой **Alias | Wavefront Conductors**. Благодаря этой программе компания Alias | Wavefront обеспечивает производственную и маркетинговую поддержку разработчиков, стремящихся перевести свои продукты на коммерческую основу. Поддержка оказывается даже тем [разработчикам](#), кто хотел бы распространять свои модули как условно-бесплатное или свободное программное обеспечение.

Highend3D

www.highend3d.com

Сайт **Highend3D**, наиболее широко известный своим глубоким анализом основных пакетов анимации и моделирования, тоже является прекрасным архивом сценариев MEL и подключаемых модулей. Кроме того, на нем размещается форум **Maya Developers Forum**, где вы можете задавать свои вопросы другим разработчикам.

Bryan Ewert

www.ewertb.com/maya

Этот сайт содержит большое число обучающих [материалов](#) по MEL и C++ API, а также много примеров. Рассмотрены основы языка MEL и некоторые более сложные вопросы. Разделы «How To» («Инструкции») представляют интерес для разработчиков, нуждающихся в разрешении конкретной проблемы.

Система Maya

Система **Maya** поставляется с обширным набором документации по программированию и множеством примеров. Кроме того, не забывайте, что нередко вы можете узнать о том, как **Maya** решает тот или иной круг задач, включив опцию **Show AH Commands** в редакторе **Script Editor**. Понапалу это может стать прекрасным руководством, если вы захотите сделать нечто подобное.

Документация

В частности, постоянным источником важной информации по вопросам программирования станут справочные материалы о языке MEL и интерфейсе C++ API. К любой документации в среде Maya можно обратиться, нажав клавишу F1 или выбрав пункт главного меню Help. Обратите, пожалуйста, внимание, что приведенные ниже ссылки могут зависеть от того, какой версией Maya и каким языком вы пользуетесь.

Изучение языка MEL (Learning MEL)

maya_instalAdocs\en_US\html\Use rGuide\Mel\Mel.htm
fflaya_install\docs\en_US\htnl\InstantMaya\InstantMaya\InstantMaya.htm

(Перейдите в конец раздела для обращения к обучающим материалам по Expression и MEL.)

Изучение C++ API (Learning C++ API)

maya_install\docs\en_US\html\DevKit\PlugInsAPI\PlugInsAPI.htm

За получением справочных материалов по общим вопросам программирования обратитесь к разделу Reference Library (Справочная библиотека) в **Master Index** (Главный указатель). Справочники по конкретным аспектам программирования находятся в следующих разделах.

Справочник по языку MEL (MEL Reference)

maya_install\docs\en_US\html\Commands\Index\index.html
maya_install\docs\en_US\html\Nodes\Index\indexAlpha.html

Справочник по C++ API (C++ API Reference)

maya_install\docs\en_US\html\DevKit\PlugInsAPI\classDoc\index.html
maya_install\docs\en_US\html\Nodes\Index\indexAlpha.html

Примеры

Стандартная поставка Maya содержит большое число примеров сценариев на языке MEL и подключаемых модулей на C++.

Примеры на языке MEL (MEL Examples)

Интерфейс Maya полностью написан на языке MEL, поэтому сценарии, которые им **используются**, входят в состав приложения. Нет лучше способа проникнуть в суть составления профессиональных сценариев MEL, чем **посмотреть**.

реть на сценарии, написанные разработчиками Maya. Множество подобных сценариев **вы** найдете в следующем каталоге, а также его подкаталогах.

flraya_Install\scripts

Я настоятельно рекомендую внимательно просмотреть **их**, чтобы понять, как пишутся грамотно разработанные сценарии. Однако прошу вас обратить внимание и на то, что все предоставленные сценарии охраняются авторским правом компании Alias | Wavefront и их нельзя применять в ваших собственных разработках. Изучите эти сценарии, чтобы они послужили вам хорошим **примером**, но не поддавайтесь искушению скопировать их и использовать без изменений.

Кроме того, будьте осторожны при просмотре сценариев, с тем чтобы нечаянно не внести в них никаких изменений. Эти сценарии используются средой Maya, поэтому любые изменения могут привести к нестабильной работе системы и, возможно, ее краху. По этой причине лучше всего сделать копию этих сценариев **заранее**.

Примеры с использованием C++ API (C++ API Examples)

В каталогах с примерами, написанными с использованием C++ API, содержится исходный код подключаемых модулей и отдельных приложений, а также серверов захвата движений. Эти примеры расположены в каталоге:

maya_install\devkit

Приложение В

MEL для программистов на языке С

Читатели, знакомые с программированием на языке С, могут нисколько не удивиться, когда, впервые взглянув на команду или сценарий MEL, заметят большое сходство с синтаксисом, применяемым в языке С. Действительно, MEL иногда неофициально именуют «С со значком \$». Хотя это и не лишено смысла, есть несколько важных признаков, по которым языки отличаются друг от друга.

Приведем перечень существенных несоответствий.

- * MEL спроектирован как язык быстрого составления прототипов, более доступный для недостаточно опытных программистов, поэтому он избавлен от многих системных функций низкого уровня, которыми обременен разработчик на языке С. Одна из таких функций - выделение и освобождение памяти. MEL предоставляет удобные динамические массивы, поэтому вам не потребуется составлять функции увеличения и уменьшения их размера. MEL занимается выделением и очисткой памяти без вашего участия, тем самым упрощая код и снижая риск возникновения таких связанных с памятью проблем, как ее утечки и ошибки сегментации.
- ◆ MEL не имеет указателей. Все переменные, за исключением массивов, передаются в процедуры по своему значению. Все массивы передаются по ссылке.
- * Если вы не инициализируете локальные переменные, MEL присвоит им значения по умолчанию.
- ◆ Переменные, описанные в самой внешней области памяти, по умолчанию являются локальными, пока вы явно не опишете их как глобальные с использованием ключевого слова `global`. Это правило видимости данных противоположно аналогичному правилу в языке С, где переменная, описанная в модуле, по умолчанию имеет глобальную область видимости, если вы не укажете, что она является статической (`static`).
- ◆ Тип `float` в языке MEL эквивалентен типу `double` в языке С. Несмотря на то что конкретное число разрядов этого типа является машинозависимым,

обычно оно превышает число разрядов того типа, который служит для представления переменных `float` языка C.

- Ф Тип `int` языка MEL является машинозависимым, однако в большинстве случаев это 32-разрядное целое число со знаком.
- Ф MEL имеет встроенный строковый тип `string`. Над ним можно выполнять множество различных операций, включая сцепление строк.
- * В MEL отсутствуют поразрядные операции (`|`, `&`, `!`, `~` и т. д.).
- Ф MEL не обладает средствами преобразования типов. Вам не удастся, скажем, преобразовать целое значение в вещественное, записав (`float`). Для приведения одного типа к другому просто присвойте значение нужного типа требуемой переменной. К примеру, чтобы преобразовать целое число в вещественное значение, используйте следующие операторы:

```
int $intA = 23;  
float $fltA = $intA; // Приведение к типу float  
... используйте $fltA в операциях
```

Заметьте, что значения не всех типов могут присваиваться переменным других типов.

Преобразование из строки в число выполняется очень просто. В отличие от языка C, вы можете лишь присвоить эту строку некой числовой переменной. Этот прием будет работать, если строка содержит допустимое числовое значение.

```
string $v = "123";  
int $num = $v; // Значение $num теперь равно 123
```

- Ф К логическим константам относятся `true`, `false`, `on`, `off`, `yes`, `no`. Эти константы имеют стандартное числовое значение, равное 1 для `true` и 0 для `false`⁸. Как и в языке C, любой оператор, выполнение которого не приводит к значению 0, рассматривается как истина (`true`).
- Ф Процедуры могут описываться внутри блоков. Однако вам не удастся описать одну процедуру внутри другой.
- Ф Аргументы процедур не могут иметь присвоенных им значений по умолчанию. Поэтому следующая запись окажется недопустимой:

```
proc myScale( string $nodeName = "sphere", float $factor = 1.0 )  
{  
    ...  
}
```

⁸ Точнее говоря, значение 1 имеют константы `true`, `on`, `yes`; значение 0 - все остальные. – Примеч. перев.

- ◆ Процедуры нельзя перегружать. При описании новой процедуры с тем же именем она перекрывает, а не перегружает любое ранее сделанное описание.
- ◆ В отличие от языка С, в операторе switch можно использовать строки. Это демонстрирует следующий пример.

```
string $name = "box";
switch( $name )
{
    case "sphere";
        print "found a sphere";           // "найден объект sphere"
        break;

    case "box";
        print "found a box";            // "найден объект box"
        break;

    default:
        print "found something else"; // "найдено что-то другое"
        break;
}
```

- ◆ Наряду со всеми типовыми конструкциями управления потоком команд (for, while, do-while, switch) MEL имеет циклическую конструкцию **for-in**, которая в точности повторяет обычный цикл for за исключением того, что допускает более короткую запись.

В отличие от языка С, в конструкции цикла for нельзя описывать переменные. К примеру, следующая запись в MEL будет недопустимой:

```
for( int $i=0; ...
```

Переменная должна быть описана заранее:

```
int $i;
for( $i=0; ...
```

Приложение С

Литература для дальнейшего изучения

Компьютерная графика включает в себя широкий спектр дисциплин. Однако ее фундаментом, безусловно, является математика. Основу почти всей теории и практики компьютерной графики составляют, в частности, дискретная математика, линейная алгебра и математический анализ. Помимо знания математики, в разработке эффективных и надежных программ вам поможет глубокое понимание вопросов практики программирования.

Обладая хорошими знаниями в области математики и программирования, вы обретете прочную базу, опираясь на которую сможете изучать конкретные приемы и методы, используемые в компьютерной графике. Даже несмотря на постоянное развитие этой области в ней есть множество принципов, изучив которые один раз, вы обеспечите себе неплохую основу для будущей деятельности.

Ниже приведен неполный перечень книг, которые дадут вам хорошую подготовку в соответствующей области. В каждом из этих разделов книги перечислены в порядке от простого к сложному.

Математика

Selby, Peter, and Steve Slavin. *Practical Algebra*. New York: John Wiley and Sons, 1991.

Thompson, Silvanus P., and Martin Gardner. *Calculus Made Easy*. New York: St. Martin's Press, 1998.

Mortenson, Michael E. *Mathematics for Computer Graphics Applications*. New York: Industrial Press, 1999.

Lengyel E. *Mathematics for 3D Game Programming and Computer Graphics*. Hingham, Mass.: Charles River Media, 2001.

Программирование

Общие вопросы

- Deitel, Harvey M., and Paul J. Deitel. *C: How to Program*. Upper Saddle River, N.J.: Prentice Hall, 2000. (Имеется русский перевод: Дейтел Х.М., Дейтел П.Дж. Как программировать на С. М.: Бином, 2002.)
- Knuth, Donald E. *The Art of Computer Programming*, 3 vols. Boston: Addison-Wesley Publishing Co., 1998. (Имеется русский перевод: Кнут Д.Э. Искусство программирования: В 3 т. М.: Вильяме, 2000; 2001.)
- Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge: MIT Press, 2001.

Язык C++

- Liberty, Jesse. *Sams Teach Yourself C++ in 21 Days Complete Compiler Edition*. Indianapolis: Sams Technical Publishing, 2001. (Имеется русский перевод: Либерти Д. Освой самостоятельно C++ за 21 день. М.: Вильяме, 2001.)
- Deitel, Harvey M., and Paul J. Deitel. *C++: How to Program*. Upper Saddle River, N.J.: Prentice Hall, 2000. (Имеется русский перевод: Дейтел Х.М., Дейтел П.Дж. Как программировать наC++. М.: Бином, 2001.)
- Stroustrup, Bjarne. *The C++ Programming Language*. Boston: Addison-Wesley Publishing Co., 2000. (Имеется русский перевод: Струstrup Б. Язык программирования C++. М.: Бином, 2001.)
- Meyers, Scott. *Effective C++ : 50 Specific Ways to Improve Your Programs and Design*. Boston: Addison-Wesley Publishing Co., 1997.
- Bulka, Dov, and David Mayhew. *Efficient C++: Performance Programming Techniques*. Boston: Addison-Wesley Publishing Co., 1999.

Компьютерная графика

Общие вопросы

- Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C*. Boston: Addison-Wesley Publishing Co., 1995.
- Watt, Alan H. *3D Computer Graphics*. Boston: Addison-Wesley Publishing Co., 1999.

Glassner, Andrew S. *Graphics Gems I*. San Francisco: Morgan Kaufmann Publishers, 1990.

См. также: *Graphics Gems II, III, IV, V*.

Моделирование

Rogers, David F. *An Introduction to NURBS, with Historical Perspective*. San Francisco: Morgan Kaufmann Publishers, 2001.

Warren, Joe, and Henrik Weimer. *Subdivision Methods for Geometric Design: A Constructive Approach*. San Francisco: Morgan Kaufmann Publishers, 2001.

Анимация

Parent, Rick. *Computer Animation: Algorithms and Techniques*. San Francisco: Morgan Kaufmann Publishers, 2002.

Синтез изображений

Watt, Alan, and Mark Watt. *Advanced Animation and Rendering Techniques*. New York: ACM Press, 1992.

Glassner, Andrew S. *Principle of Digital Image Synthesis*. San Francisco: Morgan Kaufmann Publishers, 1995.

Ebert, David S., et al. *Texturing and Modeling*. San Diego: Academic Press, 1998.

Shirley, Peter. *Realistic Ray Tracing*. Natuk, Mass.: A K Peters Ltd., 2000.

Blinn, James. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. San Francisco: Morgan Kaufmann Publishers, 1996.

Словарь терминов

ANSI - сокращенное наименование Национального института стандартизации США (American National Standards Institute). Названный институт принимает участие в разработке стандартов многих компьютерных языков, в том числе С и C++.

API - сокращенное обозначение *интерфейса прикладного программирования* (application programming interface). Такие интерфейсы входят в состав операционных и прикладных систем. Каждый прикладной интерфейс содержит описание методов, посредством которых программисты могут обращаться к данной системе и управлять ею.

ASCII - сокращенное обозначение американского стандартного кода обмена информацией (American Standard Code for Information Interchange). Представляет собой систему 7-разрядного кодирования символов.

boolean [логическое (булево) значение] - служит для обозначения результата логической операции. Булево значение может быть истинным (true) либо ложным (false).

C++ - объектно-ориентированный язык программирования, в основе которого лежит язык С.

DG — сокращенное обозначение *графа зависимости* (Dependency Graph). Граф DG состоит из всех узлов Maya и соединений между ними.

double (вещественное число двойной точности) - в языке C++ это тип данных, который предназначен для хранения чисел с несколькими десятичными разрядами после запятой. Часто, хотя и не всегда, имеет большую точность по сравнению с типом данных float.

ELF - сокращенное обозначение *системы расширенных слоев* (extended layer framework). Эта система служит для описания интерфейсов в языке MEL. Проектирование интерфейсов требует создания общих схем размещения, в которых располагаются элементы управления. Система поддерживает произвольную вложенность схем размещения и элементов управления,

Expression - в контексте Maya - ряд команд языка MEL, управляющих одним или несколькими атрибутами узла. Эти команды позволяют реализовать программное управление атрибутами.

float (вещественное число одинарной точности) - тип данных, предназначенный для хранения чисел с несколькими десятичными разрядами после запятой. Размер типа **float** в языке MEL не обязательно совпадает с размером этого же типа в языке C++.

GUI - сокращенное обозначение *графического интерфейса пользователя* (graphical user interface). Представляет собой систему окон, диалогов и других элементов пользовательского интерфейса, с которыми вы можете взаимодействовать при использовании пакета Maya.

int - тип данных, который служит для хранения целых чисел.

MEL - сокращенное обозначение языка Maya Embedded Language. Это встроенный интерпретируемый язык сценариев Maya. Синтаксически очень напоминает язык C, однако более прост в изучении и позволяет быстро составлять программы, необходимые для доступа к функциям Maya и управления пакетом.

NURBS -- сокращенное обозначение *неравномерных рациональных бисплайнов* (nonuniform rational B-splines). NURBS служат для математического представления гладких кривых и *поверхностей*.

string (строка) - последовательность символов; **текст**.

underworld - параметрическое пространство (u, v), отличное от декартова пространства (x, y, z). Положение в параметрическом пространстве гарантированно связано с базисным параметрическим объектом (NURBS-кривой или поверхностью).

анимация по ключевым кадрам (keyframe animation) - при использовании анимации по ключевым кадрам необходимо выбрать параметр анимации, указав его точное значение в заданные моменты времени. Тогда компьютер сможет самостоятельно выполнить интерполяцию и рассчитать, какие значения должен принимать этот параметр в интервалах между **ключевыми кадрами**.

аргумент (argument) - команды или процедуры - это значение, подаваемое на ее вход при **выполнении** операции.

атрибут (attribute) - то или иное свойство узла. Например, узел `makeNurbsSphere` имеет атрибут `radius`. При изменении этого атрибута сфера меняет свой размер.

атрибут уровня объекта (per object attribute) - единый атрибут, используемый всеми частицами.

атрибут уровня частицы (per particle attribute) - атрибут, имеющийся у каждой частицы.

аффинное преобразование (affine transformation) - преобразование, состоящее из линейного преобразования и последующей трансляции (поворота).

библиотека (library) - в контексте языка C++ - хранилище функций, которые могут использоваться другими программами. Библиотека для работы с файлами поддерживает их создание, открытие и редактирование. Пользуясь библиотеками в своей программе, вы избавляетесь от необходимости самостоятельной разработки некоторых технологий.

бит изменений (dirty bit) - флаг, который входит в состав атрибута и указывает на необходимость его обновления.

вектор (vector) - задает направления. Кроме того, векторы обладают размером, соответствующим их длине. Вектор, длина которого равна 1, называется *единичным*.

векторное произведение (cross product) - двух векторов - новый вектор, перпендикулярный обоим исходным векторам. Эта операция часто используется для определения направления вектора, являющегося нормалью к поверхности.

висячий узел (orphaned node) - узел, ранее связанный с другими узлами, но к настоящему времени утративший все свои соединения.

вращать/вращение, поворачивать/поворот (rotate/rotation) - *объект/объекта* – значит крутить его. Эта операция изменяет ориентацию объекта. Точка, относительно которой выполняется вращение объекта, называется *точкой вращения (поворота)*. Точка вращения колеса расположена в его центре.

входной атрибут (input attribute) - атрибут, соответствующий входу узла. Значение входного атрибута нередко используется функцией `compute` для вычисления значения одного или нескольких выходных атрибутов.

входящая касательная (in-tangent) - определяет скорость, с которой анимационная кривая сближается с ключевым кадром.

выполнить рендеринг (render) - Используя информационные модели сцены, источники освещения, камеру и др., получить окончательное изображение.

выполнить трансляцию/трансляция (translate/translation); трансляция объекта - то же, что перенос.

выражение времени выполнения (runtime expression) - выражение, которое выполняется тогда, когда возраст частицы превышает 0.

выражение времени создания (creation expression) - выражение, которое выполняется в тот момент, когда возраст частицы равен 0, т. е. в момент ее рождения.

выходной атрибут (output attribute) - атрибут, содержащий результат вычислений. Имеющаяся в составе узла функция `comprite` принимает на вход один или несколько входных атрибутов, а затем рассчитывает выходное значение, сохраняющееся в выходном атрибуте.

глобальная (область видимости) (global) - термин характеризует область видимости переменных или процедур. Если они описаны как глобальные, то к ним можно обращаться из любой точки программы.

группа (group) - в пакете Maya - узел `transform`, который становится родителем всех входящих в нее узлов. Все потомки узла `transform` образуют группу.

дву направленная модель (push-pull model) - концептуальная модель, в которой данные одновременно «вытягиваются» и «проталкиваются» сквозь множество узлов. Эта модель более эффективна, нежели поток **данных**, поскольку предполагает обновление лишь тех узлов, которые необходимо обновить. По этому принципу работает граф зависимости Maya.

действие (action) - команда языка MEL, которая не вносит изменений в состояние Maya. Действие часто инициирует запрос к сцене, но не изменяет ее.

декартовы координаты (Cartesian coordinates) - система координат, в которой положение определяется на основании проекций на множество ортогональных осей.

дерево (tree) - метафора, которая служит для описания **иерархий**.

деформатор (deformer) - берет одну или несколько точек и переносит их на новое место.

динамически подключаемая библиотека (dynamic link library) - библиотека программных функций, загружаемая в память только по мере необходимости.

динамический атрибут (dynamic attribute) - атрибут, добавляемый к тому или иному узлу. Такой атрибут не находится в совместном пользовании всех узлов этого типа, а однозначно принадлежит только данному узлу.

доводка (tweak) — термин относится к процессу редактирования чего-либо. Выполняя доводку объекта, вы лишь его изменяете. Прежде чем сцена приобретет законченный вид, она часто проходит множество стадий художественного и технического редактирования.

единичная матрица (identity matrix) - матрица преобразования, не оказы-вающая никакого влияния на ту точку, по отношению к которой она применяется. С технической точки зрения, такая матрица полностью состоит из нулей, и лишь ее диагональ заполнена единицами.

зависимый атрибут (dependent attribute) - это выходной атрибут. Если итоговое значение одного атрибута зависит от значений других атрибутов, то первый из них считается зависимым. Для установки отношения зависимости между атрибутами служит функция `MPxNode::attributeAffects`.

законы пружины (spring laws) - определяют реакцию множества пружин на данную совокупность сил, которые к ним приложены.

значение инициализации (initialization) - значение, присваиваемое переменной при первоначальном описании.

иерархия (hierarchy) - любая система, в которой имеются отношения «родитель - потомок».

иерархия классов (class hierarchy) - при использовании стандартных методов объектно-ориентированного проектирования большинство сложных систем разбиваются по принципу иерархии. Корнем (вершиной) иерархии становится класс, обладающий функциями наиболее общего характера. От него порождаются другие классы (**потомки**), которые наделяют его более специальными **функциями**. В ходе этого процесса образуется дерево иерархии классов.

иерархия преобразований (transformation hierarchy) - отдельное преобразование позволяет задать положение, ориентацию и размер данного объекта. Поместив такое преобразование в иерархическую структуру, можно получить ряд преобразований, которые будут применяться по отношению к объекту. При трансформации родительского объекта его потомок подвергается тем же преобразованиям.

ИК-решатель (IK-solver) - Maya позволяет создавать пользовательские системы инверсной кинематики. ИК-решатель определяет ориентацию промежуточных суставов.

инверсная **кинематика/ИК** (inverse kinematics/IK) - посредством инверсной кинематики аниматор может управлять множеством суставов, просто указав место расположения последнего из них. Все промежуточные суставы рассчитываются компьютером.

инструмент (tool) - определяет ряд конкретных шагов, которые **нужно** пройти для выполнения операции. Нередко до завершения операции инструменты требуют от пользователя, чтобы тот выбрал что-нибудь **мышью**.

интерпретируемый язык (interpreted language) - компьютерный язык, исходный код на котором интерпретируется и сразу же выполняется. В этом состоит его отличие от компилируемого языка, где исходный код до своего выполнения сначала должен быть скомпилирован, а затем собран. Интерпретируемые языки, как правило, медленнее компилируемых, хотя нередко они лучше подходят для быстрого создания прототипов.

интерфейс (interface) - специальные методы взаимодействия, позволяющие общаться с системой. Графический интерфейс пользователя предусматривает набор графических элементов, которыми вы пользуетесь, чтобы сообщить вычислительной системе о своих намерениях.

исходящая касательная (out-tangent) - определяет скорость, с которой анимационная кривая покидает ключевой кадр.

кадрируемый (keyable) атрибут - атрибут, который можно анимировать путем установки ключевых кадров, называется **кадрируемым**.

касательная (tangent) - в контексте ключевых кадров на анимационной кривой касательная определяет способ интерполяции значений атрибута между последовательными ключевыми кадрами. Изменяя касательную в ключевых кадрах, можно ускорить или замедлить анимацию между **ними**.

класс (class) - в языке C++ основной элемент описания самодостаточного объекта. Классы имеют собственные функции и члены **данных**.

ключ разбиения (breakdown key) - **кадр**, который зависит от предшествующих и следующих за ним ключей. При перемещении других кадров он автоматически определяет свою позицию относительно них.

ключи, управляемые множеством (set-driven keys) - применяются для задания отношений между двумя параметрами. В отличие от ключевых кадров, предполагающих использование времени, **ключ**, управляемый множеством, может быть основан на любом параметре (драйвере), который управляет другим параметром. Для описания этого отношения необходимо выполнить редактирование кривой.

команда (command) - служит для выполнения различных операций. Команда `sphere`, например, предназначена для создания и редактирования сфер. Для выполнения практически всех операций Maya повсеместно используются те или иные команды.

командные режимы (command modes) - одна и та же команда может работать в таких режимах, как создание, правка и запрос. При выполнении команды в указанном режиме она выполняет ограниченный набор операций. В режиме запроса команда возвращает требуемое значение. В режиме создания она создает нечто новое.

комментарий (comment) - некоторый текст описательного характера, который вводится программистом в исходный код, для того чтобы другие люди могли прочитать программу и понять те действия, которые она выполняет. Комментарий служит средством документирования функций программы. *Многострочный комментарий* - это комментарий, занимающий более одной строки текста.

компиляция-сборка (compile-and-link) - чтобы запустить программу, компилируемые языки, такие, как C и C++, требуют компиляции и сборки исходного кода в машинное представление. Этим они отличаются от языков сценариев наподобие `MEL`, которые интерпретируют инструкции и немедленно их выполняют.

компонент (component) - отдельное значение, входящее в **вектор**, точку или другой элемент данных. Точка имеет три компонента: *x*, *y*, *z*.

конвейер (pipeline) - в практике студийной работы конвейер включает все разнообразные стадии производства фильма. Процесс начинается с моделирования, которое сменяется этапом **анимации**, работы со светом и, наконец,рендерингом; нередко в организации конвейера участвуют отдельные специализированные отделы студии,

контекст (context) - при вызове функции `compute`, входящей в состав узла, контекст определяет, в какой момент времени выполняются вычисления над узлом.

контроллер анимации (animation controller) - многие пакеты трехмерной графики содержат специальные функции анимации объектов. В 3dsmax они называются контроллерами. В программе Softimage их называют f-кривыми. В пакете Maya стандартные элементы управления анимацией представлены узлами animCurve. Они позволяют создавать и **редактировать** кривые, управляющие параметром в диапазоне анимации.

корень (root) - воображаемый узел, являющийся родителем всех остальных узлов сцены. Существует понятие корневого узла, поэтому сцену полностью можно представить как дерево, которое начинается в таком корне.

локальная (область видимости) (local) - термин **описывает** метод доступа к процедурам и переменным. Объявив их как локальные, вы сможете обращаться к ним только **из** данного файла сценария или из текущего блока.

локальное пространство (local space) - пространство координат, в котором происходит первоначальное описание объекта. В этом пространстве над объектом не выполняются никакие преобразования.

локатор (locator) - трехмерный объект, отображаемый в среде Maya. Локаторы отсутствуют в окончательном варианте изображения.

манипулятор (manipulator) - визуальный элемент управления, посредством которого пользователь может изменять атрибуты в трехмерном пространстве.

массив (array) – перечень элементов.

масштаб (scale) - под масштабированием объекта понимают изменение его размеров. Масштабирование называют равномерным, если размеры объекта изменяются в одинаковой мере. При неравномерном масштабировании объект может стать шире, выше или глубже, утратив свои исходные пропорции.

матрица (matrix) - совокупность строк и столбцов чисел. В компьютерной графике матрицы служат для выполнения преобразований над точками.

матрица преобразования (transformation matrix) - способ краткой записи, который предназначен для описания **позиционирования**, вращения и задания размеров объекта. В результате применения матрицы преобразования по отношению к объекту тот часто занимает иное положение, приобретает иную ориентацию в пространстве и принимает иные размеры. **Обратная** матрица преобразования восстанавливает исходное положение, исходную ориентацию и исходные размеры объекта.

мировое пространство (world space) - пространство, в котором отображаются все объекты сцены. Мировое пространство является результатом всех преобразований, выполненных над родительскими узлами объекта.

множество (set) — перечень элементов. Когда объект помещается в некоторое множество, он становится частью этого перечня.

модальный (modal) - модальное диалоговое окно не позволяет обращаться к главному окну приложения до своего закрытия.

модель потока данных (data flow model) - концептуальная модель, в которой данные передаются по цепочке узлов, с первого до последнего. В каждом следующем узле данные изменяются.

набор **функций** (function set) - в соответствии с принятой в Maya схемой разделения данных и функций для их обработки - класс **C++**, предоставляющий **программисту** возможность доступа к данным. Набор функций можно использовать для создания, редактирования объектов и выполнения запросов к **данным**, при этом не обязательно знать подробности его реализации.

нажатие клавиши (keystroke) - происходит, когда вы нажимаете клавишу на клавиатуре.

начальное число (seed) - число, которое служит для инициализации генератора случайных чисел.

нормаль (normal) - вектор, перпендикулярный к поверхности.

ОАГ (DAG) - сокращенное обозначение *ориентированного ациклического графа* (*directed acyclic graph*). Этим техническим термином называют иерархическую структуру, в которой ни один из потомков не может находиться среди узлов-родителей самого себя. Пройдя по этой структуре от первого до последнего узла, вы никогда не встретите один и тот же узел дважды.

область видимости (scope) — область видимости переменной определяет, можно ли обратиться к последней. Если переменная имеет глобальную область видимости, к ней можно обратиться отовсюду. Если переменная имеет локальную область видимости, к ней можно обратиться только из того блока, в котором она описана, а также из всех внутренних блоков.

образование покрова (skinning) - процесс, в ходе которого вокруг скелета выстраивается модель. Когда скелет движется, соответствующим образом движется и модель. Модель образует кожный или другой поверхностный слой, окружающий суставы скелета.

объектное пространство (object space) - См. **локальное пространство**.

оператор (operator) - метод краткой записи, используемый для обозначения операции над одним или несколькими значениями. Оператор сложения записывается при помощи знака «плюс» (+). Среди других операторов - умножение (*), деление (/) и др.

писатель, «рукоятка» (handle) - нечто, предоставляемое системой для обращения к некоторому объекту.

ось (axis) - задает **направление**. Трехмерные объекты **имеют** три оси: x, y, z.

от первого ключевого кадра до бесконечности (preinfinity) - любой кадр, который предшествует первому ключевому кадру на анимационной кривой.

от последнего ключевого кадра до бесконечности (postinfinity) - любой кадр, который расположен за последним ключевым кадром на анимационной кривой.

отмена (undo) - чтобы аннулировать результат команды, ее можно отменить. Отмена команды возвращает Maya в **состояние**, предшествовавшее ее выполнению.

параметрическое пространство (parametric space) - положение, определяемое посредством параметрических координат (u, v), а не путем явного использования декартовых координат (x, y, z). Такое пространство задается относительно поверхности объекта и при его движении перемещается вместе с ним.

плавающая запятая [плавающая точка (floating point)] - *числа с плавающей запятой* служат в вычислительной технике для **хранения** значений с несколькими десятичными разрядами после запятой. Данное понятие отражает тот факт, что запятая (точка) может менять свое положение.

платформа (platform) - конкретная конфигурация компьютера. Включает в себя операционную систему и другие специфичные компоненты (процессор и т. д.). Примерами платформ являются Irix, Linux и Windows.

повторное выполнение (redo) - после того как команда отменена, ее действие можно произвести еще раз путем повторного **выполнения**.

подключаемый модуль (plugin) - программа, встраиваемая в другое приложение. Такая программа **подключается** к некоторому приложению. Подключаемые модули часто обеспечивают дополнительные функции, которые отсутствуют в приложении.

подключение (plug) - идентифицирует атрибут конкретного узла. Служит для доступа к значениям атрибутов заданного узла.

полиморфизм (polymorphism) - в контексте такого объектно-ориентированного языка программирования, как C++ - способность объекта по-разному вести себя в зависимости от того, какой тип он имеет. Данная способность составляет основу мощного механизма расширения объектов.

• **потомок** (child) — нечто, имеющее родителя.

присваивание (assignment) - заключается в сохранении некоторого значения в той или иной переменной. Чтобы сохранить значение, используется оператор присваивания (=), например, \$a = 2.

приоритет (precedence) - операторов - определяет порядок вычислений, принятый в языке программирования. Оператор с более высоким приоритетом вычисляется прежде, нежели другой оператор с более низким приоритетом. К примеру, приоритет умножения выше приоритета сложения.

пространство (space) - конкретная система координат объекта. В частности, пространство определяет те преобразования, которые выполняются над объектом при его погружении в некоторую систему координат.

пространство имен (namespace) - область, где пребывает множество имен программы. Все имена находятся в одном и том же пространстве, поэтому они должны отличаться. При наличии повторений возникает конфликт в пространстве имен.

процедура (procedure) - средство описания отдельной операции на языке MEL, служит для выполнения операции и во многих случаях возвращает результат. Концептуально процедура аналогична функции языка C.

процедурная анимация (procedural animation) - анимация с программным управлением.

пружина (spring) - позволяет описывать и рассчитывать упругость, массу, торможение, а также другие явления и процессы, происходящие с участием двух точек.

прямая кинематика (forward kinematics/FK) - тот случай, когда аниматор должен явно задать ориентацию всех суставов.

псевдокод (pseudocode) - метод краткой записи, применяемый для описания компьютерных программ. Специальный синтаксис компьютерных языков заменяет собой реальные языки программирования.

няется более универсальными формулировками. Использование псевдокода упрощает понимание общего принципа работы программы людьми, не являющимися профессиональными программистами,

«пустой» указатель (void pointer) - базовый указатель, который может содержать адрес объекта любого типа.

путь по ОАГ (DAG path) - полный путь к данному узлу. Включает данный узел и всех его предков.

распространение бита изменений (dirty bit propagation) - процесс, в ходе которого сообщение, содержащее бит изменений, передается по графу зависимости от одного узла к другому. В конечном итоге, это сообщение пройдет путь от первого атрибута до всех других атрибутов, на которые тот влияет.

режим запроса (query mode) - режим выполнения команды при запросе параметров.

режим правки (edit mode) - режим выполнения команды при изменении параметров.

режим создания (creation mode) - режим выполнения команды при создании объекта или узла.

родитель (parent) - нечто, имеющее одного или нескольких потомков. Так как у родителя тоже может быть предок, его потомки могут иметь непрямых (косвенных) родителей. К их числу относятся предки прямых родителей - «деды», «прадеды» и т. д.

родитель по умолчанию (default parent) - при отсутствии явно указанных элементов интерфейса данный элемент будет связан с родителем по умолчанию. Родители по умолчанию имеются почти среди всех типов элементов.

родительский атрибут (parent attribute) - в составном атрибуте или атрибуте-массиве это атрибут самого верхнего уровня. Является родителем всех дочерних атрибутов нижних уровней.

«рукоятка» (handle) - См., описатель, «рукоятка».

сестринский узел (sibling) - потомок, имеющий того же родителя.

сетка (mesh) - ряд сгруппированных многоугольников, которые образуют поверхность.

скалярное произведение (dot product) - результат простого перемножения всех пар компонентов двух векторов и сложения полученных произведений: скалярное произведение $(a, b) = a.x * b.x + a.y * b.y + a.z * b.z$. Скалярное произведение часто применяется для расчета косинуса угла между двумя векторами.

скелет (skeleton) - иерархия суставов, определяющая внутреннюю структуру персонажа.

случайное число (random number) – число, значение которого полностью определяется **случаем**.

соединение (connection) – если значение одного атрибута поступает на вход другого, **между ними** устанавливается соединение. Соединения можно свободно устанавливать и разрывать.

составной атрибут (compound attribute) – атрибут, состоящий из **других** атрибутов, которые объединены в иной, более сложный атрибут.

структурное программирование (structured programming) – подход к проектированию, в соответствии с которым сложные системы разбиваются на мелкие, более простые в управлении фрагменты.

сустав (joint) — суставы напоминают кости. Они могут объединяться и образовывать прилатки. Нередко именно суставы перемещаются аниматорами, которые занимаются управлением персонажами.

сцена (scene) – состоит из всех данных Maya. В нее входят все модели, их анимация, эффекты, настройки и т. д.

сценарий (script) – текстовый файл, содержащий операторы языка **MEL**.

цепление (concatenation) – процесс построения цепочки из одного или нескольких элементов.

тип данных (data type) – определяет разновидность информации, которую может содержать переменная. Примерами типов данных являются **string**, **int** и **float**.

точка (point) – точка описывает местоположение. Для задания точек в среде Maya служит декартова система координат: *x*, *y*, *z*,

точка входа (entry point) – функция, которая вызывается при загрузке динамически подключаемой библиотеки в память.

точка выхода (exit point) - функция, которая вызывается при выгрузке динамически подключаемой библиотеки из памяти.

транслятор (translator) - в пакете Maya - программный модуль, способный переводить данные из одного формата в другой формат, которые поддерживает Maya. К примеру, транслятор может взять файл Softimage и преобразовать его в файл Maya. О трансляторах часто говорят как о средствах импорта и экспорта, поскольку они могут передавать информацию в систему Maya и из нее.

узел (node) - основополагающий строительный блок графа зависимости Maya. Узлы содержат атрибуты, которые могут изменяться пользователем. Кроме того, в них входит функция compute, автоматически вычисляющая некоторые атрибуты.

узел **curveAnim** (curveAnim node) - узел анимационной кривой. Содержит кривую анимации, которую можно редактировать и изменять с целью анимации конкретного параметра. Способен выполнять обычную анимацию по ключевым кадрам и анимацию с управляемыми ключами.

узел времени (time node) - узел time используется для хранения времени. Имеет один атрибут **outTime**, где содержится время. Помимо того, что текущее время хранится в узле **time1**, вы можете создавать дополнительные узлы времени.

узел доводки (tweak node) - применяется Maya для хранения всех независимых передвижений точек некоторого геометрического объекта.

узел зависимости (dependency node) - узел графа зависимости. Такими узлами являются все узлы Maya.

узел преобразования (transform node) - узел ОАГ, который предназначен для указания положения, ориентации и размера формы.

узел формы (shape node) - узел, содержащий такую форму, как полигональная сетка, кривая, **NURBS**-поверхность, или множество частиц.

управляемый ключ (driven key) - в то время как в анимации ключевой кадр определяется своим положением на временной оси, управляемый ключ позволяет задать ключевой кадр относительно некоторого атрибута.

усекать (truncate) - усечение происходит в тех случаях, когда нечто большее лишают его части ради совместимости с чем-то меньшим. Вещественные числа часто усекают, т. е. округляют, до целых путем отбрасывания десятичных разрядов справа от запятой.

фильтр (filter) — при работе с большим множеством элементов описывает множество ограниченного размера.

флаг (flag) - используется для указания одного из двух возможных состояний чего-либо («да» - «нет»). Служит как средство сигнализации.

флаг распространения (propagation flag) - флаг, который определяет, будет ли сообщение, содержащее бит изменений, пересыпаться выходным соединением.

форма (shape) - общее понятие, используемое в отношении всех трехмерных данных, которые может отобразить Maya. К формам относятся кривые, поверхности и точки.

функции (functors) - класс, реализующий множество функций, но не предоставляющий собственных данных.

функция (function) - в языках программирования C и C++ функция определяет способ выполнения отдельной операции. Функция может, к примеру, сложить два числа, или повернуть объект, или сделать что-то еще. Для выполнения функций их *вызывают*.

функция compute (compute function) - функция узла, вычисляющая его выходные атрибуты. Функция compute принимает входные атрибуты узла и рассчитывает окончательные значения выходных.

цикл (loop) - будучи помещена в цикл, операция повторяется несколько раз.

частицы (particles) — задают точки в пространстве. Часто для анимации и управления частицами к ним прикладывается сила либо на них влияет иное физическое явление.

«черный ящик» (black box) - о «черном ящике» говорят в том случае, когда точный способ функционирования данной системы неизвестен за ее пределами. Другими словами, снаружи не видно того, что делается внутри.

чтение в память (sourcing) - процесс, в ходе которого выполняется считывание сценария на языке MEL и его выполнение в среде Maya.

чувствительный к регистру (case-sensitive) - если операция чувствительна к регистру, то существует различие между двумя именами, состоящими из букв разной высоты. Так, к примеру, в системе, чувствительной к регистру символов, имена **bill** и **Bill** будут считаться различными.

шейдер (shader) - определяет окончательные свойства поверхности объекта. С его помощью можно, например, задать цвет, отражательную способность и прозрачность поверхности.

шум (noise) - псевдослучайное число. При постоянном входном воздействии шум порождает статистически постоянные, хотя и случайные числа.

экземпляр (instance) объекта - его точная копия. По сути, экземпляр представляет собой объект, имеющий те же **свойства**, что и оригинал. Он всегда сохраняет свойства оригинала независимо от того, каким изменениям оригинал подвергается.

элемент размещения (layout element) - **заполнитель**, на место которого будут помещены другие элементы. Схема размещения определяет окончательное расположение и задает размеры прочих элементов, которые будут к ней присоединяться.

язык сценариев (script language) - отличается от **других** типичных языков тем, что обычно он более прост в изучении и применении, а также не нуждается в компиляции. Код на таком языке интерпретируется во время **выполнения**, поэтому инструкции можно выполнять сразу после того, как они составлены.

Предметно-именной указатель

- (знак «минус»), оператор вычитания, 95
- (оператор декремента), 123
- > (черточка, угловая скобка), стрелка, в составе путей к узлам, 50
- ! (восклицательный знак), оператор «не», 103
- != (восклицательный знак, знак равенства), оператор «не равно», 100
- % (знак процента), взятие остатка, 97
- && (амперсанды), оператор «и», 103
- * (звездочка), умножение, 96
- . (точка)
 - оператор доступа к членам данных, 273
 - оператор «точка», 286
- / (левая косая), оператор деления, 96
- /*...*/ (левая косая, звездочка...звездочка, правая косая), ограничители многострочного комментария, 94
- // (левые косые), ограничители комментария, 93
- ; (точка с запятой), разделитель команд, 73
- ? (вопросительный знак, двоеточие), условный оператор, 123
- [] (квадратные скобки), описание массивов, 86
- ^ (знак карата), оператор векторного умножения, 97
- ` (обратная кавычка)
 - выполнение команд, 119
 - «горячая клавиша» MEL, 71
- { } (фигурные скобки), ограничители блока кода в языке MEL, 112
- | (вертикальная линия), в составе путей к узлам, 47
- || (вертикальные линии), оператор «или», 102
- + (знак «плюс»), оператор сложения, 95-96
- ++ (оператор инкремента), 123
- < (угловая скобка), оператор «меньше», 100
- <= (угловая скобка, знак равенства), оператор «меньше или равно», 100
- = (знак равенства), оператор присваивания, 95
- == (знаки равенства), оператор сравнения, 100
- > (угловая скобка), оператор «больше», 100
- >= (угловая скобка, знак равенства), оператор «больше или равно», 100
- 2-мерное параметрическое пространство, 47-50

A

Alias | Wavefront, 474

ANSI, 483

API, 483

ASCII, 483

B

boolean, 483

C

C++ API

- блоки данных, 467-470
- генераторы, 292
- заместители, 464
- инструменты/контексты, 290
- контексты графа зависимости, 466-467
- несетевые подключения, 465
- обзор, 14, 483
- обработка сквозного прохода, 471-472
- обращение к узлам, 463
- поддержка аргументов, 334
- поля, 291
- префикс MPx, 464
- примеры режима создания, 353-362
- решатели, 292
- сетевые подключения, 465
- справка, автоматическая, 339-340
- трансляторы файлов, 290
- узлы графа зависимости, нестандартные, 290

- флаги распространения, 470
 формы, 292
- C++ API, деформаторы**
 вспомогательные инструменты, 459
 вспомогательный локатор как «рукотка», 459
 изменения в графе зависимости, 457–459
 команда `deformer`, 458
 определение, 290, 486
 пример модуля `SwirlDeformer`, 451–457
 пример модуля `SwirlDeformer2`, 459–463
- C++ API, команды. См. также команды.**
MDGModifier, 352
 действия, 349
 информация о команде, 339–340
 нестандартные, 289
 обратимые команды, 349
 отмена, 342–343, 347–352
 очередь отмены, 343–347
 повторное выполнение, 342–343, 347–352, 492
 поддержка аргументов, 334
 примеры режима запроса, 353–362
 примеры режима правки, 353–362
 примеры режима создания, 353–362
 режим запроса, 353–354
 создание, 327–338
 справка, автоматическая, 339–340
 справка, создание собственной помощи, 340–342
 функция `isUndoable`, 362
- C++ API, локаторы**
 инструмент `Arc Length`, 428
 инструмент `Distance`, 428
 инструмент `Parameter`, 428
 определение, 291
 создание элементарного локатора, пример, 429–437
 функция `boundingBox`, 428, 435
 функция `color`, 429
 функция `colorRGB`, 429
 функция `draw`, 428, 433–435
 функция `getCirclePoint`, 435
 функция `isBounded`, 428, 435
- C++ API, манипуляторы**
- базовые манипуляции, 440
 определение, 291, 490
 потомки основного узла манипулятора, 440
 пример модуля `BasicLocator2`, 438–450
- C++ API, понятия**
MObject, 301–303
 абстрактный уровень, запрет доступа к ядру `Maya`, 293–294
 классический подход к проектированию классов, 295–296
 наборы функций `MFn`, 303–305
 нестандартные наборы функций, 305
 подход `Maya` к проектированию, 298–301
 полиморфизм, 296–298
 проектирование функций, 299
 «пустой» указатель в составе `MObject` (`void *`), 302
 соглашение об именах классов, 295
- C++ API, разработка подключаемых модулей**
MStatus, 318
 вывод сообщений, 321–322
 загрузка и выгрузка, 324–325
 инициализация и деинициализация, 314–317
 интеграция, 323–324
 информация на Web-сайте, 307
 обновления при выходе последней версии, 326
 отказ от вывода ошибок и предупреждений во всплывающих окнах, 322
 отображение предупреждающих сообщений, 321
 проверка на наличие ошибок, 318
 размещение, 326
 расширение `.mll`, 306
 расширение `.so`, 306
 функция `errorString()`, 321
 функция `MGlobals::displayError()`, 321
 функция `MGlobals::displayWarning()`, 321
 функция `name()`, 318–321
 функция `parent()`, 321

- C++ API, узлы.** Си. *также* узлы.
 MArrayDataHandle, 423–425
 MDATAHandle, 423–425
 атрибуты, описанные в классе
 MFnAttribute, 404
 динамические атрибуты, 411–412
 значения по умолчанию, 407
 команда melt, 378
 команды setAttr и getAttr, 412–413
 навигация
 по составным атрибутам, 416
 подключения, используемые для
 доступа к данным узла, 414–415
 подключения-массивы, 417–422
 руководство по проектированию,
 контекст узла, 426–427
 руководство по проектированию,
 локальность узла, 427
 руководство по проектированию,
 простота, 426
 руководство по проектированию, сети
 узлов, 428
 свойства атрибутов, 408–411
 свойство cached, 411
 свойство connectable, 409
 создание атрибутов, 405
 составные атрибуты, 406–407
 составные подключения, 416
 тени объекта, 391
 узел melt, 378
 флаг readable, 408
 функция addAttribute класса
 MFnDependencyNode, 412
 функция child класса MDATAHandle, 425
 функция compute, 412–414
 функция findPlug, 414
 функция getValue, 415
 функция inputValue, MDATABlock, 423
 функция isArray, 411
 функция isKeyable, 410
 функция isReadable, 408
 функция outputValue, MDATABlock, 423
 функция redoIt, 399
 функция setArray, 411
 функция setKeyable, 410
 функция setReadable, 408
 функция setValue, 415
 функция undoIt, 399
 функция viewFrame, 416
- CircleSweepManip**, 440
- Command Input Panel**, 71
- CommandLine**, 70–72, 74
- Command Shell**, 74
- CurveSegmentManip**, 440
- D**
- DG.** См. граф зависимости.
- DirectionManip**, 440
- DiscManip**, 440
- DistanceManip**, 440
- double**, 483
- E**
- ELF**, система расширенных слоев, 211, 483
- Expression.** См. *также* анимация.
 . (точка), оператор доступа к членам
 данных, 273
 автоматическое удаление, 274–275
 атрибуты в соединениях, 274
 в сравнении с выражениями, 261
 «висячие» узлы, 275
 времени создания, вычисление, 283
 определение, 261, 484
 отказ от setAttr и getAttr, 273
 отладка, 275–278
 отмена команд, 272
 поддерживаемые типы атрибутов, 272–273
 пустой атрибут, 275
 сочетание с анимацией по ключевым
 кадрам, 286–288
 узел выражения, 274–275
 частица, 278–286
 эффект вспышки, 265–267
 эффект магнита, 268–271
 эффект прыгающего объекта, 261–265
- Expression**, для частиц
 . (точка) оператор «точка», 286
 вычисление Expression времени
 выполнения, 283

- вычисление Expression времени создания, 283
 команда `asphrand`, 279
 компоненты векторов, 285–286
 управление, пример, 278–281
 уровня объекта, 283–285
 уровня частицы, 283–285
 функция `sphrand`, 279
- F**
`float` (действительные числа), 82
`FreePointManip`, 440
- H**
`Highend3D`, 474
`Hypergraph`, 26, 42
- I**
`int`, 484
- M**
`MArrayDataHandle`, 423–425
`Maya Embedded Language`. См., язык MEL.
`MAYA_SCRIPTS_PATH`, 130
`MDataBlock`, 423–425
`MDGModifier`
`MEL Command Reference`, 78
`MObject`, 301–303
`MStatus`, 318
- N**
`NURBS`, 484
`NURBS`-поверхность, 48–49
- P**
`PointOnCurveManip`, 440
`PointOnSurfaceManip`, 440
- S**
`Script Editor`, 71, 74, 119
`Shelf`, 72, 75
`sphereVolume`, 34
`StateManip`, 440
`string`, 484
- T**
`ToggleManip`, 440
- U**
`Underworld`, 47, 484
- V**
`(void *)` «пустой» указатель в составе `MObject`, 302
- W**
Web-сайт книги, 473
- A**
автоматизация, обзор, 12
автоматическое описание глобальных процедур, 119
амперсанды (`&&`), оператор «и», 103
анимация. Си. также Expression.
воспроизведение, 165–167
время, 162
единицы времени, 163–164
команда `currentUnit`, 163
по ключевым кадрам, 286–288, 484
траектории движения, 207–210
анимация, кривые. Си. также узел анимационной кривой.
буфер обмена для ключевых кадров, 182
касательные, 178–182
ключевые кадры, 173–178
ключи разбиения, 178
команда `copyKey`, 182
команда `cutKey`, 182
команда `getAttr`, 172
команда `isAnimCurve`, 171
команда `keyframe`, 171
команда `listAnimatable`, 171
команда `pasteKey`, 182
команда `setInfinity`, 173
команда `snapKey`, 177
обычные ключи
и ключи разбиения, 178
от первого ключевого кадра до бесконечности, 172

- от последнего ключевого кадра до бесконечности, 172
- процедура `printAnim`, 182–186
- процедура `printTangentPositions`, 186–190
- редактирование, примеры, 182–190
- создание, примеры, 182–190
- управляемый ключ, 170
- установка ключевых кадров, 170–172
- функции, 168–170
- анимация, скелеты**
- запрос параметров вращения суставов, 192
 - инверсно-кинематические (ИК)
 - описатели, 193
 - команда `joint`, 190
 - команда `ListRelatives`, 199–200
 - команда `removeJoint`, 192
 - команда `xform`, 192
 - образование покрова, 190
 - описание, 190–207
 - прорисовка оболочек, 190
 - процедура `copySkeletonMotion`, 201–207
 - редактирование
 - положения суставов, 191–192
 - сценарий `outputJoints`, 193–196
 - сценарий `ScaleSkeleton`, 199–200
- аргумент, 484
- арифметические операторы, 95
- атрибут уровня объекта, 283–285, 485
- атрибут уровня частицы, 283–285, 485
- атрибут-потомок, 30
- атрибуты. См. *также* свойства.
- динамические, 160–161
 - имена, 30
 - команда `attributeQuery`, 162
 - команда `deleteAttr`, 161
 - команда `getAttr`, 158
 - команда `renameAttr`, 161
 - массивы, 31
 - определение наличия в узле, 160
 - определение, 28, 485
 - организация, 30
 - поддерживаемые типы, 272–273
- получение информации, 162
- потомки, 30
- простые, 30
- родительские, 30
- составные, 31
- типы данных, 30
- узлы, перечень, 161
- уровня объекта, 283–285, 485
- уровня частицы, 283–285, 485
- функция, 34–35
- атрибуты, C++**
- динамические, 411–412
 - значения по умолчанию, 407–408
 - описание, 405
 - свойства, 408–411
 - свойство `array`, 411
 - свойство `cached`, 411
 - свойство `connectable`, 409
 - свойство `keyable`, 410
 - свойство `readable`, 408
 - свойство `storable`, 410
 - свойство `writable`, 408
 - создание, 405
 - составные, 406–407
- аффинное преобразование, 153, 485
- Б**
- базовое ядро, C++ API, 293–294
- базовые манипуляции, 440
- базовый вариант узла, 29
- бесконечный цикл, 108
- библиотека, 485
- бит изменений, 485
- блоки данных, 467–470
- «больше» (>), 100
- «больше или равно» (>=), 100
- Брайан Эверт (Bryan Ewert), 474
- буфер обмена для ключевых кадров, 182
- В**
- векторное произведение, 485
- векторы, 485
- вещественных значений, 85
 - для хранения точек и направлений, 85

- вертикальная линия (|), в составе путей кузлам, 47
- вертикальные линии (||), оператор «или», 102
- «висячие» узлы, 275, 485
- вложенные комментарии, 94
- вложенные операторы if, 100
- возврат результата, 112
- вопросительный знак, двоеточие (?), условный оператор, 123
- восклицательный знак, знак равенства (!=), оператор «не равно», 100
- восклицательный знак, оператор «не», 103
- воспроизведение
- анимация, 165–166
 - диапазон анимации, 166–167
 - диапазон воспроизведения, 166–167
 - диапазон, 166–167
 - команда `playblast`, 167
- вращать/вращение, 485
- время жизни переменных, 113
- вспомогательные инструменты для деформаторов, 459
- входной атрибут, 485
- входящая касательная, 485
- выбор интерфейса программирования, 15–18
- вывод
- значений переменных, 136
 - сообщений
 - в подключаемых модулях, 321–322
 - сообщения об ошибке
 - в текущий поток `stderr`, 321
 - строк в стандартный выходной поток, 137
- выделение (освобождение) памяти, 15
- выполнение подключаемых модулей при разработке под Windows, 309–310
- выполнить трансляцию/трансляция, 486
- выпуск подключаемых модулей при разработке под Windows, 313
- выражение времени создания, 486
- выражения
- в сравнении с Expression, 261
 - времени выполнения, 283
 - процедурная анимация, 261
- выход из цикла, 109
- выходной атрибут, 486
- вычисление. См. также функция compute.
- Expression времени выполнения, 283
 - выходных атрибутов
 - на основе входных, 412
 - выходы, 36
- Г**
- генераторы, 292
- генерация. Си. создание.
- главный узел-манипулятор, потомки, 440
- глобальная область видимости, 486
- глобальные переменные, 114
- глобальные процедуры, 118
- граф зависимости
- `Hypergraph`, 26
 - блоки данных, 467–470
 - двунаправленная модель, 24
 - изменения, 457–459
 - конвейеры, 23
 - контексты, 466–467
 - модель потока данных, 23–25, 27–28
 - нестандартные узлы, 290
 - обновление, 53–66
 - определение, 483
 - отображение, 26–27
 - подход `Maya`
 - подходы из прошлого, 21–25
 - разработка
 - 3-мерных приложений, 21–25
 - сцена, 25–26
 - узлы, 24, 28–29
 - анимационной кривой, 28–29
 - времени, 28–29
 - преобразования, 28–29
- функция compute, 29
- графический интерфейс пользователя. См. интерфейс GUI.
- группирующие возможности, 47
- группы, 245, 486

Д

дву направленная модель, 24, 486
 деинициализация подключаемого модуля в C++ API
 пример, модуль `helloworld2`, 314–317
 точка входа, 314
 • точка выхода, 314
 действие, 486
 декартовы координаты, 486
 дерево, 486
 деформаторы, C++ API
 вспомогательные инструменты, 459
 вспомогательный локатор как «рукотка», 459
 изменения в графе зависимости, 457–459
 команда `deformer`, 458
 определение, 290, 486
 пример модуля `SwirlDeformer`, 451–457
 пример модуля `SwirlDeformer2`, 459–463
 динамически подключаемая библиотека, 486
 динамические атрибуты, 160–161, 411–412, 487
 доступность переменных, 114
 дочерний атрибут. См. атрибут-потомок,

Е

единичная матрица, 487
З
 зависимый атрибут, 487
 загрузка и выгрузка подключаемых модулей, 324–325
 задание имен атрибутов, 160–161
 законы пружины, 487
 заместители, 464
 запрос параметров вращения суставов, 192
 запрос текущих настроек бесконечности, 172–173
 защита информации, 18
 звездочка (*), оператор умножения, 96
 знак «минус» (-), оператор вычитания, 95
 знак «плюс» (+), оператор сложения, 95–96

знак карата (^), оператор векторного произведения, 97
 знак процента (%), взятие остатка, 97
 знак равенства (=), оператор присваивания, 95
 знаки равенства (==), оператор сравнения, 100
 значение инициализации, 487
 значок C, 81
 значок E, 81
 значок M, 81
 значок Q, 81

И

иерархия
 изменение порядка сестринских элементов, 147
 команда `group`, 146–147
 команда `List Relatives`, 147
 команда `reorder`, 147
 команда `ungroup`, 148
 образование покрова (прорисовка оболочек), 190
 определение, 487
 преобразований, 45, 487
 прорисовка оболочек (образование покрова), 190
 просмотр списка
 дочерних элементов узла, 147
 «родитель - потомок», 42
 «сверху – вниз», 43
 создание узлов преобразования, 147
 список прямых и косвенных потомков, 147
 список форм-потомков узла, 147
 суставов, 191
 человекоподобный робот, 45
 извлечение. См. считывание.
 изменение. См. редактирование.
 изображение локаторов нестандартны ч цветом, 429
 изображения, 246–248
 ИК-решатель, 488
 имена атрибутов, 30
 инверсная кинематика (ИК), 193, 488
 инициализация
 генератор случайных чисел, 279

- модуля в C++ API, 314–317
 пример: модуль `helloWorld2`, 314–317
 точка входа модуля, 314
 точка выхода модуля, 314
- инструмент**, 488
 Arc Length, 428
 Distance, 428
 Parameter, 428
- инструменты/контексты**, 290
- интеграция**
 модулей, 323–324
 обзор, 12
- интерпретируемые языки**, 14, 488
- интерфейс GUI**
 группы, 245
 изображения, 246–248
 кнопка разворачивания и свертывания, 218
 кнопки, 237
 команда
`attrColorSliderGrp`, 254
 команда `attrFieldGrp`, 253
 команда `checkBox`, 239
 команда
`colorSlideButtonGrp`, 245
 команда `columnLayout`, 225
 команда
`floatSlideButtonGrp`, 245
 команда `formLayout`, 227–229
 команда `frameLayout`, 230
 команда `gridLayout`, 226–227
 команда
`hardwareRenderPanel`, 248
 команда `hyperPanel`, 248
 команда `iconTextButton`, 238
 команда `iconTextCheckBox`, 239
 команда `iconTextRadioButton`, 241
 команда `image`, 247
 команда `layout`, 224
 команда `menuBarLayout`, 232–237
 команда `menuItem`, 236
 команда `modelPanel`, 248
 команда `nodeOutliner`, 248
 команда `outlinerPanel`, 248
 команда `picture`, 246
- команда
`progressWindow`, 258–260
 команда `promptDialog`, 224
 команда `radioCollection`, 240
 команда
`radioMenuItemCollection`, 236
 команда `rowLayout`, 225–226
 команда `scrollField`, 242
 команда `setParent`, 237
 команда `sphere`, 252
 команда `symbolButton`, 238
 команда `symbolCheckBox`, 239
 команда `tab Layout`, 230–231
 команда `text`, 242
 команда `textField`, 241
 команда `textFieldButtonGrp`, 245
 команда `textScrollList`, 243
 команда `waitCursor`, 258
 команда `window`, 213
 меню, 234–237
 модальное диалоговое окно, 224
 обратная связь с пользователем, 256–260
 окна, 218
 определение, 484
 панели, 248
 переключатели, 240–241
 процедура `showMyWindow()`, 218–224
 родитель по умолчанию, 215
 свойства окон, 218
 связывание элементов, 251–254
 система расширенных слоев (ELF), 211
 строка заголовка, 218
 схемы размещения, 224–233
 текст, 241–243
 флагки, 239
 элемент `helpLine`, 257
 элемент `tool Collection`, 249–250
 элемент размещения, 214
- интерфейс**, 33–34, 488
 исходящая касательная, 488
- К**
- кадрируемый**, 488
 касательные, 178–182, 488

- каталоги сценариев по умолчанию, 128–130
квадратные скобки ([]), описание массивов, 86
классы. См. *также* имена конкретных классов.
иерархия, 487
определение, 488
проектирование,
классический подход, 295–296
соглашение об именах, 295
ключевые кадры, 173–178
ключи разбиения, 178, 488
и обычные ключевые кадры, 178
ключи, управляемые множеством, 489
кнопка разворачивания, 218
кнопка свертывания, 218
кнопки, 237
команда
 getAttr, отказ от нее, 273–274
 help
 C++ API, 339–340
 MEL, 77–78
 menuBarLayout, 232–237. См. *также* интерфейс GUI.
 setAttr, отказ от нее, 273–274
 правки, 80
командный режим
 запроса, 81
 правки, 81
 создания, 81
команды. См. *также* C++ API, команды.
 about, 141
 attrColorSliderGrp, 254
 attrFieldGrp, 253
 attributeQuery, 162
 checkBox, 239
 colorSlideButtonGrp, 245
 columnLayout, 225
 copyKey, 182
 currentUnit, 163
 cutKey, 182
 deformer, 458
 delete, 144
 deleteAttr, 161
 error, 138
 eval, 120
 exists, 139
 floatSlideButtonGrp, 245
 formLayout, 227–229
 frameLayout, 230
 getAttr, 158, 172
 gridLayout, 226–227
 group, 146–147
 hardwareRenderPanel, 248
 hyperPanel, 248
 iconTextButton, 238
 iconTextCheckbox, 239
 iconTextRadioButton, 241
 image, 247
 isAnimCurve, 171
 joint, 190
 keyframe, 171
 layout, 224
 listAll(), 145–146
 listAnimatable, 171
 listRelatives, 147. 199–200
 ls, 144
 melt, 378
 menuBarLayout, 232–237. См. *также* интерфейс GUI.
 menuItem, 236
 modelPanel, 248
 nodeOutliner, 248
 objExists, 144
 outlinerPanel, 248
 pasteKey, 182
 picture, 246
 playblast, 167
 posts, 327
 print, 136
 progressWindow, 258–260
 promptDialog, 224
 radioCollection, 240
 radioMenuItemCollection, 236
 removeJoint, 192
 rename, 144

renameAttr, 161
reorder, 147
rowLayout, 225–226
scrollField, 242
setAttr, 158
setInfinity, 173
setParent, 237
snapKey, 177
sphere, 252
sphrand, 279
symbolButton, 238
symbolCheckbox, 239
tabLayout, 230–231
text, 242
textField, 241
textFieldButtonGrp, 245
textScrollList, 243
trace, 137
ungroup, 148
waitForCursor, 258
warning, 138
whatIs, 139
window, 213
xform, 192
 методы выполнения, 119
 необратимые, 349
 определение, 489
отмена, 272
 отменяемые, 349
 подсказка
 C++ API, 339–340
 MEL, 77–78
 проверка доступности, 139
 хранение команд как файлов, 73
 команды, запуск
 ; (точка с запятой), разделитель команд, 73
 Command Input Panel, 71
 Command Line, 70–72
 Command Shell, 74
 History Panel, 71
 Script Editor, 71, 74
 Shelf, 72
 «горячая клавиша», 71
 команда правки. 80
 расширение .mel, 73
 файлы сценариев, 75
комментарий
 /*...*/ (левая косая,
 звездочка...звездочка, правая косая),
 ограничители многострочного
 комментария, 94
 // (левые косые), ограничитель
 однострочного комментария, 93
 MEL, 93–94
 определение, 489
компиляция и сборка, 489
компонент, 489
компоненты векторов, 285–286
конвейер, 23, 489
контекст, 489
контроллер анимации, 490
копирование элементов анимации одного
 скелета в другой, 201–207
корень, 490
корневой узел, 47
косые
 / (левая косая), оператор деления, 96
 /*...*/ (левая косая,
 звездочка...звездочка, правая косая),
 ограничители многострочного
 комментария, 94
 // (левые косые), ограничители
 комментария, 93
курсор мыши – «песочные часы», 258
Л
 левая косая (/), оператор деления, 96
 левая косая, звездачка...звездачка, правая
 косая /*...*/, ограничители
 многострочного комментария, 94
 левые косые (//), ограничители
 комментария, 93
 логические операторы, 102–104
 логическое значение, 483
 локальная область видимости, 490
 локальное пространство, 490
 локальные **переменные**, 114, 116

локаторы, C++ API, 291, 428, 490

инструмент

Arc Length, 428

Distance, 428

Parameter, 428

определение, 490

создание элементарного локатора,

пример, 429–437

функция *boundingBox*, 428, 435

функция *color*, 429

функция *colorRGB*, 429

функция *draw*, 428, 433–435

функция *getCirclePoint*, 435

функция *isBounded*, 428, 435

M

манипуляторы, C++ API

базовые манипуляции, 440

определение, 291

потомки главного узла-манипулятора, 440

пример модуля *BasicLocator2*, 438–450

массивы

атрибутов, 31

описание, 86–88

определение, 490

масштаб, 490

матрицы, 88–89, 490

преобразования, 46, 149–150, 490

межплатформенная переносимость, 17

«меньше или равно» (\leq), 100

меню, 234–237. См. также интерфейс GUI.

мировое пространство, 150, 491

многострочные комментарии, 94

множество, 491

модальное диалоговое окно, 224

модальный, 491

модель потока данных, 23–25, 27–28, 491

моноширинный шрифт, 127

H

наборы функций, 305, 491

MFn, 303–305

нажатие клавиши, 491

настройка

обзор, 11–12

среды разработки, 131–133

начальное число, 491

необратимые команды, 349

несетевые подключения, 465

нормаль, 491

O

ОАГ

| (вертикальная линия), в составе путей

к узлам, 47

-> (стрелка), 50

Hypergraph, 42

Underworld, 47

возможности группировки, 47

иерархия «родитель – потомок», 42

иерархия «сверху – вниз», 43

иерархия узлов, 46

иерархия человекоподобного робота, 45

корневой узел, 47

определение, 44, 491

параметрическое пространство, 47–50

преобразование, 45–47

пути, 46, 494

родительские преобразования,

множественные, 46

связь с узлами DG, 42

узел преобразования, 43–47

узел **формы**, 43–47

узел-локатор категории shape, 44

экземпляры, 50–53

ОАГ, экземпляры

изменения, влияющие на все

экземпляры, 50

новый узел преобразования, 50–51

преимущества, 52–53

совместно используемый узел формы, 51

область видимости

{ }, ограничители блока кода в языке

MEL, 112

Script Editor, 119

автоматическое описание глобальных

процедур, 119

время жизни переменных, 113

- глобальная, 134
- доступность, 114
- локальная, 114, 116
- описание процедуры как **глобальной**, 118
- определение, 4911
- поиск глобальных процедур, 118
- пространство имен, 118
- 的独特性 имен как средство предотвращения конфликтов, 116–117
- явная инициализация глобальных переменных, 115
- обновление версий**, 143
- обновление графа зависимостей** бит изменений, 58–62
- дву направленная модель, 54–58
- обновление при анимации по ключевым кадрам**, 62–66
- подход на основе потока данных, 55
- распространение бита изменений, 60–62
- распространение обновлений, 61
- связанные узлы, 54
- флаг необходимости обновления, 55–58
- обновление при выходе последней версии, 326
- обработка сквозного прохода, 471–472
- образование покрова (прорисовка оболочки), 190, 491
- обратимые команды, 349
- обратная кавычка ` выполнение команд, 119
- «горячая клавиша» MEL, 71
- обратная связь с пользователем, 256–260
- объектное пространство, 492
- объекты
 - переименование, 278
 - свойства, считывание, 79
 - тени, 391
- объекты, MEL
 - атрибуты, 158–162
 - доступ к свойствам, 79
 - иерархии, 146–148
 - команда delete, 144
 - команда listAll(), 145–146
 - команда ls, 144
- команда objExists, 144
- команда rename, 144
- описание, 79
 - организация цикла, 145–146
 - отображение имен и типов, 145–146
 - проверка существования, 144
 - составление списка, 144
- обычные ключевые кадры и разбиения, 178
- однострочные комментарии, 93
- однострочный ограничитель, 93
- окна
 - добавление панели, 248
 - описание, 218
- онлайновые ресурсы
 - Alias[®] Wavefront, 474
 - Ewert 474
 - Hipheend3D 474
 - Web-сайт книги, 473
- операторы
 - (знак «минус»), оператор вычитания, 95
 - (оператор декремента), 123
 - % (знак процента), взятие остатка, 97
 - * (звездочка) умножение, 96
 - / (левая косая), оператор деления, 96
 - ? : (вопросительный знак, двоеточие), условный оператор, 123
 - + (знак «плос»), оператор сложения, 95–96
 - ++ (оператор инкремента), 123
 - else if, 100
 - else, 99
 - for-in, 126
 - if (..), 99
 - switch, 124–126
- арифметические, 95–97
- группировка (), 105–106
- декремента (…), 123
- логические, 102–104
- логические значения, 97–98
- определение, 492
 - отношений, 99–100
 - подставляемые, 123
 - приоритет, 104–106
 - присваиваний (=), 95
- описатель, 492
- определение

версии Maya, 141
диапазона и *скорости воспроизведения*, 166
текущего *времени*, 163
типа переменной, 139
типа процедуры, 140
типа *схемы размещения*, 224
типов переменных, 139
организация цикла
по элементам, 126, 145–146
ориентированный ациклический граф.
См. ОАГ
оси, 492
от первого ключевого кадра
до бесконечности, 172, 492
от последнего ключевого кадра
до бесконечности, 172, 492
отключение *Expressions* при отладке, 277
отладка, *Expressions*
 отключение, 277
 переименование объектов, 278
 проверка на наличие ошибок, 275
 разорванные соединения, 277–2
 трассировка, 276–277
отладка, *разработка подключаемых модулей*, 311–313
отладка, сценарии
 MAYA_SCRIPTS_PATH, 130
 автоматический запуск сценариев, 131
 каталоги сценариев по умолчанию, 128–130
 команда *about*, 141
 команда *error*, 138
 команда *exists*, 139
 команда *print*, 136
 команда *trace*, 137
 команда *warning*, 138
 команда *whatIs*, 139
 моноширинные шрифты, 127
 обновление версий, 143
 определение *типов*, 139–140
 отображение номеров строк, 135–136
 отображение предупреждений и ошибок, 138–139
подготовка к разработке,
пользовательская *среда*, 131–133
поддержка версий, 141
пользовательские
каталоги сценариев, 130–131
проверка во время работы, 139
размещение, 142–143
расширение имени файла *.mel*, 128
сохранение перед выполнением, 134
текстовый редактор, 127
файл *userSetup.mel*, 131
чтение в память, 133–134
отмена
 класс *MPxCommand*, 347–349
 иды *меню*, 342–343
 команды *Expression*, 272
 определение, 492
 пример, 349–352
отношение «родитель – потомок», 42
отображение
 графа зависимости, 26–27
 единственной строки неизменяемого текста, 242
 единственной строки редактируемого текста, 241
 значения параметра точки, 428
 кнопки *навигации*, автоматическое, 254
набора переключателей
 со значками, 241
 нескольких строк
 редактируемого текста, 242
номеров строк, 135–136
полос прокрутки, 232
предупреждений и ошибок, 138–139
расстояния вдоль кривой, 428
расстояния между
двумя локаторами, 428
списка текстовых элементов, 243
статического
растрового изображения, 246–248
отыскание
 анимируемых узлов, 171
 атрибутов узла, 160
 анимационной кривой, 171
 отмены, 343–347

П

панели

- внедрение в состав окон, 248
- истории команд, 71
- команда
 - `hardwareRenderPanel`, 248
 - команда `hyperPanel`, 248
 - команда `modelPanel`, 248
 - команда `nodeOutliner`, 248
 - команда `outlinerPanel`, 248

панель истории команд, 71

параметрическое пространство

- `NURBS`-поверхность, 48–49
- двухмерное, 47–50
- определение, 492

параметры при флагах, 78

переименование атрибутов, 161

переключатели, 240–241

переменные

- автоматическое определение типов переменных, 89–91
- автоматическое преобразование типов, 91–93
- векторы (вещественных значений), 85
- векторы для хранения точек и направлений, 85
- вещественные (действительные) числа, 82
- время жизни, 113
- значения хранимых данных, 81
- массивы, 86–89
- матрицы, 88–89
- строки (текстовых символов), 83–84
- типы, 82–89

плавающая запятая, 492

платформа, 492

повторное выполнение

- класс `MPxCommand`, 347–349
- команды C++ API, 342–343
- команды Expression, 272
- определение, 492
- пример, 349–352

поддержка версий, 141

подключаемые модули, разработка под Windows

выполнение, 309–310

выпуск программного продукта, 313

отладка, 311–313

первые шаги, 308–309

редактирование, 310–311

подключаемые модули, разработка с

использованием C++ API

`MStatus`, 318

вывод сообщений, 321–322

загрузка и выгрузка, 324–325

инициализация

и деинициализация, 314–317

интеграция, 323–324

информация на Web-сайте, 307

обновления при выходе последней

версии, 326

отказ от вывода ошибок и предупреждений

во всплывающих окнах, 322

отображение предупреждающих

сообщений, 321

проверка на наличие ошибок, 318

размещение, 326

расширение `.rll`, 306расширение `.so`, 306функция `errorString()`, 321

функция

`MGlobal::displayError()`, 321

функция

`MGlobal::displayWarning()`, 321функция `name()`, 318–321функция `permor()`, 321

подключаемый модуль, 492

подключение, 493

подключения-массивы, 417–422

подменю, выход из них, 237

подставляемые

арифметические операторы, 123

подход Maya к проектированию, 298–301

в сравнении

с прежними подходами, 21–25

поиск, 79

глобальных процедур, 118

показ. См. [отображение](#).

полиморфизм, 296–298, 493

- полоса индикатора, 258
пользовательские каталоги сценариев, 130–131
помощь
 команды, C++ API, 339–342
 команды, MEL, 77–79
постфиксная форма, 123
поток узлов, 28
потомок, 493
предупреждения
 во всплывающих окнах, 322
 команды `setAttr` и `getAttr` внутри функции `compute`, 412–413
 передача нескольких атрибутов на один вход, 40
 процедуры внутри процедур, 112
преимущества Maya, 19
преобразование
 расчет окончательного положения точки локального пространства, 152–154
 точки из локального пространства в мировое, 150–151
преобразования, язык MEL. См. язык MEL, преобразования.
префикс MPx, 464
префиксная форма, 123
примеры
 `menuBarLayout`, 234–237. См. также интерфейс GUI.
 `sphereVolume`, 34
 базовый вариант узла, 29
 денициализация модуля в C++ API, 314–317
 матрицы составных атрибутов, 32
 модуль `BasicLocator2`, 438–450
 модуль `GoRolling`, 363–377
 модуль `groundShadow`, 391–404
 модуль `helloWorld2`, 314–317
 модуль `Melt`, 378–391
 модуль `posts1`, 328–333
 модуль `posts2`, 334–336
 модуль `posts3`, 336–338
 модуль `posts4`, 340–342
 модуль `posts5`, 353–362
 модуль `SwirlDeformer`, 451–457
модуль `SwirlDeformer2`, 459–463
написание команд и узлов, 391–404
написание простых узлов, 363–377
написание сложных узлов, 378–391
простая процедура, 110–112
простой узел, 30
режим правки в C++, 353–362
создание и редактирование кривых при анимации
по ключевым кадрам, 182–190
создание элементарного локатора, 429–437
составные атрибуты, 31
узел `boxMetrics`, 36
узел времени, 37
узел с составным атрибутом, 31
управление частицами, 278–281
примечания. См. [комментарии](#).
приоритет, 493
присваивание
 запись в цепочку, 122
 определение, 493
проблемы обновления узлов, 53
проверка
 во время работы, 139
 на наличие ошибок, 275. См. также отладка
 подключаемых модулей на наличие ошибок, 318
 существования объекта, 144
продолжение цикла, 108–109
проектирование
 классов, классический подход, 295–296
 функций, 299
производительность
 интерфейсов программирования, 18
 языка MEL, 14
прорисовка оболочек
(образование покрова), 190
простой атрибут, 30
простой узел, 30
простота применения, 16
пространства. 150–152, 493
пространство имен, 118, 493

- простые типы данных, 30
 процедурная анимация, 261, 493
 процедуры
 copySkeletonMotion, 201–207
 getInstanceIndex, 154–158
 objToWorld, 152–154
 printAnim, 182–186
 printTangentPositions, 186–190
 showMyWindow(), 218–224
 spaceToSpace, 154–158
 transformPoint, 152–154
 внутри процедур, 112
 возврат результата, 112
 глобальная область видимости, 117
 определение, 493
 пример, 110–112
 проверка доступности, 139
 структурное программирование, 109–110
 тип, определение, 140
 пружина, 493
 прямая кинематика, 493
 псевдокод, 16, 493
 пустой атрибут, 275
 «пустой» указатель, 494
 в составе MObject (void *), 302
 пути, 46, 494
 путь преобразования, 46
- P**
- разглашение информации, 18
 размещение
 подключаемых модулей, 326
 сценариев, 142–143
 разорванные соединения, 277–278
 разработка 3-мерных приложений, 21–25
 подход Maya в сравнении с прежними подходами, 21–25
 распространение бита изменений, 494
 расширение
 .mel, 73
 .mll, 306
 .so, 306
 групп, создание кнопок, 245
 файла .mel, 128
- расширения, обзор, 13
 редактирование
 именузлов, 37, 47, 171, 200, 205, 443
 исходного экземпляра, 50
 ключевых кадров, 174–178
 положения суставов, 191–192
 порядка сестринских элементов, 147
 при разработке модулей под Windows, 310–311
 родительских узлов, 393
 текущего времени, 416
- режим
 запроса, 353–354, 494
 команды по умолчанию, 81
 правки, 494
 создания, 494
- режимы команд, 81
- рендеринг, 486
- ресурсы системы Maya
 изучение C++ API, 475
 изучение языка MEL, 475
 примеры на языке MEL, 475
 примеры с использованием C++ API, 476
 справочник по C++ API, 475
 справочник по языку MEL, 475
- решатели, 292
- родитель, 494
 по умолчанию, 215, 494
- родительские преобразования, множественные, 46
- родительский атрибут, 30, 494
- руководство по проектированию узлов
 контекст, 425–427
 локальность узла, 427
 простота, 426
 сети узлов, 428
- «рукоятка», 492
- C**
- свертывание длинных операторов, 123
 свойства, C++. См. также атрибуты.
 array, 411
 cached, 411
 connectable, 409

- keyable, 410
- readable, 408
- storable, 410
- writable, 408
- атрибутов, 408–411
- объекта, считывание, 79
- окон, 218
- связанные атрибуты, 274
- связывание
 - атрибутов, 465
 - управляющих элементов, 251–254
- связь узлов ОАГ и DG, 42
- сестринский узел, 494
- сетевые подключения, 465
- сетевые узлы, 24, 38
- сетка, 494
- система расширенных слоев (ELF), 211
- скалярное произведение, 96, 495
- скелет, 495
- сложные типы данных, 30
- случайное число, 495
- совместно используемый экземпляр узла
- формы, 51
- соединение, 495
 - узлов, 38–41
- создание
 - атрибутов, 405–406
 - группы для хранения дочерних атрибутов масштабирования, 253
 - группы с бегунком для управления вещественным значением, 244
 - изменение Dependency Graph, 457
 - интерфейса с возможностью растяжения и сжатия, 230
 - ключевых кадров, 173–174
 - кнопок, 237–239
 - нового узла преобразования, 147
 - объектов, обладающих притяжением 268–271
 - окон, 211–212
 - переключателей, 240–241
 - подключения к атрибуту, 414–415
 - скелетон при помощи MEL, 190
 - случайных чисел, 279
- стандартных флагков, 239
- стоеч вдоль кривых, 327
- узла melt, внедрение в историю построения, 378
- флагков, 239
- элементов меню, работающих как кнопки, 236
- составные атрибуты
 - навигация, 416
 - описание, 406–407
 - определение, 31, 495
- составные подключения, 416
- сохранение
 - команд как файлов, 73
 - отчета о выполнении сценария, 137
 - перед выполнением, 134
 - сценариев, 128–130
 - сцены как распределенной сети, 25–26
- сочетание Expressions и анимации по ключевым кадрам, 286–288
- список
 - атрибутов узла, 161
 - дочерних суставов
 - корневого узла, 199–200
 - дочерних элементов узла, 147
 - кривых анимации, связанных с данным узлом, 171
 - объектов, 144
 - прямых и косвенных потомков, 147
 - текущих выделенных объектов, 144
 - форм-потомков узла, 147
- средства
 - импорта данных, 12
 - экспорта данных, 12
- ссылка на узел, 463
- стрелка (->), 50
- строка, 484
 - заголовка, 218
 - текстовых символов, 83–84
- структурное
- программирование, 109–110, 495
- сустав, 495
- схемы размещения, 224–233
- сцена

- определение, 495
 сохранение
 как распределенной сети, 25–26
 центральное хранилище данных доступа, 26
- сценарии**
 запускаемые автоматически, 131
 определение, 495
- сценарии, отладка и тестирование**
`MAYA_SCRIPTS_PATH`, 130
 автоматический запуск сценариев, 131
 каталоги сценариев
 по умолчанию, 128–130
 команда `about`, 141
 команда `errgor`, 138
 команда `exists`, 139
 команда `print`, 136
 команда `trace`, 137
 команда `warning`, 138
 команда `whatIs`, 139
 моноширинные шрифты, 128
 обновление версий, 143
 определение типов, 139–140
 отображение номеров строк, 135–136
 отображение предупреждений
 ошибках, 138–139
 подготовка к разработке,
 пользовательская среда, 131–133
 поддержка версий, 141
 пользовательские
 каталоги сценариев, 130–131
 проверка во время работы, 139
 размещение, 142–143
 расширение файла `.mel`, 127
 сохранение перед выполнением, 134
 считывание сценариев `MEL`, 133–134
 текстовый редактор, 128
 файл `userSetup.mel`, 131
 чтение в память, 133–134
- сценарий**
`outputJoints`, 193–196
`ScaleSkeleton`, 199–200
- сцепление**, 495
- считывание**
 данных, 415
 значений анимированных атрибутов, 172
- индекса экземпляра узла, 155
 информации об атрибутах, 162
 полного списка объектов, 144
 потомков составного атрибута, 425
 свойств объекта, 79
 списка анимационных кривых, 171
 списка атрибутов узла, 161
 списка прямых и косвенных потомков, 147
 списка текущих выделенных объектов, 144
 список форм-потомков узла, 147
 сценариев `MEL`, 133–134
 фактического количества частиц, 281
 флага `argray`, 411
 флага `keyable`, 410
 флага `readable`, 408
- T**
- Тейлор, Майк (Taylor, Mike), 11
текст, 241–243
 сообщения, соответствующий коду
 ошибки, 321
- текстовый редактор**, 127
- тени** объекта, 391
- тестирование** сценариев. См. отладка, сценарии,
- типы** данных
 атрибутов, 30
 определение, 495
 простые, 30
 сложные, 30
- точка**, 495
- точка** (.)
 оператор «точка», 286
 оператор доступа к членам данных, 273
- точка входа**, 495
- точка выхода**, 496
- точка** с запятой (;), разделитель команд, 73
- трансляторы**, 12, 496
 файлов, 290
- трассировка и обнаружение ошибок**, 276–277
- Y**
- уведомление пользователя
 о занятости компьютера, 258–260

- об ошибках и предупреждениях, 321–322
 угловая скобка (<), оператор «меньше», 100
 угловая скобка (>), оператор «больше», 100
 угловая скобка, знак равенства (<=),
 оператор «меньше или равно», 100
 угловая скобка, знак равенства (>=),
 оператор «больше или равно», 100
 удаление
 выражений Expressions, 274–275
 составов, 192
 узлов, 144
- узел
 boxMetrics, 36
 melt, 378
 анимационной кривой, 28–29, 496
 времени, 28–29, 37, 496
 выражения, 274–275
 доводки, 496
 зависимости, 42, 496
 преобразования
 (трансформации), 28–29, 43–47, 496
 формы, 43–47, 496
 узел-локатор категории shape, 44
 узлы. См. также C++ API, узлы.
 атрибуты, 28–38
 доступ к одному из атрибутов узла, 423
 иерархия, 46
 как простое хранилище данных, 37
 ОАГ, 41–53
 объединение в сеть, 24, 38
 описание интерфейса
 через атрибуты, 33–34
 описание, 28–29
 определение, 24, 496
 передача нескольких атрибутов на один
 вход, 40
 поток, 28
 проблемы обновления, 53
 свойства, 408–411
 содержащие массивы составных
 атрибутов, 32
 содержащие составные атрибуты, 31
 соединение, 38–41
 типы, 27
 узел зависимости, 42
 управление данными, 25
 функция compute, 29
 узлы, типы
 boxMetrics, 36
 sphereVolume, 34
 базовый вариант узла, 29
 простой узел, 30
 содержащие составные атрибуты, 31
 узел времени, 37
 узел с составным атрибутом, 31
 указатели, 15
 управление данными, 25
 управляемый ключ, 170, 496
 усечение, 496
 установка
 выравнивания, смещения
 и возможностей настройки столбца,
 225–226
 значения атрибута в текущее время, 415
 ключевых кадров, 170–172
 флага array, 411
 флага keyable, 410
 флага readable, 408
- Ф**
- файл userSetup.mel, 131
 файлы сценариев, 75
 фигурные скобки ({ }), ограничители
 блока кода в языке MEL, 112
 фильтр, 497
 флаги, 497
 разрешения считывания, 408
 распространения, 470, 497
 справочная информация о них, 77–78
 флажки (элементы размещения), 239
 формы, 292, 497
 функции
 addAttribute, MFnDependencyNode, 412
 boundingBox, 428, 435
 child, MDataHandle, 425
 color, 429
 colorRGB, 429
 draw, 428, 433–435

- errorString(), 321
 findPlug, 414
 getCirclePoint, 435
 getValue, 415
 inputValue, MDataBlock, 423
 isArray, 411
 isBounded, 428, 435
 isKeyable, 410
 isReadable, 408
 isUndoable, 362
 MGlobal::displayError(), 321
 MGlobal::displayWarning(), 321
 outputValue, MDataBlock, 423
 name(), 318–321
 perror(), 321
 redoIt, 399
 setArray, 411
 setKeyable, 410
 setReadable, 408
 setValue, 415
 sphrand, 279
 undoIt, 399
 viewFrame, 416
 атрибутов, 34–35
 кривые анимации, 168–170
 определение, 497
- функция compute**
- выходной атрибут в качестве входа, 36
 - вычисление выходов, 36
 - команды `setAttr`/`getAttr`, 412
 - локальные атрибуты узла, 33
 - описание, 29
 - определение, 497
 - с одним и более входным атрибутом, 33
 - узлы, 412–414
- Ц**
- циклические конструкции**
- `break`, 109
 - `continue`, 108–109
 - `do-while`, 108
 - `for`, 106–107, 126
 - `while`, 107
 - бесконечный цикл, 108
- Ч**
- определение, 497
- Ч**
- частицы, 497
 «черный ящик», 497
 чтение в память, 497
 чувствительность к регистру, 497
- Ш**
- шайдер, 498
 шум, 498
- Э**
- экземпляр,
 новый узел преобразования, 50–51
 экземпляры
- изменение исходного, 50
 - обзор, 50–53
 - определение, 498
 - преимущества, 52–53
- элемент**
- `helpLine`, 257
 - `toolCollection`, 249–250
 - размещения, 214, 498
- эффект**
- вспышки, 265–267
 - магнита, 268–271
 - прыгающего объекта, 261–265
- эффективность, размеры массивов, 122**
- Я**
- явная инициализация глобальных переменных, 115
 явное задание типа переменной, 121
 язык C, в сравнении с MEL, 477–479
 язык MEL
- '(одинарная кавычка), выполнение команд, 120
 - (оператор декремента), 123
 - ?; (вопросительный знак, двоеточие), условный оператор, 123
 - '(обратная кавычка), выполнение команд, 119
 - ++ (оператор инкремента), 123

- Echo All Commands, 69
 анимация, 162–210
 в сравнении с языком C, 477–479
 выделение (освобождение) памяти, 16
 выражения. См. Expression; выражения, графические интерфейсы пользователя, 67, 210–260
 для программистов на языке C, отличия, 477–479
 запрос, 81
 значок C, 81
 значок E, 81
 значок M, 81
 значок Q, 81
 интерпретируемый язык, 14
 команда eval, 120
 команда запроса, 79–80
 командное ядро, 20–21
 комментарии, 93–94
 методы выполнения, 119–121
 оператор for-in, 126
 оператор switch, 124–126
 определение, 484, 484
 организация цикла по элементам, 126
 переменные, присваивание одного значения нескольким, 122
 по умолчанию, 81
 повторное использование, 81
 поддерживаемые командные режимы, 81
 подставляемые арифметические операторы, 123
 правка, 81
 префиксная и постфиксная форма, 123
 производительность, 14
 псевдокод, 16
 различные способы выполнения команд, 119
 редактор сценариев, 68
 режим запроса, 80
 режим создания, 80
 режимы команд, 81
 результаты команд, 120
 свертывание длинных операторов, 123
 создание сценариев, 127–143
 создание, 81
 указатели, 16
- цепочка присваивания, 122
 язык программирования, 70–126
 язык сценариев, 67
- язык MEL, объекты**
- атрибуты, 158–162
 - доступ к свойствам, 79
 - иерархии, 146–148
 - команда delete, 144
 - команда listAll (), 145–146
 - команда ls, 144
 - команда objExists, 144
 - команда rename, 144
 - описание, 79
 - организация цикла, 145–146
 - отображение имен и типов, 145–146
 - проверка существования, 144
 - составление списка, 144
- язык MEL, отладка и тестирование сценариев**
- MAYA_SCRIPTS_PATH, 130
 - автоматический запуск сценариев, 131
 - каталоги сценариев по умолчанию, 128–130
 - команда about, 141
 - команда error, 138
 - команда exists, 139
 - команда print, 136
 - команда trace, 137
 - команда warning, 138
 - команда whatIs, 139
 - моноширинный шрифт, 127
 - обновление версий, 143
 - определение типов, 139–140
 - отображение номеров строк, 135–136
 - отображение предупреждений и ошибок, 138–139
 - подготовка к разработке, пользовательская среда, 131–133
 - поддержка версий, 141
 - проверки во время работы, 139
 - размещение, 142–143
 - расширение файла .mel, 128
 - собственные каталоги сценариев, 130–131
 - сохранение перед выполнением, 134

считывание сценариев MEL, 133-134
текстовый редактор, 127
файл `userSetup.mel`, 131
чтение в память, 133-134
язык **MEL**, преобразования
аффинное преобразование, 153
из локального пространства
в мировое, 150-151
матрицы преобразования, 149-150
мировое пространство, 150
перевод точки из локального
пространства, расчет окончательного
положения, 152-154
получение индекса экземпляра узла, 155
пространства, 150-152
процедура
`getInstanceIndex`, 154-158
процедура `objToWorld`, 152-154
процедура
`spaceToSpace`, 154-158
процедура
`transformPoint`, 152-154
язык сценариев, 498
языки программирования. *См. также*
C++; язык MEL.
выбор, 15-18
дополнительные конструкции, 122-126
дополнительные
методы выполнения, 119-121
защита информации, 18
команды, 70-81
комментарии, 93-94
межплатформенная переносимость, 17
область видимости, 112-119
операторы, 95-106
организация циклов, 106-109
переменные, 81-93
производительность, 18
простота применения, 15
процедуры, 109-112
разглашение, 18
сравнение функциональности, 16-17
эффективность, 121

Содержание

Отзывы критики на «Полное руководство по программированию Maya» -	5
Что делает эту книгу уникальной?.....	6
Исходные тексты.....	6
Предисловие.....	7
Об авторе.....	9
1. Введение.....	10
1.1. Возможности программирования Maya.....	11
1.1.1. Настройка.....	11
1.1.2. Интеграция.....	12
1.1.3. Автоматизация.....	12
1.1.4. Расширения.....	13
1.2. Интерфейсы программирования.....	13
1.2.1. MEL.....	13
1.2.2. C++.....	14
1.2.3. MEL или C++?.....	15
2. Основы организации Maya.....	19
2.1. Архитектура Maya.....	20
2.1.1. Обзор.....	20
2.2. Граф зависимости.....	21
2.2.1. Структура приложения.....	21
2.2.2. Сцена.....	25
2.2.3. Отображение графа зависимости.....	26
2.2.4. Поток данных.....	27
2.2.5. Узлы.....	28
2.2.6. Атрибуты.....	30
2.2.7. Функция compute.....	32
2.2.8. Соединение узлов.....	38
2.2.9. Узлы ОАГ.....	41
2.2.10. Обновление графа зависимости.....	53
3. Язык MEL.....	67
3.1. Введение.....	67
3.1.1. «За кулисами».....	68

3.2. Язык программирования MEL.....	70
3.2.1. Команды.....	70
3.2.2. Переменные.....	81
3.2.3. Комментарии.....	93
3.2.4. Операторы.....	95
3.2.5. Организация циклов.....	106
3.2.6. Процедуры.....	109
3.2.7. Область видимости.....	112
3.2.8. Дополнительные методы выполнения команд.....	119
3.2.9. Эффективность.....	121
3.2.10. Дополнительные конструкции языка.....	122
3.3. Создание сценариев.....	127
3.3.1. Текстовый редактор.....	127
3.3.2. Сохранение сценариев.....	128
3.3.3. Подготовительный этап разработки.....	131
3.3.4. Чтение в память.....	133
3.3.5. Отладка и тестирование.....	134
3.3.6. Поддержка версий.....	141
3.3.7. Размещение.....	142
3.4. Объекты.....	143
3.4.1. Основы.....	143
3.4.2. Иерархии.....	146
3.4.3. Преобразования.....	149
3.4.4. Атрибуты.....	158
3.5. Анимация.....	162
3.5.1. Время.....	162
3.5.2. Воспроизведение.....	165
3.5.3. Анимационные кривые.....	168
3.5.4. Скелеты.....	190
3.5.5. Траектории движения.....	207
3.6. Графический интерфейс пользователя.....	210
3.6.1. Введение.....	210
3.6.2. Основные понятия.....	213
3.6.3. Окна.....	218
3.6.4. Схемы размещения.....	224
3.6.5. Элементы управления.....	234
3.6.6. Связывание элементов.....	250
3.6.7. Обратная связь с пользователем.....	256
3.7. Выражения.....	261

3.7.1.	Освоение выражений.....	261
3.7.2.	Рекомендации по составлению выражений.....	271
3.7.3.	Отладка выражений.....	275
3.7.4.	Выражения для работы с частицами.....	278
3.7.5.	Выражения повышенной сложности.....	286
4.	C++ API	289
4.1.	Введение.....	289
4.2.	Основные понятия	292
4.2.1.	Абстрактный уровень	293
4.2.2.	Классы.....	294
4.2.3.	Класс MObject	301
4.2.4.	Наборы функций MFn.....	303
4.3.	Разработка подключаемых модулей.....	306
4.3.1.	Разработка подключаемых модулей под Windows.....	307
4.3.2.	Инициализация и deinициализация.....	314
4.3.3.	Ошибки.....	318
4.3.4.	Загрузка и выгрузка.....	324
4.3.5.	Размещение.....	326
4.4.	Команды.....	327
4.4.1.	Создание команд.....	327
4.4.2.	Модуль Posts1	328
4.4.3.	Поддержка аргументов.....	334
4.4.4.	Модуль Posts2	334
4.4.5.	Модуль Posts3	336
4.4.6.	Организация справочной системы.....	338
4.4.7.	Отмена и повторное выполнение.....	342
4.4.8.	Редактирование и запросы.....	353
4.5.	Узлы.....	362
4.5.1.	Модуль GoRolling	363
4.5.2.	Модуль Melt	378
4.5.3.	Модуль GroundShadow	391
4.5.4.	Атрибуты.....	404
4.5.5.	Функция compute.....	412
4.5.6.	Подключения.....	414
4.5.7.	Блоки данных.....	423
4.5.8.	Рекомендации по проектированию узлов.....	425
4.6.	Локаторы.....	428
4.6.1.	Модуль BasicLocator	429
4.7.	Манипуляторы.....	438

4.7.1. Модуль BasicLocator2.....	438
4.8. Деформаторы.....	450
4.8.1. Модуль <i>SwirlDeformer</i>	451
4.8.2. Изменения в графе зависимости.....	457
4.8.3. Вспомогательные инструменты.....	459
4.8.4. Модуль <i>SwirlDeformer2</i>	459
4.9. Расширенные возможности C++ API.....	463
4.9.1. Общие вопросы.....	463
4.9.2. Граф зависимости.....	466
Приложение А. Дополнительные ресурсы.....	473
Сетевые ресурсы.....	473
Web-сайт книги.....	473
Дополнительные Web-сайты.....	474
Система Maya.....	474
Документация.....	475
Примеры.....	475
Приложение В. MEL для программистов на языке С.....	477
Приложение С. Литература для дальнейшего изучения.....	480
Математика.....	480
Программирование.....	481
Общие вопросы.....	481
Язык C++.....	481
Компьютерная графика.....	481
Общие вопросы.....	481
Моделирование.....	482
Анимация.....	482
Синтез изображений.....	482
Словарь терминов.....	483
Предметно-именной указатель.....	499

ИЗДАТЕЛЬСТВО

«ОЦ КУДИЦ-ОБРАЗ»

Тел.: (095) 333-82-11; ok@kudits.ru, <http://www.kudits.ru/publish>

КНИГИ В ПРОДАЖЕ

ACCESS 2003. Самоучитель с примерами. Гончаров А. Ю.

272 с. 2004 г. Опт. цена 99 р.

ArchiCAD 8 на практике. Ланцов А. Л.

656 с. 2004 г. Опт. цена 250 р.

ArchiCAD 8.0. Справочник с примерами. Титов С.А

480 с. 2003 г. Опт. цена 209 р.

ArchiCAD 7.0. Титов С.

400 с. 2003 г. Опт. цена 154 р.

ArchiCAD: полезные рецепты. Комплект. Титов С.

272 с. 2003 г. Опт. цена 171,6 р.

AutoCAD 2002/2002 LT/2000. Справочник команд. Россоловский А.

720 с. 2002 г. Опт. цена 220 р.

IC: практика настройки оперативного учета. Ражиков М.Ю.

256 с. 2003 г. Опт. цена 121 р.

AES - стандарт криптографической защиты. Конечные поля. Зензин О.С., Иванов М.А.

176 с. 2002 г. Опт. цена 55 р.

C++ & Visual Studio. NET. Самоучитель программиста. Баженова И.Ю.

448 с., 2003 г. Опт. цена 132 р.

C#. Визуальное проектирование приложений. Фролов А., Фролов Г.

512 с. 2003 г. Опт. цена 220 р.

Delphi 7. Самоучитель. Климова Л.М.

480 с. 2004 г. Опт. цена 124,3 р.

Delphi 7. Самоучитель программиста. Баженова И.Ю.

432 с. 2002 г. Опт. цена 110 р.

Dreamweaver MX. Базовый курс Божко А.Н.

Flash & XML. Руководство разработчика. Джекобсон Д. Пер., с англ.

352 с. 2003 г. Опт. цена 111 р.

Стих: настольная книга. Клейтон Е. Крукс. Пер. с англ.

320 с. 2004 г. Опт. цена 132 р.

IT-безопасность. Стоит ли рисковать корпорацией? Маккарти Л. Пер. с англ.

208 с. 2004 г. Опт. цена 110 р.

Jamagie: программирование игр и симуляторов. Перес Серхио. Пер. с англ.

288 с. 2004 г. Опт. цена 132 р.

Java: основы Web-служб. Тост Андре. Пер. с англ.

464 с. 2004 г. Опт. цена 143 р.

Linux: создание виртуальных частных сетей (VPN). Колесников О., Хетч Брайан

464 с. 2004 г. Опт. цена 154 р.

.Net Framework: Библиотека классов. Темплман Дж., Виттер Д.
672 с. 2003 г. Опт. цена 298,1 р.

Office XP. Афанасьев Д., Баричев С., Плотников О.
356 с. 2002 г. Опт. цена 84,7 р.

PageMaker 6.5/7.0. Самоучитель. Вовк Е.Т.
352 с. 2002 г. Опт. цена 121 р.

Pascal 7.0. Основы практического программирования. Решение типовых задач. Климова Л.М.
528 с. 2000 г. Опт. цена 105,6 р.

Photoshop CS: технология работы. Сканирование, ретушь. Божко А.Н.
624 с. 2004 г. Опт. цена 176 р.

Sendmail: настройка и оптимизация. Кристенсон Ник. Пер. с англ.
272 с. 2004 г. Опт. цена 110 р.

QuarkXPress 5.0. Самоучитель. Вовк Е.Т.
288 с. 2002 г. Опт. цена 88 р.

Linux®/Linux: теория и практика программирования. Моли Брюс.
576 с. 2004 г. Опт. цена 143 р.

Visual Basic.NET, Visual Basic 6.0, Visual Basic for Applications 6.0. Король В.И.
496 с. 2002 г. Опт. цена 143 р.

Web-дизайн: Photoshop & Dreamweaver. 3 ключевых этапа. Смит Колин. Пер. с англ.
264 с. 2004 г. Опт. цена 99 р.

Windows XP Professional. Проффит Б.
416 с. 2002 г. Опт. цена 112,2 р.

Администрирование баз данных. Крейг С.Маллинс
752 с. 2003 г. Опт. цена 253 р.

Ассемблер в задачах защиты информации. Иванов М.А.
320 с. 2002 г. Опт. цена 96,8 р.

Безопасность: технологии, средства, услуги. Барсуков В.С.
496 с. 2001 г. Опт. цена 99 р.

Брендинг – Дорога к мировому рынку. Анхолд Симон. Пер. с англ.
272 с. 2004 г. Опт. цена 99 р.

Информационная архитектура. Чертежи для сайта. Уодтке К. Пер. с англ.
320 с. 2004 г. Опт. цена 110 р.

Искусство дизайна с компьютером и без... Пер. с англ.
208 с. 2004 г. Опт. цена 88 р.

Исследуем Maya 4: 30 уроков в 3D. Шонхер М. Пер. с англ.
288 с. 2002 г. Опт. цена 99 р.

Кабельные системы: проектирование, монтаж и обслуживание. Верити Бет, Пер. с англ.
400 с. 2004 г. Опт. цена 165 р.

Как преподнести себя на рынке труда. Хангерленд Бафф. Пер. с англ.
224 с. 2003 г. Опт. цена 67,10 р.

Как успешно руководить проектами. Серебрянная пуля. Фергус О'Коннелл
288 с., 2003 г. Опт. цена 121 р.

- Коммутаторы CISCO.** Олои Ш., Ноттингем Х.
528 с., 2003г. Опт.цена 231 р.
- Компьютерная анимация. Теория и алгоритмы,** РНК-Прент, Пер. с англ.
560 с., 2004г. Опт.цена 242 р.
- Компьютерные игры: секреты бизнеса.** Франсуа Ларами. Пер. с англ.
416 с. 2004 г. Опт.цена 169,4 р.
- Компьютерные презентации: от риторики до слайд-шоу.** Елизаветина Т.М. Пер. с англ.
240 с. 2004 г. Опт.цена 77 р.
- Лечение псориаза - естественный путь.** Пегано Дж. Пер. с англ.
288 с. 2001 г. Опт.цена 132 р.
- Маршрутизаторы CISCO для IP-сетей.** Руденко И., Tsunami Computing.
656 с. 2003 г. Опт.цена 242 р.
- Мир InterBase. Архитектура, администрирование и разработка приложений баз данных в InterBase/FireBird/Yaffil.** Изд-е 2-е, дополн. Ковязин А., Востриков С.
496 с. 2003 г. Опт.цена 220 р.
- Наука отладки.** Тэллес М., Хсих Ю.
560 с. 2003 г. Опт.цена 187 р.
- Объектно-ориентированное программирование на ActionScript.** Пер. с англ.
416 с. 2003 г. Опт.цена 125,4 р.
- Оптимизация баз данных. Принципы, практика, решение проблем.** Денис Шаша, Филипп Бонне.
432 с. 2004 г. Опт. цена 121,00
- Основы пространственных баз данных.** Шаши Ш., Санжей Ч. Пер. с англ.
336 с. 2004 г. Опт.цена 121 р.
- Персональная защита от хакеров. Руководство для начинающих.** Форд Дж.
272 с. 2002 г. Опт.цена 92,4 р.
- Платформа программирования J2ME** для портативных устройств. Пирумян В.
352 с. 2003 г. Опт.цена 132 р.
- Популярные Web-приложения на Flash MX.** Чанг Тим К., Кларк Шон
272 с. 2003 г. Опт.цена 129,8 р.
- Практическая отладка в C++.** Форд А., Теорн Т.
144 с. 2002 г. Опт.цена 44 р.
- Поточные шифры.** Асоксов А.В., Иванов М.А
332 с. 2003 г. Опт.цена 110р.
- Принятие управленческих решений.** Варфоломеев В.И., Воробьев С.Н.
288 с. 2001 г. Опт.цена 99 р.
- Программирование Access 2002 в примерах.** Вилларикал Б.
496 с. 2003 г. Опт.цена 171,6 р.
- Программирование** для Adobe Illustrator на языках Visual Basic и AppleScript, Пер. с англ.
192 с. 2003 г. Опт.цена 79,20 р.
- Программирование на MEL для Maya.** Марк Р. Уилкинс, Крис Казмиер Пер. с англ.
192 с. 2004 г. Опт.цена 79,20 р.
- Секреты дизайнера. Профессиональные приемы в Adobe Photoshop 7 и Adobe Illustrator 10.** Сеймур-Коэн Л.
192 с. 2003 г. Опт.цена 79,2 р.

- Секреты разработки игр в Macromedia Flash MX. Джоб Макар. Пер. с англ.
576 с. 2004 г. Опт. цена 154 р.
- СИ++. Практическое программирование. Решение типовых задач. Климова Л.М.
596 с. 2001 г. Опт. цена 125,40 р.
- Смарт-карты: настольная книга разработчика. Юргенсен Т.М., Гатери С.Б.
416 с. 2003 г. Опт. цена 220 р.
- Создание Web-страниц средствами CSS. Шмитт Кристофер Пер. с англ.
368 с. 2003 г. Опт. цена 121 р.
- Стандарты вычислительных сетей. Взаимосвязи сетей. Справочник. Щербо В.К.
276 с., 2000 г. Опт. цена 90,20 р.
- Тестирование Web-приложений. Стотлемайер Д.
240 с. 2003 г. Опт. цена 94,6 р.
- Технология цифровой фотографии. Надеждин Н. Я.
240 с. 2004 г. Опт. цена 88 р.
- Цифровая модель человека. Пер. с англ. Кен Бриллиант.
400 с. 2004 г. Опт. цена 156,20
- Этот великолепный Illustrator 10. Пер. с англ.
432 с. 2003 г. Опт. цена 154 р.

**ПРИГЛАШАЕМ
АВТОРОВ КНИГ ПО КОМПЬЮТЕРНОЙ ТЕМАТИКЕ,
ПЕРЕВОДЧИКОВ И НАУЧНЫХ РЕДАКТОРОВ**

ВОЗМОЖНО ВАС ЗАИНТЕРЕСУЮТ ДРУГИЕ НАШИ ИЗДАНИЯ



A. V. Фролов, G. V. Фролов
Визуальное программирование
приложений C#.

ISBN 5-9579-0001-X



С. Перес
Jamagic: программирование
игр и симуляторов

ISBN 5-9579-0008-7



M. R. Вилкинз, K. Казмие
Программирование на MEL
для Maya

ISBN 5-93378-097-9



K. Е. Крукс II
gmax: настольная книга

ISBN 5-9579-0026-5



M. Шонхер
Исследуем Maya 4:
30 уроков в 3D

ISBN 5-93378-057-X



К. Бриллиант
Цифровая модель человека

ISBN 5-9579-0006-0



P. Пэрент
Компьютерная анимация

ISBN 5-93378-095-2



Под ред. Ф. Д. Ларами
Компьютерные игры:
секреты бизнеса

ISBN 5-9579-0011-7

**По вопросу приобретения книг обращайтесь в издательство по тел.: 333-82-11,
с 11⁰⁰ до 17⁰⁰. Наш адрес: ул. Профсоюзная, д. 84/32, под. 6, эт. 11**

ISBN 5-93378-098-7

9 785933 780984

КУДИЦ-ОБРАЗ

Тел./факс: (095) 333-82-11, 333-65-67
E-mail: ok@kudits.ru; <http://books.kudits.ru>
121354, Москва, а/я 18, "КУДИЦ-ОБРАЗ"

