

# Написание скриптов для Blender 2.49

Расширьте мощность и гибкость Блендера с помощью Питона: высокоуровневого, легкого для изучения скриптового языка

Michel Anders  
BIRMINGHAM - MUMBAI

Перевод: Striver

# Credits

- **Author** Michel Anders
- **Reviewer** Pang Lih-Hern
- **Acquisition Editor** Sarah Cullington
- **Development Editor** Mehul Shetty
- **Technical Editor** Namita Sahn
- **Indexers** Hemangini Bari, Rekha Nair
- **Editorial Team Leader** Akshara Aware
- **Project Team Leader** Lata Basantani
- **Project Coordinator** Shubhanjan Chatterjee
- **Proofreader** Jeff Orloff
- **Graphics** Geetanjali Sawant
- **Production Coordinator** Melwyn D'sa
- **Cover Work** Melwyn D'sa

# Об авторе

Michel Anders, После завершения его исследований химии и физики, где он тратил больше времени на компьютерное моделирование, чем на реальные эксперименты, он понял что его реальные интересы лежат в области IT и технологий Internet. Он работал как IT менеджер для нескольких различных компаний, включая провайдера Internet и больницу.

К настоящему времени он управляет R&D отделом в Aia Software, ведущим разработчиком программного обеспечения композиции документа (?? document composition software). Он счастливо живет на небольшой, преобразованной ферме со своим партнером, 3 кошками, и 12 козлами.

Он использует Блендер с версии 2.32, хотя он первым признаёт, что его искусство в лучшем случае наивно. Он любит помогать людям с Блендером и вопросам, связанным с Питоном и к нему можно обращаться, на [blenderartists.org](http://blenderartists.org) он известен как "varkenvarken".

Сначала, я хотел бы благодарить всех замечательных людей в Packt Publishing. Без их помощи эта книга не была бы написана. Также, я хотел бы поблагодарить своего партнера и моих коллег по работе, мирящихся с моими бесконечными переговорами о Блендере. Наконец, я хотел бы поблагодарить всех тех людей в сообществе Блендера, которые вместе сделали Блендер таким замечательным приложением.

# Оглавление

<b>Предисловие.....</b>	<b>6</b>	Объекты Блендера.....	24
Что эта книга охватывает.....	6	Модуль bpy.....	26
Что Вам нужно для этой книги.....	7	Рисование на экране.....	26
Для кого эта книга.....	7	Утилиты.....	26
Соглашения.....	7	Итог.....	27
Обратная связь с читателем.....	8	<b>Создание и редактирование объектов.....</b>	<b>28</b>
Поддержка покупателей.....	8	Срееру crawlies (ползучий ужас) - графический интерфейс пользователя для конфигурирования объектов.....	28
Опечатки.....	8	Создание интерфейса пользователя.....	29
<b>Расширение Блендера с помощью Питона.....</b>	<b>10</b>	Создание жуков — требует некоторой сборки.....	29
API Блендера.....	11	Создание пользовательского интерфейса.....	29
Много энергии.....	11	Запоминание выбора.....	30
Некоторые батарейки включены.....	12	Вся мощь графики Блендера .....	31
Проверьте наличие полного дистрибутива Питона.....	12	Создание меш-объекта.....	34
Инсталлирование полного дистрибутива Питон.....	13	Преобразование топологии меша.....	34
Интерактивная консоль Питона.....	13	Схема кода сшивания рёберных циклов.....	35
Изучение встроенных модулей, функция help() .....	14	Ослепите вашего босса - гистограммы в стиле Блендер.....	37
Изучение встроенных функций, функция dir().....	15	Скрипт построения гистограммы.....	38
Знакомство со встроенным редактором.....	15	Таинственные грани - выбор и редактирование граней в мешах.....	40
Пример с редактором.....	16	Выбор искривлённых (не-планарных) четырёхугольников.....	40
Первые шаги: Hello world.....	17	Схема и код выбора искривлённых граней.....	42
Внедрение скрипта в меню Блендера.....	18	Выбор слишком острых граней.....	43
Внедрение скрипта в систему помощи Блендера.....	19	Выбор вершин со множеством рёбер.....	44
Не запутайтесь, оставайтесь объективным.....	20	Выбор полюсов .....	44
Добавление различных типов объектов из скрипта.....	21	Выбор полюсов, ещё раз.....	45
Добавление меш-объекта.....	21	Определение объема меша.....	46
Распространение скриптов.....	23	Определение центра масс меша.....	48
API Блендера.....	23	Некоторые замечания о точности.....	50
Модуль Blender.....	24		

Растущий подсолнечник - присвоение родителей и группирование объектов.....	51
Группы.....	51
Отношения родитель-потомок.....	52
Выращивание подсолнуха из семечка.....	52
Дублирование против связанной копии.....	53
Выращивание подсолнуха.....	53
Итог.....	55
<b>Группы вершин и материалы.....</b>	<b>56</b>
Группы вершин.....	56
Весомый вопрос.....	58
Схема кода: leftright.py.....	58
Модификаторы.....	59
Гравировка .....	61
Конвертация объекта Text3d в меш.....	62
Выдавливание рёберного цикла.....	63
Расширение (Expanding) рёберного цикла.....	63
Собираем всё вместе: Engrave.py.....	66
Полет искр.....	69
Расчет локальной кривизны.....	70
Схема кода: curvature.py.....	70
Собираем всё это вместе: Огни святого Эльма.....	71
Кости.....	71
Тик-так.....	72
Схема кода: clock.py.....	72
Get a bit of backbone boy!.....	74
Материалы.....	76
Материалы Объекта против материалов ObData.....	76
Назначение материалов частям Объекта.....	77
Вершинные цвета против материалов граней.....	78
Добавление материалов в нашу гравюру.....	79

Итог.....	80
<b>PyDrivers и Constraints (Управляющие объекты и Ограничения)81</b>	
Акцентируем внимание на свойствах анимации.....	81
IPO.....	81
IPO-каналы и IPO-кривые.....	82
Ограничения.....	84
Различия между управляющими объектами (drivers) и ограничениями...	84
Программирование ограничений.....	85
Программирование кривых IPO.....	85
Управляющие объекты (PyDrivers).....	85
Ограничения на Питоне (PyConstraints).....	86
Установка времени - один управляет всеми.....	86
Сокращения.....	89
Преодоление ограничений: pydrivers.py.....	89
Внутреннее сгорание — корреляция сложных изменений.....	90
Больше мощности — комбинирование нескольких цилиндров в двигателе...	93
Добавление простых ограничений.....	94
Определяем сложные ограничения.....	94
Шаблон ограничения в Блендере.....	95
Вы тоже находите меня притягательным?.....	96
Привязка к вершинам меша.....	97
Выравнивание вдоль вершинной нормали.....	99
Привязка к вершинам в вершинной группе.....	101
Итог.....	103
<b>Действия при изменениях кадров.....104</b>	
Анимация видимости объектов.....	104
Потускнение материала.....	104
Изменение слоев.....	106
Обратный отсчет - анимация таймера с помощью скриптсвязи.....	106

Я буду следить за вами.....	109
Схема программы: AuraSpaceHandler.py.....	109
Использование тем.....	111
Снова о мешах — создание отпечатков.....	113
Пользовательский интерфейс.....	115
Вычисление отпечатка.....	118
Итог.....	120
<b>Ключи формы, кривые IPO, и Позы.....</b>	<b>121</b>
Обидчивый субъект - определение IPO из ничего.....	121
Схема программы: orbit.py.....	121
Много проглотил - определение поз.....	123
Применение peristaltic.py к арматуре.....	126
Get down с ритмом - синхронизация ключей формы со звуком.....	127
Манипуляция звуковыми файлами.....	127
Схема кода: Sound.py.....	128
Анимация меша .wav-файлом: последовательность действий.....	130
Итог.....	131
<b>Создание заказных шейдеров и текстур с помощью Pynodes...132</b>	
Основы.....	134
От нодов к Pynodes.....	134
Регулярное заполнение.....	136
Anti-aliasing.....	139
Индексирование текстуры вектором.....	140
Свежий бриз - текстуры с нормальями.....	141
Капли - анимированные Pynodes.....	143
Параметры времени рендера.....	143
Всё, что выглядит хорошо — это хорошо.....	143
Хранение дорогостоящих результатов для многократного использования...144	
Вычисление нормалей.....	145

Собираем всё это вместе.....	145
Грозовой перевал — материал, зависимый от наклона.....	147
Определение уклона.....	148
Мировое пространство против пространства камеры.....	148
Мыльные пузыри — шейдер, зависимый от точки зрения .....	150
Итог.....	154
<b>Рендеринг (визуализация) и обработка изображений.....155</b>	
Различные виды - комбинирование множества направлений камеры . .155	
Схема кода - combine.py.....	156
Рабочий процесс - как продемонстрировать вашу модель.....	160
Now, strip — создание киноленты из анимации.....	160
Рабочий процесс — использование strip.py.....	164
Рендер билбордов.....	164
Рабочий процесс - использование cardboard.py.....	169
Генерация вопросов CAPTCHA.....	170
Разработка сервера CAPTCHA.....	171
Итог.....	174
<b>Расширение вашего инструментария.....175</b>	
В Сеть и дальше - публикация готового рендера на FTP.....	175
Весенняя уборка - архивация неиспользуемых изображений.....	178
Расширение редактора - поиск с регулярными выражениями.....	181
Расширение редактора - взаимодействие с Subversion.....	183
Отправка (commit) файла в хранилище.....	183
Обновление (updating) файла из хранилища.....	184
Работа с хранилищем.....	185
The need for speed (жажда скорости) — использование Psyc.....	186
Включение Psyc.....	187
Итог.....	188

# Предисловие

Блендер несомненно является наиболее мощным и разносторонним 3D-пакетом, доступным с открытыми исходными текстами. Его функциональность близка к профессиональным пакетам, или даже превосходит многие из них. Встроенный в Блендер интерпретатор языка Питон играет важную роль в наращивании этой мощности и позволяет расширять функциональность ещё дальше. Тем не менее, освоение написания скриптов языка и знакомство со многими возможностями, которые предлагает Блендер через свой Питон-API может быть непростым предприятием.

Эта книга покажет, как получить максимум от Блендера, показывая практические решения многих реальных проблем. Каждый пример является полностью рабочим скриптом, который объясняется шаг за шагом самым подробным образом.

## **Что эта книга охватывает**

*Глава 1, Расширение Блендера с помощью Питона*, дает вам обзор того, что может и что не может быть выполнено с помощью Питона в Блендер. Это покажет Вам как установить полный дистрибутив Питона, и как использовать встроенный редактор. Вы также узнаете как записывать и запускать простой скрипт на Питоне, и как внедрить его в систему меню Блендера.

*Глава 2, Создание и Редактирование Объектов*, вводит объекты и меши, и Вы увидите как манипулировать ими программно. В частности, Вы узнаете как создавать конфигурируемые меш-объекты, разрабатывать графический интерфейс пользователя, и как добиться, чтобы ваш скрипт сохранил настроенные пользователем опции, чтобы впоследствии использовать их многократно. Вы также узнаете как выбирать вершины и грани в меше, делать один объект родителем другого, и как создавать группы. Наконец, эта глава показывает как запускать Блендер с командной строки, рендерить в фоне, и как обрабатывать параметры командной строки.

*Глава 3, Группы Вершин и Материалы*, расскажет Вам о множественном использовании групп вершин и о том, какими

разносторонними они могут быть. Вы узнаете как задавать группы вершин и как назначать вершины в группу. Вы также узнаете, как Вы можете использовать эти группы вершин для модификаторов и арматур. Вы также посмотрите на приложение различных материалов на разные грани, и на то, как назначать цвета вершинам.

Глава 4, Pydrivers и Ограничения, показывает, как Вы можете соединить встроенные Ограничения с объектами Блендера и как задавать сложные связи между анимированными свойствами, используя так называемые pydrivers. Вы также определите новые сложные ограничения, которые могут быть использованы также, как и встроенные Ограничения. В частности, Вы увидите, как управлять одним IPO из другого посредством выражения на Питоне, как работать с некоторыми ограничениями, встроенными в pydrivers, и как ограничивать движение объектов и костей, добавляя Ограничения. Эта глава научит Вас, как писать ограничение на Питоне, которое привяжет объект к ближайшей вершине на другом объекте.

*Глава 5, Действия на изменении кадров*, фокусируется на написании скриптов, которые могут использоваться, чтобы действовать на определенных событиях. Вы можете изучить скриптовые связи и пространственные операторы (You can learn what script links and space handlers) - и как они могут быть использованы для выполнения мероприятий по изменению каждого кадра в анимации. Вы также увидите как ассоциируется дополнительная информация с объектом, как использовать скриптовые связи, чтобы воспроизводить появление или исчезновение объекта, изменяя формат или изменяя прозрачность,и

как реализовать схему, связывающую различные меши с объектом в каждом кадре.

Наконец, Вы можете изучить способы увеличить функциональность 3D-вида.

*Глава 6, Ключи Формы, IPOs, и Poses*, открывает, что есть ещё множество кривых IPO, которые могут оказаться полезными в анимированных сценах. Хотя кривые IPO были введены в Главе 4,

здесь Вы узнаете, как определять IPO для всех типов объектов, связанных ключей формы с мешем, и как задавать кривые IPO для этих ключей формы. Вы также взглянете на позирование арматуры и объединение поз в действия.

*Глава 7, Создание заказных Шейдеров и Текстур с помощью Pynodes*, вводит Pynodes и вы узнаете, как они позволяют определять совершенно новые текстуры и материалы. Вы узнаете как писать Pynodes, которые создают простые цветные узоры, Pynodes, которые создают узоры с нормальными, а также Вы узнаете как анимировать Pynodes. Эта глава также объясняет Pynodes, которые производят материалы, зависящие от высоты и наклона, и даже создают шейдеры, которые реагируют на угол падающего света.

*Глава 8, Рендеринг и обработка изображения*, переходит к процессу рендеринга в целом. Вы можете автоматизировать этот процесс, объединять различными способами результирующие изображения, и даже превращать Блендер в специализированный веб-сервер. В частности, Вы узнаете как автоматизировать процесс рендеринга, создавать многочисленные виды для продукта презентации, и создавать рекламные щиты из сложных объектов. Вы узнаете о путях расширения Блендера с помощью некоторых внешних библиотек манипулирования изображениями, включая результаты рендера.

*Глава 9, Расширение ваших инструментов*, - меньше о рендеринге и больше об облегчении жизни при использовании Блендера изо дня в день, расширением его функциональности. В этой главе Вы узнаете как каталогизировать и архивировать активы, такие как карты изображений, публиковать отрендеренные изображения автоматически через FTP, расширять функциональность встроенного редактора посредством поиска регулярными выражениями, ускорять вычисления используя Psyc - компилятор «на лету», и добавлять управление версиями к вашим скриптам с помощью Subversion.

Приложение А, Ссылки и Ресурсы, дает Вам список большинства ресурсов, использованных в этой книге вместе с некоторой обычно полезной информацией.

*Приложение В, Частые Западни*, освещает некоторые общие

вопросы, которые появляются более часто, чем другие, что делать с некоторыми ошибками.

*Приложение С, Будущие Разработки*, является заключительным приложением, которое пробует показать что предвидится в будущем, и как это может затронуть Вас, поскольку и Блендер и Питон постоянно развиваются далее.

## **Что Вам нужно для этой книги**

Все примеры в книге используют Блендер 2.49 (доступный на сайте [www.blender.org](http://www.blender.org)) и встроенный язык Питон 2.6.x. Многие примеры допускают, что у вас есть полный дистрибутив Питона ([www.python.org](http://www.python.org)). В Главе 1, Расширение Блендера с помощью Питона, Вам рассказывается как установить полный дистрибутив, если Вы его ещё не имеете. Блендер и Питон - платформонезависимые, и все примеры должны работать одинаково хорошо в Windows, Linux, и Mac. Несколько дополнительных модулей могут также использоваться, и инструкции по их загрузке предусмотрены, где это необходимо. Все примеры могут быть загружены с веб-сайта издателя (<http://www.packtpub.com>).

## **Для кого эта книга**

Эта книга предназначена для пользователей, довольных Блендером, как инструментом моделирования и рендеринга, и которые хотят расширить свои навыки использованием скриптов Блендера для автоматизации трудоемких задач и достижения результатов, в противном случае невозможных. Опыт в Блендере имеет важное значение, а также небольшой опыт в программировании на Питоне.

## **Соглашения**

В этой книге Вы найдете множество стилей текста, которые определяют различные типы информации. Вот некоторые примеры этих стилей, и объяснения их значений.

Кодовые слова в тексте показаны следующим образом: "файл на Питоне с мешевыми строительными блоками называется

`mymesh.py`, так что первая часть нашего кода содержит следующий оператор `import`."

Блок кода вставляется следующим образом:

```
def event(evt, val):
    if evt == Draw.ESCKEY:
        Draw.Exit() # exit when user presses ESC
    return
```

Когда мы хотим привлечь ваше внимание к конкретной части блока кода, важные строки или пункты показываются жирным шрифтом:

```
def error(text):
    Draw.Register(lambda:msg(text), event, button_event)
```

Любая командная строка ввода или вывода пишется следующим образом:

**blender -P /full/path/to/barchart.py**

Новые условия и важные слова показываются жирным шрифтом. Слова, которые Вы видите на экране в меню или в диалогах, например, появляются в тексте подобно этому: "Затем мы можем применить эту группу вершин к параметру плотности на **дополнительной** панели контекста частиц, чтобы управлять эмиссией."



Предупреждения или важные примечания появляются в блоке, подобном этому.



Подсказки и хитрости появляются вот так.

## Обратная связь с читателем

Обратная связь с нашими читателями всегда приветствуется. Сообщите то, что Вы думаете об этой книге - что Вам понравилось

или может быть не понравилось. Обратная связь с Читателем важна для нас, чтобы разрабатывать издания, от которых Вы действительно получите максимальную отдачу.

Для общей обратной связи, просто пошлите эл.почту на [feedback@packtpub.com](mailto:feedback@packtpub.com), и упомяните название книги в теме вашего сообщения.

Если есть книга, которая Вам нужна, и Вы хотели бы увидеть, как мы её опубликовали, пожалуйста пошлите нам заметку в SUGGEST A TITLE (Предлагаем издание) в [www.packtpub.com](http://www.packtpub.com) или эл.почте [suggest@packtpub.com](mailto:suggest@packtpub.com).

Если существует тема, в которой вы являетесь экспертом и Вы заинтересованы в написании или содействии написанию книги об этом, смотрите наше руководство автора по адресу [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Поддержка покупателей

Теперь, когда Вы - гордый владелец книги Packt, у нас есть множество вещей Вам в помощь, чтобы Вы могли получить максимум от вашего приобретения.

### Загрузка примеров кода для книги



Посетите [https://www.packtpub.com/sites/default/files/downloads/0400\\_Code.zip](https://www.packtpub.com/sites/default/files/downloads/0400_Code.zip), чтобы непосредственно загрузить код примера.

Загружаемые файлы содержат инструкции о том, как их использовать.

## Опечатки

Хотя мы заботились о гарантировании точности нашего содержимого, ошибки случаются. Если Вы найдёте ошибку в одной из наших книг - может быть ошибку в тексте или в коде, мы будем благодарны, если Вы сообщите нам об этом. Сделав так, Вы можете



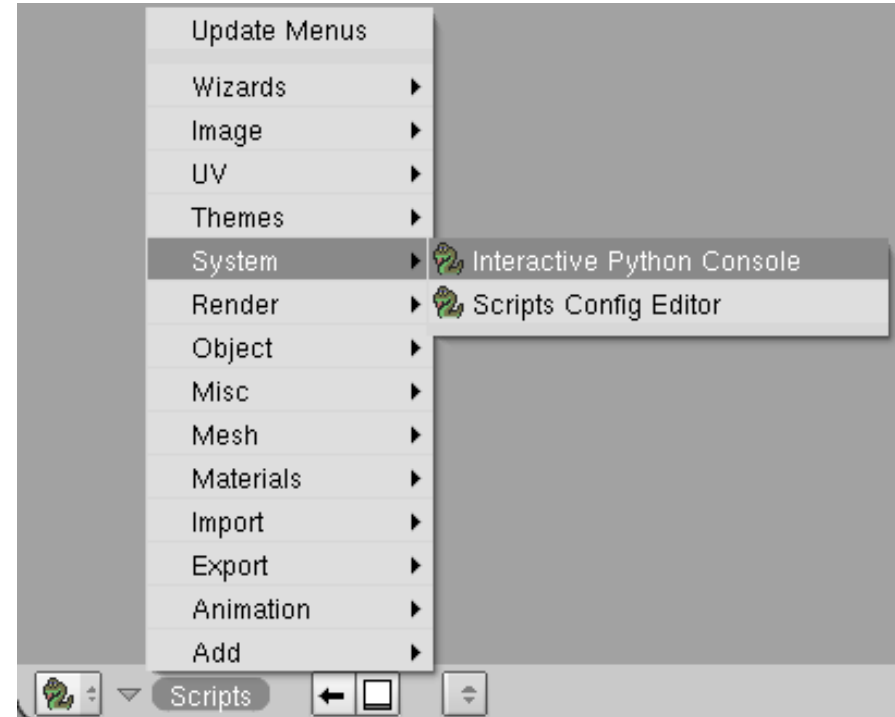
оградить других читателей от расстройства и поможете нам улучшить последующие версии этой книги. Если Вы найдёте любые опечатки, пожалуйста сообщите о них, посетив <http://www.packtpub.com/support>, выбрав вашу книгу, нажмите на ссылку **let us know**, и введите детали ваших опечаток. Как только ваши опечатки будут проверены, ваше сообщение будет принято и опечатки будут загружены на наш веб-сайт или добавлены к любому списку существующих опечаток, в секции Errata этого издания. Любые существующие опечатки можно увидеть, выбрав ваше издание в <http://www.packtpub.com/support>.

## Расширение Блендера с помощью Питона

Прежде, чем мы начнём разработку скриптов в Блендере, мы во всяком случае, должны удостовериться, что у нас есть все необходимые инструменты. После этого мы должны познакомиться этими инструментами, чтобы мы могли с уверенностью использовать их. В этой главе, мы посмотрим на:

- Что можно и чего нельзя выполнить с помощью Питона в Блендере
- Как установить полный дистрибутив Питона
- Как использовать встроенный редактор
- Как запускать скрипты на Питоне
- Как изучать встроенные модули
- Как писать простой скрипт, который добавляет объект в сцену Блендера
- Как регистрировать скрипт в меню скриптов Блендера
- Как документировать ваш скрипт в дружественном к пользователю виде
- Как распространять скрипт

С таким количеством пунктов кажется нужно очень много изучить, но к счастью, кривая обучения не такая уж крутая, как она могла бы показаться. Давайте просто по-быстрому наберём несколько строк на Питоне, чтобы разместить простой объект в нашей сцене Блендера, просто, чтобы показать что мы можем, перед тем, как мы уйдём с головой в более глубокие воды.



2. Откройте интерактивную консоль Питона (посмотрите на скриншот вверху, чтобы увидеть, где её найти).

3. Наберите следующие строки (заканчивайте каждую с помощью *Enter/Return*).

```
mesh = Mesh.Primitives.Monkey()
Scene.GetCurrent().objects.new(mesh, 'Suzanne')
Window.RedrawAll()
```

Вуаля! Это - все, что нужно для добавления в сцену Сюзанны, знаменитого талисмана Блендера.

1. Запустите Блендер с пустой сценой



## API Блендера

Почти всё в Блендере доступно из скриптов Питона, но есть некоторые исключения и ограничения. В этом разделе мы проиллюстрируем, что это в точности означает, и какие заметные возможности не доступны через Питон (например, динамика жидкостей).

API Блендера состоит из трех основных областей интереса:

- Доступ к объектам Блендера и их свойствам, например объект *Camera* (Камера) и его свойство *angle* (угол) или объект *Scene* (Сцена) и его свойство *objects* (объекты)
- Доступ к операциям для выполнения, например добавление новой Камеры или рендеринг изображения
- Доступ к графическому интерфейсу пользователя, используя простое построение из блоков или взаимодействие с системой событий Блендера

Есть также несколько утилит, которые не попали ни в одну из этих категорий, так как они касаются абстракций, не имеющих прямого отношения к объектам Блендера, видимым конечному пользователю, например функции, манипулирующие векторами и матрицами.

## Много энергии

В совокупности это означает, что мы можем достичь многого из скриптов на Питоне. Мы можем:

- Создавать новый объект Блендера любого типа, включая камеры, лампы, меши, и даже сцены
- Взаимодействовать с пользователем с помощью графического интерфейса
- Автоматизировать общие задачи в Блендере, такие как, рендеринг
- Автоматизировать задачи поддержки вне Блендера, как, например, очистка каталогов
- Манипулировать любым свойством объекта Блендера, на который можно воздействовать с помощью API

Это последнее утверждение показывает одну из текущих слабостей API Блендера: любое свойство объекта, который разработчики добавляют Блендер в коде на С, должна предусматриваться отдельно в API на Питоне. Нет автоматического преобразования из внутренних структур в доступного на Питоне интерфейса, а это означает, что усилия должны дублироваться и это может привести к пропущенной функциональности. Например, в Блендере 2.49 невозможно задавать моделирование жидкостей из скриптов. Хотя возможно настроить систему частиц, нет возможности установить поведенческие характеристики системы частиц boids.

Другая проблема API Питона 2.49 - то, что многие из действий пользователя, которые можно выбрать для выполнения над объектом, не имеют эквивалента в API. Настройка простых параметров, как например, угла камеры или выполнение вращения любого объекта является легким и даже, например, применение модификатора *subsurface* к мешу - просто несколько строк кода, но обыкновенные действия, особенно над меш объектами, как например, подразделение выбранных ребер или выдавливание граней, отсутствуют в API и должны быть осуществлены разработчиком скрипта.

Эти проблемы заставили разработчиков Блендера полностью переработать API Питона Блендера для версии 2.5, уделяя особое внимание паритету (то есть, все возможное в Блендере должно быть возможно, используя API Питона). Это означает, что во многих ситуациях, должно быть значительно легче будет получить те же результаты в Блендере 2.5.

Наконец, Питон используется в большем количестве мест, чем просто в автономных скриптах: PyDrivers и PyConstraints позволяют нам управлять путём, которым ведут себя объекты Блендера, и мы столкнемся с ними в последующих главах. Питон также позволяет нам писать заказные текстуры и шейдеры как часть системы нодов, что мы увидим в *Главе 7, Создание заказных шейдеров и текстур*.

Также, важно иметь в виду, что Питон предлагает нам значительно больше, чем просто инструменты (уже впечатляющие), для автоматизации всех видов задач в Блендере. Питон является общим языком программирования с включенной расширенной библиотекой, так что мы не обязаны прибегать ко внешним инструментам для общесистемных задач, таких, как например, копирование файлов или архивации каталога. Даже сетевые задачи могут быть весьма легко реализованы, что доказывает множество предоставляемых рендер-ферм.

## Некоторые батарейки включены

Когда мы устанавливаем Блендер, интерпретатор Питона уже является частью приложения. Это означает, что нет необходимости устанавливать Питон как отдельное приложение. Но Питон это больше, чем просто интерпретатор. Питон поставляется с огромным набором модулей, которые обеспечивают массу функциональности. В нём доступно что-нибудь от манипуляции файлами до работы с XML и более, и лучше всего то, что эти модули являются стандартной частью языка. Они так же хорошо функционируют, как сам интерпретатор Питона и (с некоторыми исключениями), имеются на любой платформе, на которой работает Питон.

Обратная сторона этого - конечно, что этот набор модулей довольно большой (40MB или около того), так что разработчики Блендера решили распространять только чистый минимум, в первую

очередь математический модуль. Это имеет смысл, если Вы хотите держать размер Блендера приемлемым для загрузки. Многие разработчики на Питоне пришли к зависимости от стандартного дистрибутива, потому что при этом не приходится изобретать колесо, и это экономит огромное количество времени, не говоря уже, что это не простая задача - разработать и протестировать полноценную XML-библиотеку только потому, что вам нужно уметь читать простые XML-файлы. Вот почему теперь более или менее единодушно считается, что хорошо бы устанавливать полный дистрибутив Питона. К счастью, установка так же легка, как и установка самого Блендера, даже для конечных пользователей, так как бинарные инсталляторы предусмотрены для многих платформ, как например, Windows и Mac, а также в 64-битовых версиях. (Распространение для Linux предусмотрено в виде исходного кода с инструкциями о том как его компилировать, но множество дистрибутивов Linux или уже автоматически предоставляют Питон, или впоследствии установить его очень легко из пакетного репозитория).

## Проверьте наличие полного дистрибутива Питона

Есть шанс того, что у вас уже есть полный дистрибутив Питона в вашей системе. Вы можете проверить это, запустив Блендер и проверив консольное окно (термин консоли относится либо к DOS-окну, которое появится параллельно в Windows или окно X-терминала, если Вы запускаете Блендер в других системах), чтобы увидеть, отображает ли оно следующий текст:

```
Compiled with Python version 2.6.2.  
Checking for installed Python... got it!
```

Если это так, тогда Вы ничего не должны делать, и можете сразу перейти к разделу *Интерактивная консоль Питона*. Если появляется следующее сообщение, тогда Вы должны предпринять некоторые действия:

```
Compiled with Python version 2.6.2.  
Checking for installed Python... No installed Python  
found.
```

```
Only built-in modules are available. Some scripts  
may not run.
```

```
Continuing happily.
```

## Инсталлирование полного дистрибутива Питон

Шаги для полной установки Питона для Windows или Mac - следующие:

1. Загрузите подходящий инсталлятор со страницы <http://www.python.org/download/>. На момент написания, самая последняя стабильная 2.6 версия - это 2.6.2 (использованная в Блендере 2.49). Это обычно хорошая мысль - устанавливать самую последнюю стабильную версию, так как она будет содержать самые последние исправления. Убедитесь, тем не менее, что вы используете ту же мажорную версию, как и та, с которой скомпилирован Блендер. Будет правильным использовать версию 2.6.3, когда она будет выпущена, если Блендер скомпилирован с версией 2.6.2. Но если Вы используете более старую версию Блендера, который скомпилирован с Питоном 2.5.4, Вы должны установить самую последнюю 2.5.x версию Питона (или обновиться до Блендера 2.49, если есть возможность).
2. Запустите инсталлятор: В Windows он предложит Вам выбрать, куда установить Питон. Вы можете выбрать что-нибудь, что вам нравится, но если Вы выбираете заданное по умолчанию, Блендер почти несомненно найдёт установленные здесь модули без необходимости настраивать переменную *PYTHONPATH*. (смотри ниже)
3. (Пере) запустите Блендер. Консоль Блендера должна показать текст:

```
Compiled with Python version 2.6.2.  
Checking for installed Python... got it!
```

Если этого не произойдёт, вероятно, необходимо настроить переменную *PYTHONPATH*. Посмотрите вики Блендера для более подробной информации:

<http://wiki.blender.org/index.php/Doc:Manual/Extensions/Python>

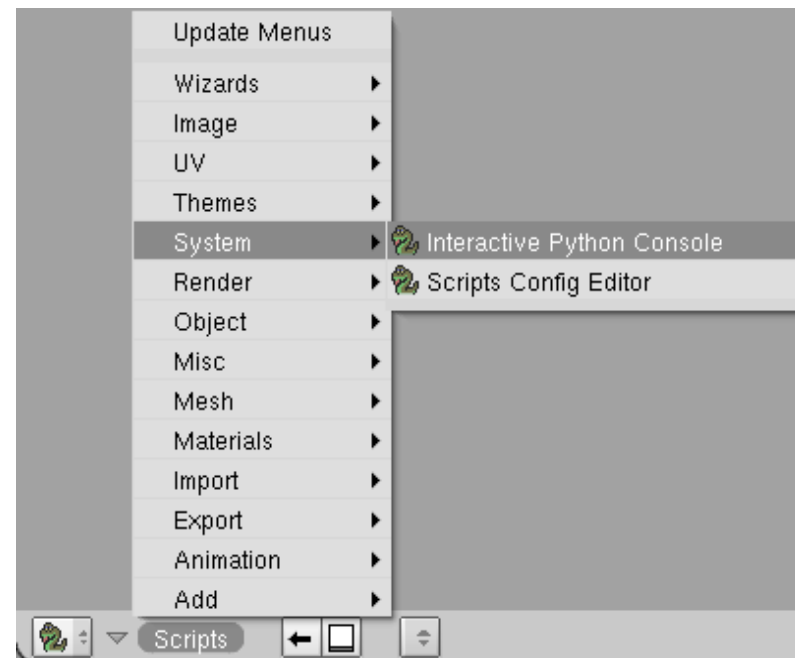
На Ubuntu Linux первый шаг не нужен, и установку можно провести посредством использования встроенного пакетного менеджера:

```
sudo apt-get update  
sudo apt-get install python2.6
```

Другие дистрибутивы могут использовать другую пакетную систему, так что Вам нужно посмотреть документацию об этом.

## Интерактивная консоль Питона

Чтобы увидеть, где Блендер действительно ищет модули, Вы можете посмотреть на переменную Питона *sys.path*. Чтобы сделать это, Вы должны запустить интерактивную консоль Питона в Блендере. Заметьте, что при этом используется другая и, возможно, здесь вы запутаетесь с понятиями консоли, - DOSBox или окно терминала, которое стартует одновременно с основным окном Блендера, и где появляются различные информационные сообщения также называется **консолью**! Интерактивная консоль Питона, которую мы хотим использовать, запускается в **окне скриптов**:



Как только интерактивная консоль Питона стартует, наберите следующие команды:

```
import sys
print sys.path
```

Заметьте, что в интерактивной консоли Питона не видно никаких подсказок (если только при этом не ожидается отступ, например в цикле *for*), но всё, что Вы набираете, будет отличаться цветом (белый на черном по умолчанию) от того, что возвращается (это будет синий или черный). Две предыдущие команды дадут нам доступ к модулю Питона *sys*, который содержит различные переменные с системной информацией. Переменная *sys.path*, которую мы печатаем, содержит все каталоги, в которых будет вестись поиск, когда мы попытаемся импортировать модуль. (Заметьте, что импортирование *sys* всегда будет работать, поскольку *sys* - встроенный модуль.) Результат будет чем-то, аналогичным этому:

```
['C:\\Program Files\\Blender Foundation\\Blender',
'C:\\Program Files\\Blender
Foundation\\Blender\\python26.zip', 'C:\\Python26\\Lib',
'C:\\Python26\\DLLs',
'C:\\Python26\\Lib\\lib-tk', 'C:\\Program Files\\Blender
Foundation\\Blender',
'C:\\Python26', 'C:\\Python26\\lib\\site-packages',
'C:\\Python26\\lib\\site-packages\\PIL',
'C:\\PROGRA~1\\BLENDE~1\\Blender',
'C:\\Documents and Settings\\Michel\\Application
Data\\Blender Foundation\\Blender\\.blender\\scripts',
'C:\\Documents and Settings\\Michel\\Application
Data\\Blender
Foundation\\Blender\\.blender\\scripts\\bpymodules']
```

Если ваш каталог установки Питона не присутствует в этом списке, тогда Вы должны настроить переменную PYTHONPATH перед запуском Блендера.

## Изучение встроенных модулей, функция *help()*

Интерактивная консоль Питона является также хорошей платформой для изучения встроенных модулей. Поскольку Питон поставляется оснащенным двумя очень полезными функциями, *help()* и *dir()*, у вас есть мгновенный доступ к большому количеству информации, содержащейся в модулях Блендера (и Питона), как к большой документации, предусмотренной в виде части кода.

Для людей, не знакомых с этими функциями, вот два коротких примера, оба работают из интерактивной консоли Питона. Для того, чтобы получить информацию о специфическом объекте или функции, наберите:

```
help(Blender.Lamp.Get)
```

Информация будет выведена в этой же консоли:

```
Help on built-in function Get in module Blender.Lamp:
```

```
Lamp.Get (name = None):
    Return the Lamp Data with the given name,
    None if not found, or Return a list with all
    Lamp Data objects in the current scene,
    if no argument was given.
```

### Перевод:

Помощь по встроенной функции Get в модуле Blender.Lamp:

```
Lamp.Get (name = None):
    Возвращает Данные лампы Lamp с именем name, None
    если не она обнаружена, или возвращает список со
    всеми объектами данных ламп в текущей сцене, если
    вызвана без аргумента.
```

Функция *help()* показывает связанную строку документирования функций, классов, или модулей. В предыдущем примере показана информация, предоставленная вместе с методом или функцией *Get()* класса *Lamp*. Строка документирования является первой строкой в определении функции, класса, или модуля. Когда вы определяете ваши собственные функции, было бы хорошо, если бы вы делали также. Это может выглядеть примерно так:

```
def square(x):
    """
    calculate the square of x.
    (вычисление квадрата x.)
    """
    return x*x
```

Мы можем теперь применить функцию помощи к нашей вновь определённой функции подобно тому, как мы делали прежде:

```
help(square)
```

На выходе появится:

```
Help on function square in module __main__:
```

```
square(x)
    calculate the square of x.
```

В программах, которые мы разработаем, мы будем использовать этот метод документирования, где это уместно.

## Изучение встроенных функций, функция `dir()`

Функция `dir()` выводит список все членов объекта. Этот объект может быть экземпляром, но также классом или модулем. Например, мы можем применить её к модулю `Blender.Lamp`:

```
dir(Blender.Lamp)
```

На выходе будет список всех членов модуля `Blender.Lamp`. Вы можете заметить функцию `Get()`, с которой мы столкнулись с ранее:

```
['ENERGY', 'Falloffs', 'Get', 'Modes', 'New', 'OFFSET',
 'RGB', 'SIZE', 'SPOTSIZE', 'Types', '__doc__',
 '__name__', '__package__', 'get']
```

Как только Вы выясните, что входит в состав класса или модуля, Вы можете затем выяснить наличие любой дополнительной информации помощи, применяя функцию `help()`.

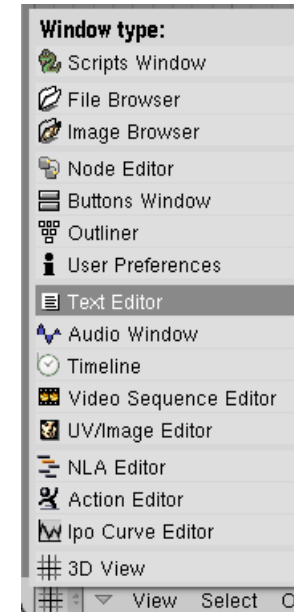
Конечно обе функции, как `dir()` так и `help()` - наиболее полезны, когда у вас уже есть какие-то мысли, где искать информацию. Но если это так, то они на самом деле могут быть очень удобными инструментами.

## Знакомство со встроенным редактором

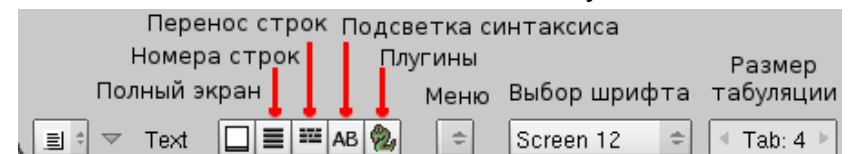
Вполне возможно использовать любой редактор (который вам нравится), чтобы писать скрипты на Питоне, а затем импортировать скрипты как текстовые файлы, но встроенного текстового редактора Блендера, вероятно, будет достаточно для всех потребностей программирования. Он представляет такие удобства, как, например, подсветка синтаксиса, нумерация строк и автоматический отступ, и дает Вам возможность запускать скрипты непосредственно из редактора. Способность запускать скрипт непосредственно из редактора даёт определенные блага при отладке из-за обратной связи, которую вы получите при возникновении ошибок. Вы не только получите информационное сообщение, но также ошибочная строка будет выделена в редакторе.

Более того, редактор поставляется с большим количеством плагинов, например, автоматическое дополнение или просмотр документации, которые очень удобны для программистов. И, конечно, можно написать дополнительные плагины самостоятельно.

Вы можете выбрать встроенный редактор, выбрав **Text Editor** из меню окон:



Когда Вы его запустите, вам будет предоставлена почти пустая область, за исключением полосы кнопок внизу:



Мы можем выбрать по умолчанию пустой текстовый буфер `TX:Text` или создать новый пустой текст, выбрав **ADD NEW** из выпадающего меню, появляющегося при щелчке на кнопку **Меню**.

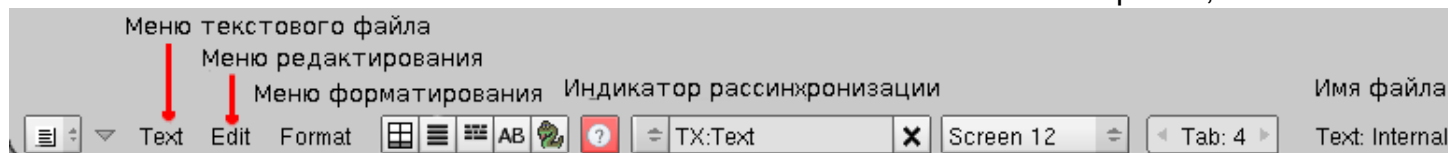
Имя по-умолчанию для этого нового текста будет `TX:Text.001`, но Вы можете изменить его на что-то более значимое, щелкнув по



имени, и исправив его. Заметьте, что если Вы хотели бы сохранить этот текст во внешний файл (с помощью **Text | Save As...**) имя текста станет отличаться от имени файла (хотя обычно было бы хорошей идеей держать имена одинаковыми, чтобы избежать неразберихи). Не обязательно сохранять тексты как внешние файлы; тексты являются объектами Блендера, которые сохраняются вместе со всей остальной информацией, когда Вы сохраняете ваш .blend файл.

Внешние файлы можно открыть как тексты, выбрав **OPEN NEW** из выпадающего **Меню** вместо **ADD NEW**. Если по некоторой причине внешний файл и связанный текст рассинхронизируются, когда запущен Блендер, отобразится кнопка рассинхронизации. Если по ней щелкнуть, отобразится множество опций для решения проблемы.

Как только новый или существующий текст будет выбран, зона меню внизу экрана немного изменится за счет дополнительных опций:



Меню текстового файла **Text** дает доступ к опциям, позволяющим открывать или сохранять файл, или запускать скрипт в редакторе. Оно также представляет множество шаблонных скриптов, которые можно использовать как основу для ваших собственных скриптов. Если Вы выберете один из этих шаблонов, будет создан новый текстовый буфер с копией выбранного шаблона.

Меню Редактирования **Edit** содержит функциональность вырезать-и-вставить, а также опции поиска и замены текста или быстрого перепрыгивания (jump) на выбранный номер строки.

Меню Форматирования **Format** имеет опции вставки и удаления отступов выбранного текста, а также опции для преобразования интервала. Последняя опция может быть очень полезной, когда интерпретатор Питона жалуется на неожиданные уровни отступа, хотя вам кажется, что в вашем файле нет никаких ошибок. Если это случилось, Вы, возможно, в какой-то момент смешали табуляцию и пробелы, что спутало Питон (поскольку они отличаются, как считает интерпретатор) и возможным выходом будет преобразовать

выбранный текст сначала в пробелы, а затем обратно в табуляцию. (Во всех книжках и руководствах по Питону настоятельно рекомендуют использовать для отступов НЕ табуляцию, а 4 пробела — дополнение пер.) Этот способ можно использовать снова, если когда-нибудь опять произойдет смешение пробелов и табуляции.

## Пример с редактором

Для того, чтобы привыкнуть к редактору, создайте новый текстовый буфер, выбрав **Text | New** и наберите следующие строки примера:

```
import sys
print sys.path
```

Большинство клавиш на клавиатуре ведут себя привычным образом, включая *Delete*, *Backspace*, и *Enter*. Клавиатурные

сокращения для вырезания, вставки, и копирования отображены в меню Редактирования как *Alt + X*, *Alt + V*, и *Alt + C* соответственно, но эквиваленты с клавишей *Ctrl*: *Ctrl + X*, *Ctrl + V*, и *Ctrl + C* (что привычно для пользователей Windows) работают так же хорошо. Полную карту клавиш можно посмотреть в Блендер-вики, [http://wiki.blender.org/index.php/Doc:Manual/Extensions/Python/Text\\_edit\\_or](http://wiki.blender.org/index.php/Doc:Manual/Extensions/Python/Text_edit_or)

Выбирать части текста можно щелчком и перетаскиванием мыши, но Вы можете также выбрать текст, перемещая текстовый курсор при нажатой клавише *Shift*.

Текст по умолчанию будет неокрашенным, но чтение скриптов можно сильно упростить для глаз, включив подсветку синтаксиса. Щелчок на небольшой кнопке **AB** переключает это (текст будет черно-белым при выключенной кнопке, и окрашенным при включенной.) Подобно многим аспектам Блендера, цвета текста можно модифицировать по желанию пользователя в секции **themes** (темы) окна **Пользовательских настроек**.



Другая возможность, которую очень удобно включить, особенно при отладке скриптов - нумерация строк. (Возможно, Вы способны написать безупречный код с первого раза, но, к несчастью, ваш покорный слуга не настолько гениален.) Каждое сообщение Питона об ошибке, которое будет показано, будет иметь имя файла и номер строки, и ошибочная строка будет выделена. Но строки вызывающих функций, если такие имеются, не будут выделены, хотя номера их строк будут показаны в сообщении об ошибке, так что наличие номеров строк включенными позволит вам быстро найти вызывающий контекст проблемного места. Нумерация строк включается щелчком по кнопке с **линиями**.

Запускается скрипт посредством нажатия *Alt + P*. Ничто не отобразится в редакторе, если программа не столкнется с ошибкой, но результат будет показан на консоли (то есть, в DOSBox'e или X-терминале, с которым стартует Блендер, *не* в интерактивной консоли Питона, с которой мы столкнулись ранее).

## ***Первые шаги: Hello world***

Традиция требует, чтобы все книги о программировании имели пример "hello world", и почему мы станем обижать людей? Мы осуществим и запустим скрипт, создающий, как иллюстрирующий пример, простой объект, и покажем как интегрировать этот скрипт в меню Блендера. Мы также покажем как, документировать его и сделать запись в справочной системе. Наконец, мы потратим несколько слов на аргументы за и против распространения скриптов в виде *.blend*-файлов, или в виде скриптов, которые пользователь сам должен устанавливать в *каталог со скриптами*.

Давайте напишем немного кода! Вы можете набрать следующие строки непосредственно в интерактивную консоль Питона, или Вы можете набрать новый текст в текстовом редакторе Блендера, и затем нажать *Alt + P*, чтобы запустить скрипт. Это - короткий скрипт, но мы пройдем через него довольно подробно, так как он отобразит множество ключевых аспектов API Питона в Блендере.

```
#!/BPY
```

```
import Blender
from Blender import Scene, Text3d, Window
```

```
hello = Text3d.New("HelloWorld")
hello.setText("Hello World!")
```

```
scn = Scene.GetCurrent()
ob = scn.objects.new(hello)
```

```
Window.RedrawAll()
```

Первая строка идентифицирует этот скрипт как скрипт Блендера. Она необязательна для запуска скрипта, но если мы хотим быть способными сделать этот скрипт частью структуры меню Блендера, нам она нужна, так что лучше мы будем привыкать к этому сразу.

Вы найдете вторую строку (которая выделена) фактически в любом скрипте Блендера, поскольку она дает нам доступ к классам и функциям API Питона в Блендере. Подобно ей, третья строка дает нам доступ к специфическим подмодулям модуля Blender, которые нам нужны в этом скрипте. Конечно, мы могли бы иметь доступ к ним как к членам модуля Blender (например, *Blender.Scene*), но явный импорт немного уменьшит количество программного текста и повысит удобочитаемость.

Следующие две строки сначала создают объект *Text3d* и назначают его переменной *hello*. Объект *Text3d* будет иметь в Блендере имя *HelloWorld*, так что пользователи могут ссылаться на этот объект по этому имени. Также это имя, которое будет видно в окне *Outliner*, и в левом нижнем углу, если выбрать объект. Если там уже существует объект того же самого типа с этим именем, Блендер добавит цифровой суффикс к имени, чтобы сделать его уникальным. Например, *HelloWorld* мог бы стать *HelloWord.001*, если мы запустим этот скрипт дважды.

По умолчанию, вновь созданный объект *Text3d* будет содержать текст **Text**, так что мы изменяем его на **Hello World!** с помощью метода *setText()*.

Вновь созданный в Блендере объект не видим по умолчанию, мы должны соединить его со Сценой, так что несколько следующих строк извлекают ссылку на текущую сцену и добавляют объект *Text3d* в неё. Объект *Text3d* не добавляется непосредственно к сцене, но метод *scene.objects.new()* вставляет объект *Text3d* в общий (generic) объект Блендера и возвращает ссылку на последний. Общий объект Блендера хранит информацию, общую для всех объектов, такую как позиция, в то время как объект *Text3d* хранит специфическую информацию, как, например, шрифт текста.

Наконец, мы сообщаем оконному менеджеру обновить все окна, это обновление нужно из-за добавления нового объекта.

## Внедрение скрипта в меню Блендера

Ваш собственный скрипт не должен быть гражданином второго сорта. Его можно сделать частью Блендера наравне с любым из скриптов, которые поставляются с Блендером. Его можно добавить к меню **Add** (добавить) в заголовке наверху окна 3D-вида.



На самом деле меню **Add** присутствует в заголовке внизу окна пользовательских настроек, но так как это окно расположено выше окна 3D-вида, и по умолчанию минимизировано именно до заголовка, оно выглядит так, как будто оно является заголовком сверху окна 3D-вида. Множество пользователей так привыкли к этому, что они видят его как часть окна 3D-вида.

Он может предоставить информацию системе помощи Блендера также, как любой другой скрипт. Следующие несколько строк кода делают это возможным:

```
"""
```

```
Name: 'HelloWorld'
```

```
Blender: 249
```

```
Group: 'AddMesh'
```

```
Tip: 'Create a Hello World text object'
```

```
"""
```

Мы запускаем скрипт с автономным строковым объектом, фактически состоящим из нескольких строк.



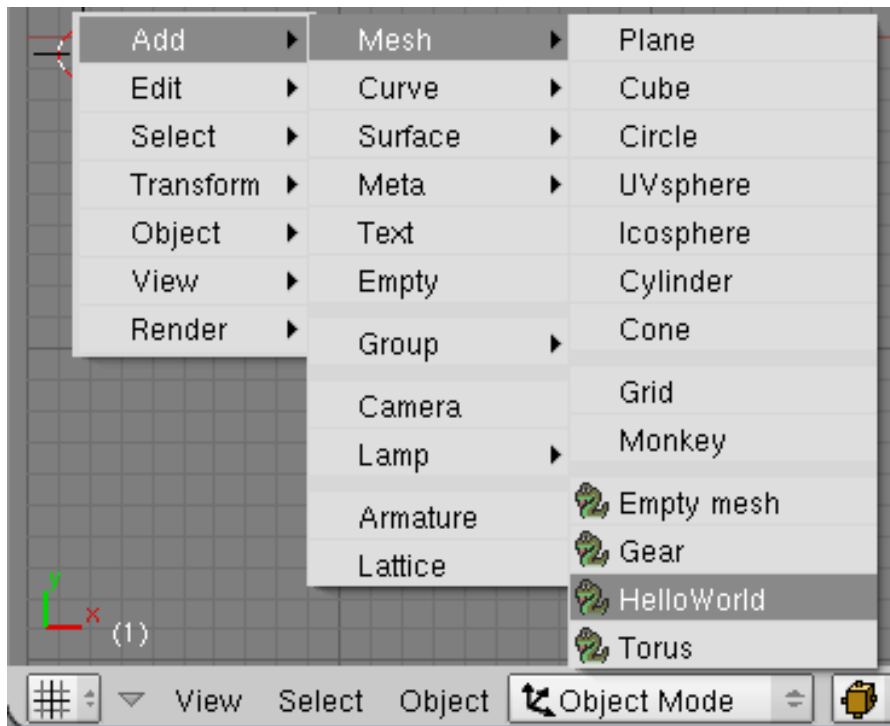
Каждая строка начинается с метки, сопровождаемой двоеточием и значением. Двоеточие должно следовать за меткой немедленно. Там не должно быть никаких промежуточных пробелов, в противном случае наш скрипт не появится ни в каком меню.

Метки в начале каждой строки служат следующим целям:

- *Name* (строка) определяет имя скриптов так, как они появятся в меню
- *Blender* (число) определяет минимально необходимую версию Блендера для использования скрипта
- *Group* (строка) - подменю в меню скриптов, под которым этот скрипт должен быть сгруппирован

Если наши скрипты должны появиться в меню **Add | Mesh** в окне 3D-вида (также доступного по нажатию *Пробела*), этот параметр должен быть *AddMesh*. Если было бы нужно другое подменю в меню скриптов, то тут могло бы быть, например, *Wizards* или *Object*. Кроме необходимых меток, могут быть добавлены следующие дополнительные метки:

- *Version* (строка) - версия скрипта в любом предпочитаемом вами формате.
- *Tip* (строка) — информация, показываемая в подсказке, появляющейся над пунктом меню в меню **Скриптов**. Если скрипт принадлежит группе *AddMesh*, никаких подсказок показываться не будет, даже если мы определим её здесь.



## Внедрение скрипта в систему помощи Блендера

Блендер имеет встроенную систему подсказки, которая доступна в меню Help наверху экрана. Оно дает доступ к онлайн-ресурсам и к информации о зарегистрированных скриптах через пункт **Scripts Help Browser** (браузер помощи по скриптам). Если его выбрать, появятся выпадающие меню для каждой группы, где Вы можете выбрать скрипт и посмотреть его информацию помощи.

Если мы хотим ввести наш скрипт во встроенную систему помощи, нам нужно определить несколько дополнительных глобальных переменных:

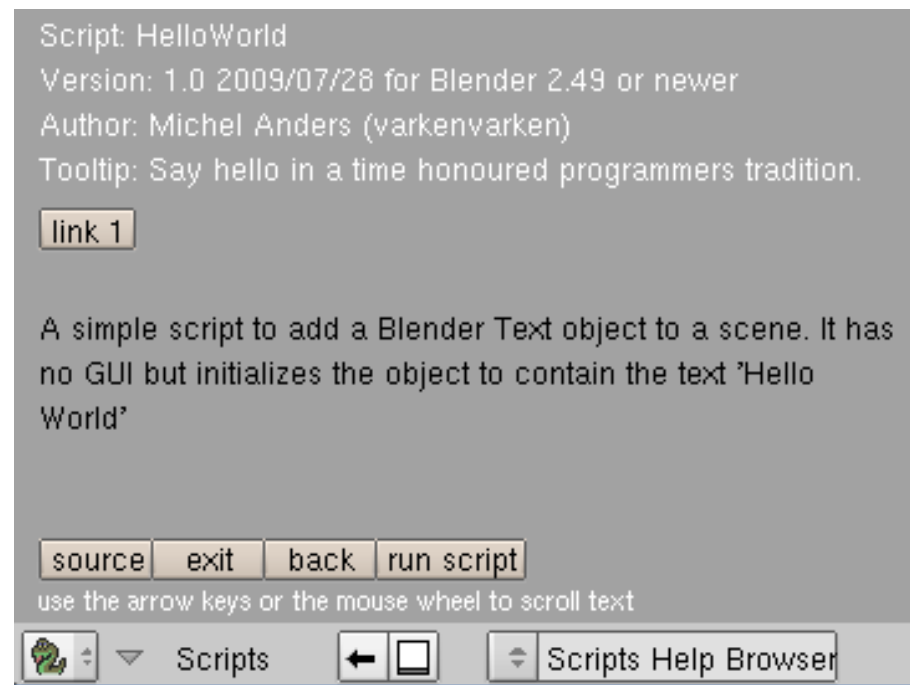
```
__author__ = "Michel Anders (varkenvarken)"
__version__ = "1.00 2009/08/01"
__copyright__ = "(c) 2009"
__url__ = ["author's site,",
http://www.swineworld.org"]
__doc__ = ""
```

A simple script to add a Blender Text object to a scene. It takes no parameters and initializes the object to contain the text 'Hello World'

```
"""
```

*Перевод строки \_\_doc\_\_:* Простой скрипт для добавления текстового объекта Блендера в сцену. Он не принимает никаких параметров и инициализирует объект, содержащий текст 'Hello World'

Эти переменные не требуют разъяснений, за исключением переменной `__url__`, - она принимает список строк, где каждая строка состоит из короткого описания, запятой, и ссылки. В результате экран помощи будет выглядеть похожим на это:



Теперь все, что нам осталось сделать, это проверить его и поместить этот скрипт в нужном месте. Мы можем протестировать скрипт, нажав **Alt + P**. Если не столкнулись ни с какими ошибками, результатом будет наш объект *Hello World Text3d*, добавленный к сцене, но скрипт не будет пока добавлен в меню **Add**.

Если скрипт должен быть добавлен к меню **Add**, он должен находиться в каталоге скриптов Блендера. Для того, чтобы сделать это, сначала сохраните скрипт, находящийся в текстовом буфере, в файл со значимым именем. Затем, убедитесь, что этот файл расположен в каталоге скриптов Блендера. Этот каталог называется *scripts* и является подкаталогом в *.blender*, каталоге конфигурации Блендера. Он расположен в каталоге установки Блендера или (на Windows) в каталоге *Application Data*. Простейший способ найти свой - просто посмотреть снова на переменную *sys.path*, чтобы увидеть, куда указывает каталог, заканчивающийся на *.blender\scripts*.

Скрипты, расположенные в каталоге скриптов Блендера, автоматически будут выполнены при запуске, так что наш скрипт *hello world* будет доступен в любое время, когда мы запустим Блендер. Если мы хотим, чтобы Блендер пересмотрел каталог скриптов (чтобы нам не пришлось перезапускать Блендер для появления нашего нового дополнения), мы можем выбрать **Scripts | Update Menus** в интерактивной консоли.

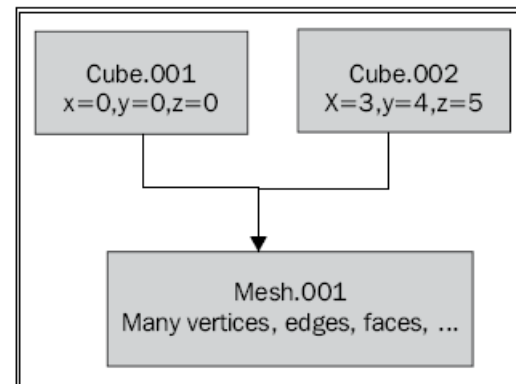
## Не запутайтесь, оставайтесь объективным

Как Вы могли обратить внимание, слово **объект** используется в двух различных (возможно, запутанно) случаях. В Блендере почти все называется *Object*. Лампа (*Lamp*), например - *Object*, но точно так же им являются Куб (*Cube*) или Камера (*Camera*). Объекты - вещи, которыми может манипулировать пользователь, и они имеют, например, позицию и вращение.

Фактически, всё несколько более структурировано (или усложнено, как говорят некоторые): любой объект Блендера содержит ссылку на более специфический объект, называемый **блок данных (data block)**. Когда Вы добавляете объект *Куб* в пустую сцену, у вас будет общий (*generic*) объект в некоторой позиции. Этот объект будет назван *Cube* и будет содержать ссылку на другой объект, *Меш* (*Mesh*). Это *Меш*-объект также будет назван по умолчанию *Cube*, но это - нормально, так как пространства имён объектов различного типа разделены.

Это разделение свойств на общие для всех объектов (как например, позиция) и специфические свойства в единственном типе

объекта (например, энергия *Лампы* или вершины *Меша*) - логичный способ упорядочить наборы свойств. Это также позволяет экземпляру иметь множество копий объекта, не поглощая много памяти; мы можем иметь более, чем один объект, указывающий на один и тот же объект *Меша*, например. (Способ создать **связанный дубликат** - использовать *Alt + D*.) Следующая диаграмма может помочь понять эту концепцию:



Другой путь использования слова **объект** - в понимании Питона. Здесь мы подразумеваем экземпляр класса. API Блендера объектно-ориентированное и почти каждая возможная часть структурных данных представлена экземпляром объекта класса. Даже довольно абстрактные понятия, как например, **Действие (Action)** или **IPO** (абстрактное в смысле, что у них нет позиции где-нибудь на вашей сцене), определены как классы.

На какое значение слова объект мы ссылаемся в данный момент, в понимании Блендера или Питона, в этой книге по большей части будет очевидным из контекста, если иметь в виду это различие. Но если нет, мы будем стремиться писать - в понимании Блендера как *Объект (Object)* или в понимании Питона *объект (object)* или *экземпляр объекта (object instance)*.

*По-моему, автор сам ввёл лишнюю путаницу в терминологию, называя блоки данных типа Mesh или Lamp объектами. Проще было бы сразу вводить различные термины и применять их всё время: Объекты, Блоки данных, Экземпляры классов. - недоумение пер.*

## Добавление различных типов объектов из скрипта

Добавление других типов объектов, во многих случаях, так же просто, как добавление нашего текстового объекта. Если мы хотим, чтобы наша сцена была заполнена таким образом, чтобы её можно было отрендерить, то мы должны добавить камеру и лампу, чтобы делать вещи видимыми. Добавление камеры на ту же сцену можно сделать подобно этому (предположим, что у нас все еще есть ссылка на нашу активную сцену в переменной `scn`):

```
from Blender import Camera
cam = Camera.New() # создаёт новый блок данных камеры
ob = scn.objects.new(cam) # добавляет новый объект
                                # камеры
scn.setCurrentCamera(ob) # делает эту камеру активной
```

Заметьте, что объект Камеры снова отличается от фактических данных камеры. Объект *Camera* содержит данные, специфичные для камеры, например, угол обзора, а объект Блендера содержит данные, общие для всех объектов, особенно позицию (местоположение) и вращение. Мы позже снова столкнемся с камерами, и увидим как мы можем указать им и установить угол обзора.

Лампы абсолютно также следуют за этим образцом:

```
from Blender import Lamp
lamp = Lamp.New() # создаёт новую лампу
ob = scn.objects.new(lamp)
```

Снова, объект *Lamp* содержит данные, специфичные для лампы, как например, тип (например, `spot` или `area`) или энергия, в то время как объект Блендера инкапсулирует заданные ему позицию и вращение.

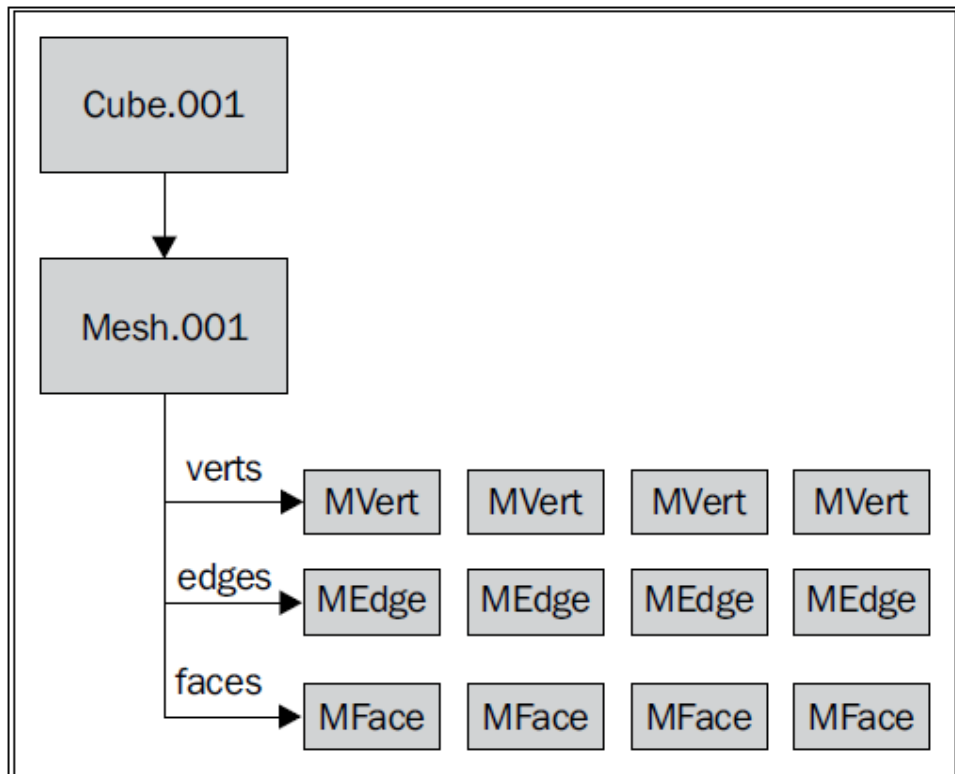
Этот образец аналогичен для объекта Меша, но ситуация здесь тонко отличается, поскольку меш - это конгломерат вершин, рёбер, и граней среди других свойств.

## Добавление меш-объекта

Подобно Лампе или Камере, Меш является объектом Блендера, который изолирует другой объект, в данном случае, объект *Blender.Mesh*. Но в отличие от объектов *Blender.Lamp* или *Blender.Camera*, на этом всё не заканчивается. Объект *Blender.Mesh* сам может содержать множество других объектов. Эти объекты — вершины (vertices), рёбра (edges) и грани (faces). Каждый из них может иметь множество связанных свойств. Они могут быть выбраны или спрятаны, и могут иметь поверхностную нормаль или ассоциированную UV-текстуру.

За исключением всех связанных свойств, единичная вершина является в основном точкой в 3D-пространстве. В объекте *Blender.Mesh* любое количество вершин организовано в списке объектов *Blender.Mesh.MVert*. В полученном меш-объекте *me* этот список может быть доступен как *me.verts*. **Ребро** является линией, соединяющей две вершины в Блендере и представлено объектом *Blender.Mesh.MEdge*. Его основные свойства - это *v1* и *v2*, которые являются ссылками на объекты *MVert*. Список рёбер в Меш-объекте может быть доступен как *me.edges*.

Объект грани *MFace* похож на ребро, в основном это список ссылок на вершины, которые определяют его. Если у нас есть *MFace*-объект *face*, этот список может быть доступен как *face.verts*.



Эта путаница объектов, содержащих другие объекты, может вызвать неразбериху, так что держите предыдущую диаграмму в уме, и давайте посмотрим на некоторый пример кода, чтобы разъяснить это. Мы определим куб. Куб состоит из восьми вершин, связанных двенадцатью рёбрами. Восемь вершин также определяют шесть сторон (или граней) куба. (*corners переводится как углы, sides — стороны — прим. пер.*)

```
from Blender import Mesh, Scene
```

```
corners=[ (-1,-1,-1), (1,-1,-1), (1,1,-1), (-1,1,-1),
           (-1,-1, 1), (1,-1, 1), (1,1, 1), (-1,1,1) ]
sides= [ (0,1,2,3), (4,5,6,7), (0,1,5,4), (1,2,6,5),
         (2,3,7,6), (3,0,4,7) ]
```

```
me = Mesh.New('Cube')
me.verts.extend(corners)
me.faces.extend(sides)
```

```
scn = Scene.GetCurrent()
ob = scn.objects.new(me, 'Cube')
Window.RedrawAll()
```

Мы начинаем с определения списка углов. Каждый из восьми углов представлен кортежем трех чисел - это координаты x, y, и z. Затем мы определяем список кортежей, задающих грани куба. Стороны куба являются квадратами, так что каждый кортеж содержит четыре целых - каждое целое является индексом в списке углов. Важно получить эти индексы в правильном порядке: если мы захотим указать первую сторону как (0,1,3,2), мы получим грань, искривленную, как галстук-бабочка.

Теперь мы можем определить Меш-объект и назвать его *Cube* (выделенная часть в предыдущем коде). Как отмечено раньше, вершины Меш-объекта доступны как список с именем *verts*. Он имеет метод *extend()*, который может взять список кортежей, представляющих позиции вершин, чтобы определить дополнительные объекты *MVert* в нашем Меше.

Точно так же мы можем добавить дополнительные грани к списку граней *faces* Меш-объекта, вызывая его метод *extend()* со списком кортежей. Поскольку все рёбра куба являются рёбрами граней, нет необходимости добавлять какие-либо рёбра отдельно. Это произойдёт автоматически, когда мы применяем *extend()* к списку граней.

Меш-объект, который мы определили, теперь можно вставить в объект Блендера, который может быть добавлен к активной сцене. Заметьте, что вполне допустимо иметь Меш-объект и Объект Блендера с одинаковым именем (*Cube* в данном случае), поскольку различные типы объектов в Блендере имеют отдельные пространства имён. В графическом интерфейсе пользователя Блендера имена всегда имеют двухбуквенный префикс, чтобы различать их. (например, LA для лампы, ME для меша, или OB для объекта Блендера)

При создании Меш-объекта много внимания нужно уделять всем добавляемым вершинам, рёбрам и граням, и правильно их нумеровать. Это только вершина айсберга при создании мешей. В



Главе 2, Создание и Редактирование Объектов, мы увидим, что прячется под водой.

## Распространение скриптов

В предыдущих секциях мы видели, что для того, чтобы внедрить наш скрипт в систему меню и систему помощи Блендера, мы должны расположить скрипт в каталоге `.blender/scripts`. Полностью интегрированный скрипт может быть большим преимуществом, но этот метод имеет очевидный недостаток: человек, который хочет использовать этот скрипт должен разместить его в правильном каталоге. Это может быть проблемой, если этот человек не знает, где расположен этот каталог или не имеет разрешения устанавливать скрипты в этом каталоге. Эту последнюю проблему можно преодолеть, настроив альтернативный каталог скриптов в *Пользовательских Настройках*, но не каждый может быть настолько технически подкованным.

Жизнеспособной альтернативой этому может быть распространение скриптов в виде текста внутри `.blend` файла. `.blend` файл может быть сохранен со скриптом, ясно видимым в главном окне, и одна из первых строк комментария скрипта, вероятно, может выглядеть так *"Press ALT-P to start this script"* (нажмите ALT-P для запуска скрипта). Этим способом скрипт сможет использовать любой, кто знает, как открывать `.blend` файл.

Дополнительным преимуществом является то, что при этом можно легко упаковать дополнительные ресурсы в тот же `.blend` файл. Например, скрипт может использовать определенные материалы или текстуры, или Вы можете захотеть включить образец результата вашего скрипта. Единственная вещь, которая очень трудна - распространять таким образом модули Питона. Вы можете использовать оператор `import`, чтобы получить доступ к другим текстовым файлам, но это может вызвать проблемы (смотри Приложение В). Если у вас есть много кода и он организован в модулях, Вам и вашим пользователям, вероятно, будет лучше, если Вы станете распространять всё в виде ZIP-файла с ясными инструкциями, куда нужно распаковывать этот ZIP-файл.

Для Pynodes (или динамических нодов, смотри Главу 7) у вас нет выбора. **Pynodes** могут ссылаться только на код Питона, содержащийся в текстах внутри `.blend` файла. На самом деле это не является ограничением, так как эти Pynodes - неотъемлемая часть материала, а материалы Блендера могут распространяться только внутри `.blend` файла. Когда эти материалы привязаны или добавлены к связанным с ними нодами, то любые тексты, ассоциированные с Pynodes, привязываются или добавляются также, полностью скрываясь от конечного пользователя через материал, который на самом деле создаётся.

## API Блендера

При разработке программ на Питоне в Блендере важно понимать, какие функции обеспечиваются API, а тем более, какие нет. API, в основном, даёт доступ ко всем данным и предоставляет функции для манипуляции этими данными. К тому же, API обеспечивает разработчика функциями для рисования на экране и для взаимодействия с интерфейсом пользователя и оконной системой. Что API Блендера не предоставляет - это объектно-специфическую функциональность, кроме присваивания простых свойств, особенно недостаёт всех функций, манипулирующих мешами на уровне вершин, рёбер и граней, кроме как добавления или удаления их.

Это означает, что очень высокоуровневые или сложные задачи, как например, добавление модификатора `subsurface` на объект Меша или отображение диалога выбора файлов, так же просто, как написание одной строки кода, тогда как важнейшие и, видимо, простые функции, такие как подразбиение (`subdividing`) ребра или выбор рёберного цикла не доступны. Это не означает, что эти задачи нельзя выполнить, но мы должны программировать их самостоятельно. Так много примеров в этой книге ссылается на модуль, называемый *Tools*, который мы разработаем в следующих главах, и который будет содержать полезные инструменты от выдавливания граней до замыкания циклов. Где это необходимо и интересно, мы осветим код этого модуля, но, главным образом, он предназначен иметь запас всего того кода, который мог бы увести нас от наших целей.

Следующие разделы дают короткий и очень поверхностный обзор того, что доступно в API Блендера. Множество модулей и утилит будут занимать важное место в следующих главах, когда мы будем разрабатывать практические примеры. Этот обзор предназначен в качестве средства, которое поможет вам начать работу, если вы хотите узнать о некоторых функциональных возможностях и не знаете, где искать в первую очередь. Это далеко не полный список документации по API Блендера. Для этого, проверьте наиболее последнюю версию документации онлайн-API. Вы можете найти ссылку в *Приложении А Ссылки и Ресурсы*.

## Модуль Blender

Модуль Blender служит в качестве контейнера для большинства других модулей и обеспечивает функциональность доступа к системной информации и выполнению общих задач.

Например, такая информация, как версия Блендер, которую вы используете, может быть извлечена с помощью функции *Get()*:

```
import Blender
version = Blender.Get('version')
```

Включение всех внешних связанных файлов в *.blend* файл (в Блендере называемое **упаковкой**) или сохранение вашего текущего сеанса Блендера в *.blend* файл - другие примеры функциональности, выполняемой в модуле Блендера верхнего уровня:

```
import Blender
Blender.PackAll()
Blender.Save('myfile.blend')
```

## Объекты Блендера

Каждый тип объекта Блендера (*Object*, *Mesh*, *Armature*, *Lamp*, *Scene* и так далее), имеют связанный с ним модуль, который является подмодулем модуля *Blender* верхнего уровня. Каждый модуль предоставляет функции для создания новых объектов и поиска объектов данного типа по имени. Каждый модуль имеет также определённый класс с тем же именем, который осуществляет функциональность, связанную с объектом Блендера.

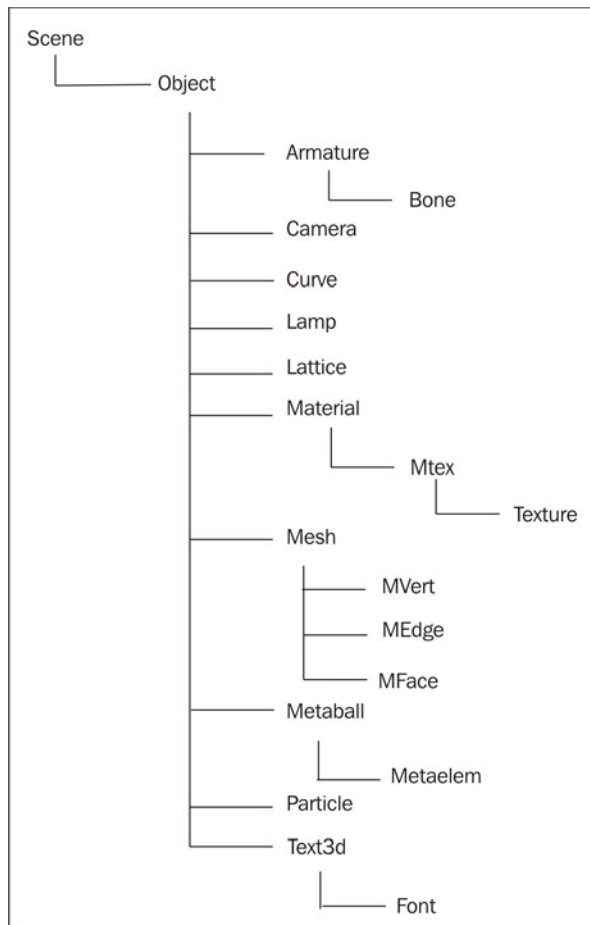
Заметьте, что в Блендере есть сущности, не только непосредственно видимые на вашей сцене, как например, меши, лампы, или камеры - объекты, но также материалы, текстуры, системы частиц, и даже IPO, действия (Actions), миры, и сцены.

Множество других видов данных в Блендере - не являются Объектами в понимании Блендера (Вы не можете добавить их из другого *.blend* файла или перемещать их по вашей сцене), но это объекты в понимании Питона. Например, вершины, рёбра, и грани внутри меша выполнены в виде классов: *Blender.Mesh.MVert*, *Blender.Mesh.MEdge*, и *Blender.Mesh.MFace* соответственно.

Множество модулей также имеют свои собственные подмодули; например Модуль *Blender.Scene* предоставляет доступ к контексту рендера посредством модуля *Blender.Scene.Render*. Между прочим, этот модуль определяет класс *RenderData*, который позволяет Вам рендерить неподвижное изображение или анимацию.

Таким образом, с тем, что мы теперь знаем, можно нарисовать два немного различных родословных дерева объектов Блендера.

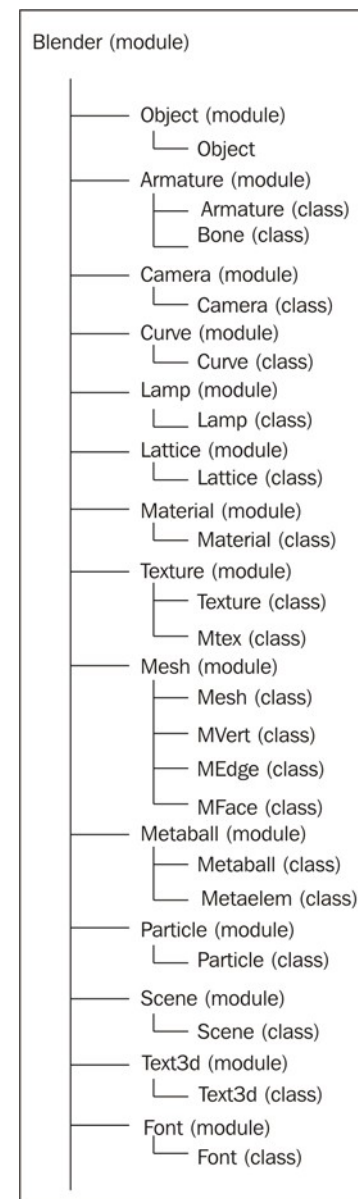




Первая иллюстрация показывает, какой тип объектов Блендера может содержаться внутри или ссылаться на другой объект Блендера, в ней мы ограничимся менее абстрактными объектами:

Конечно, диаграмма выше сильно упрощена, так как мы пропустили некоторые менее важные объекты, и так как она иллюстрирует только единственный тип отношений. Конечно, существует намного больше типов отношений в сцене, такие как, например, отношения родитель-ребенок или ограничения (constraints).

Мы можем сопоставить предшествующую диаграмму со следующей, которая показывает в каком модуле какой тип объекта (класс) определён:



Различия весьма заметны, и их важно иметь в виду, особенно при поиске конкретной информации, содержащейся в документации Blender API. Не ожидайте, что найдёте информацию об объекте *Curve* (Кривая) в документации о модуле *Blender.Object*, поскольку кривая в Блендере является специфическим объектом Блендера; класс *Curve* определён и документирован в Модуле *Blender.Curve*. В целом можно ожидать, что документация класса находится в модуле тем же названием.

## Модуль bpy

Кроме модуля Blender, есть другой модуль верхнего уровня, с именем *bpy*, который обеспечивает унифицированный путь доступа к данным. Он считается экспериментальным, но он стабилен и может быть использован как более интуитивный путь доступа к объектам. Например, если мы хотим иметь доступ к объекту с именем *MyObject*, обычным образом мы должны действовать приблизительно так:

```
import Blender
ob = Blender.Object.Get(name='MyObject')
```

С модулем *bpy* мы можем перефразировать это так:

```
import bpy
ob = bpy.data.objects['MyObject']
```

Так же, чтобы получить доступ к активному объекту сцены, мы могли бы написать это:

```
import Blender
scene = Blender.Scene.GetCurrent()
```

Что можно записать альтернативным способом:

```
import bpy
scene = bpy.data.scenes.active
```

Что из них предпочитать - дело вкуса. Модуль *bpy* будет единственным способом доступа к данным в ожидаемом Блендере 2.5, но изменения в Блендере 2.5 проникают глубже, чем просто такой способ доступа к данным, так не обманитесь поверхностным сходством имени модулей!

## Рисование на экране

Доступ к системе окон Блендера предоставлен модулем *Blender.Draw*. Здесь Вы найдёте классы и функции для определения кнопок и управляющих меню, и пути взаимодействия с пользователем. Типы графических элементов, которые Вы можете отобразить, используя модуль *Draw*, ограничены обычно используемыми, и модификации невозможны.

Более передовые функции предоставлены в модуле *Blender.BGL*, который даёт Вам доступ фактически ко всем функциям и константам OpenGL, позволяющим Вам рисовать на экране почти всё, что угодно, и позволить взаимодействовать с пользователем множеством различных способов.

## Утилиты

Наконец, есть множество модулей, включающих различную функциональность, которые не подходят для любой из предыдущих категорий:

- *Blender.Library*: Блендер позволяет Вам добавлять (то есть, импортировать) или связывать (link) объекты из другого *.blend* файла. Можно посмотреть на это по-другому - *.blend* файл может действовать как библиотека, где Вы можете сохранять ваши активы. И поскольку почти всё является объектом в Блендере, почти любой актив может быть сохранен в такой библиотеке, будь это модели, лампы, текстуры, или даже полные сцены. Модуль *Blender.Library* предоставляет авторам скриптов средства получить доступ к этим библиотекам.
- *Blender.Mathutils* и *Blender.Geometry*: Эти модули содержат, кроме прочего, классы *Векторов (Vector)* и *Матриц (Matrix)* со связанными с ними функциями, для применения всех видов векторной алгебры к объектам Блендера. С функциями, приведенными в этих модулях, Вы будете способны вращать или сдвигать координаты ваших объектов или вычислять угол между двумя векторами. Предусмотрено намного больше удобных функций, и они, часто неожиданно, будут появляться в примерах в этой книге. Не беспокойтесь, мы приведём

объяснения, где это будет необходимо, для людей, находящихся не в своей тарелке от векторной математики.

- *Blender.Noise*: Шум (Noise) используется в генерации всех (очевидно) случайных образцах, которые формируют основу многих процедурных текстур в Блендере. Этот модуль дает доступ к тем же программам, которые обеспечивают шум для этих текстур. Это может быть полезным не только в генерации ваших собственных текстур, но можно, например, использовать при произвольном размещении объектов, или осуществлении немного шаткого пути камеры, чтобы добавить реализма к вашей анимации.
- *Blender.Registry*: данные в скриптах, неважно, локальные или глобальные, не сохраняются при выходе из скрипта. Это может быть очень неудобным, например, если Вы хотите сохранить пользовательские настройки для вашего заказного скрипта. Модуль *Blender.Registry* обеспечивает способ сохранять и извлекать постоянные данные. Тем не менее, он не обеспечивает никаких средств сохранения этих данных на диске, так что это постоянство действует только в течение сеанса Блендера.

- *Blender.Sys*: По словам документации этого модуля:

```
This module provides a minimal set of helper functions and data. Its purpose is to avoid the need for the standard Python module os in special os.path, though it is only meant for the simplest cases.
```

Перевод:

Этот модуль обеспечивает минимальный набор вспомогательных функций и данных. Его цель в том, чтобы избежать потребности в стандартном модуле Питона *os* и его подмодуле *os.path*, но все же, он предназначен только для самых простых случаев.

Как мы аргументировали раньше, обычно рекомендуется устанавливать полный дистрибутив Питона, который, кроме прочего, включает модули *os* и *os.path*, они дадут Вам доступ к более широкому диапазону функциональности. Следовательно, мы не используем модуль *Blender.sys* в этой книге.

- *Blender.Types*: Этот модуль предоставляет константы, которые могут быть использованы для проверки типа объектов. Встроенная функция Питона *type()*, возвращает тип своего аргумента. Это позволяет очень легко проверить объект данного типа по сравнению с одной из констант в этом модуле.

Если мы хотим убедиться что некий объект - это объект *Curve*, мы можем, например, сделать это так:

```
...
if type(someobject) == Blender.Types.CurveType :
    ... сделать что-то, доступное только для объектов Curve ...
```

## Итог

В этой главе, мы увидели как расширять Блендер с помощью полного дистрибутива Питона и познакомились со встроенным редактором. Это позволило нам написать скрипт, хотя и простой, полностью интегрировать его в меню скриптов Блендера и систему помощи. Мы охватили множество моментов, а именно:

- Что возможно и не возможно выполнить с помощью Питона в Блендере
- Как проинсталлировать дистрибутив Питона
- Как использовать встроенный редактор
- Как запускать скрипт на Питоне
- Как изучать встроенные модули
- Как написать простой скрипт, который добавляет объект в сцену
- Как зарегистрировать скрипт в меню скриптов Блендера
- Как документировать ваш скрипт дружественным к пользователю способом
- Как распространять скрипт

В следующей главе мы сделаем шаг в направлении создания и редактирования сложных объектов, а также графического интерфейса пользователя.

## Создание и редактирование объектов

В некотором смысле, меши - наиболее важный тип объектов в 3D-приложении. Они лежат в основе большинства видимых объектов и являются сырьём, которое может быть оснащено (rigged) и анимировано в дальнейшем. В этой главе речь идет о создании мешей и способах манипулировать меш-объектом, как целиком, так и его индивидуальными сущностями, из которых он состоит - вершинами, рёбрами и гранями.



В этой главе вы изучите:

- Как создавать конфигурируемые меш-объекты
- Как разрабатывать графический интерфейс пользователя
- Как заставить ваш скрипт сохранять выбранные пользователем настройки для последующего многократного использования
- Как выбирать вершины и грани в меше
- Как сделать один объект родителем другого
- Как создавать группы

- Как модифицировать меши
- Как запускать Блендер с командной строки и рендерить в фоновом режиме
- Как обрабатывать параметры командной строки

### ***Creepy crawlies (ползучий ужас) - графический интерфейс пользователя для конфигурирования объектов***

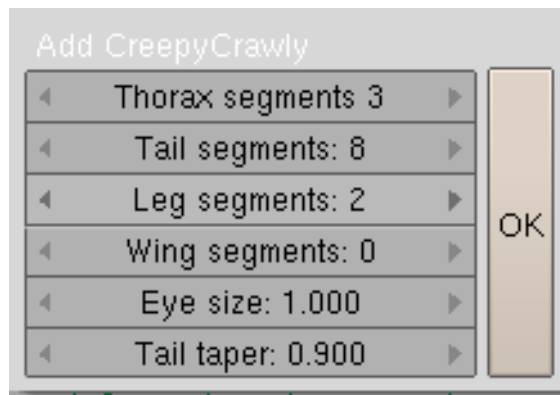
Иллюстрирование примером создания единственной копии одноразового объекта Блендера (подобно сделанному нами в примере "hello world" в Главе 1, Расширение Блендера с помощью Питона), может быть хорошим упражнением по программированию, но скрипт создания объекта действительно вступает в свои права, когда встроенных методов, таких, как например, копирование объектов, или модификаторов, как например, модификатор array - не достаточно.

Хороший пример - такой, где мы хотим создать один или много вариантов объекта, и эти варианты должны легко конфигурироваться конечным пользователем. Например, гайки и болты бывают различных форм и размеров, так что в Блендер включен скрипт для их создания. В Сети доступно намного больше скриптов, создающих что-нибудь от механических механизмов до лестниц, от деревьев до церковных куполов.

В этом разделе мы покажем, как построить маленькое приложение, которое может создать все виды жуко-подобных существ и поставляется с простым, но эффективным интерфейсом, настраивающим множество параметров. Это приложение также сохраняет пользовательские настройки для более позднего повторного использования.

## Создание интерфейса пользователя

Проектирование, строительство, и тестирование графического пользовательского интерфейса могут быть пугающими задачами, но API Блендера предоставляет нам инструменты, позволяющие сделать это намного легче. Модуль *Blender.Draw* обеспечивает простые, но часто используемые и легко конфигурируемые компоненты для быстрого определения пользовательского интерфейса. Модуль *Blender.BGL* дает доступ ко всем гайкам и болтам, чтобы проектировать графический пользовательский интерфейс на пустом месте. Мы будем главным образом использовать первый, потому что в нём есть почти все, в чём мы нуждаемся, но мы также дадим пример последнего, чтобы сформировать простое сообщение об ошибке. Наш главный пользовательский интерфейс будет похож на это:



Когда мы вызываем наш скрипт из Меню **Add** (обычно доступно на панели меню сверху экрана или по нажатию Пробела в окне 3D-вида), появится меню как на иллюстрации, и пользователь может подбирать параметры по его или её вкусу. По нажатии кнопки **OK**, скрипт создаст насекомо-подобный меш. Из появившегося меню также можно выйти, нажав *Esc*, тогда скрипт завершится, не создавая меш.

## Создание жуков — требует некоторой сборки

Наша миссия в том, чтобы строить простые существа из небольших образцов строительных блоков, которые могут сцепляться вместе. Схема нашего скрипта такая:

1. Импортировать строительные блоки для наших существ.
2. Отобразить пользовательский интерфейс
3. Собрать меш существа из строительных блоков так, как определил пользователь.
4. Вставить меш как объект в сцену.

Мы пройдем через скрипт постепенно, показывая важные части в подробностях. (Полный скрипт доступен как [сгеепускраулис.py](http://сгеепускраулис.py).) Первый шаг включает создание частей тела, которые пригодны для сборки вместе. Это означает, что мы должны смоделировать эти части в Блендере, определяя подходящее соединение и отмечая это соединение как группу вершин. Затем мы экспортируем эти меши в виде кода на Питоне, используя скрипт, с которым мы столкнемся снова в следующей главе, поскольку она имеет дело с группами вершин.

Сейчас мы используем этот сгенерированный код на Питоне просто как модуль, содержащий несколько списков вершин, определяющих каждую часть тела. Мы должны убедиться, что этот модуль находится где-нибудь в пути поиска Питона, например, `.blender/scripts/bpymodules` будет логичным выбором, или это может быть альтернативный пользовательский каталог скриптов. Файл на Питоне с мешевыми строительными блоками называется `mymesh.py`, так что первая часть нашего кода содержит следующий оператор *import*:

```
import mymesh
```

## Создание пользовательского интерфейса

При рисовании простого интерфейса пользователя материалом будет использование *Draw.Create()* для создания необходимых кнопок, и сборка и инициализация этих кнопок с *Draw.PupBlock()*

Это несколько ограничено по сравнению со вполне оперившимися библиотеками, доступными для некоторых языков программирования, но очень легко для использования. Основная идея в том, чтобы создать интерактивные объекты, такие, как например, кнопки, затем собрать их в окне диалога, чтобы показать пользователю. В то же самое время, окно диалога задаёт некоторые ограничения на величины, которые кнопка может породить. Диалог или выпадающее меню будет показываться в позиции курсора. Блендер способен воспроизводить более сложный интерфейс пользователя, но пока мы придерживаемся основ.

Хотя *Draw.Create()* может воспроизвести кнопки-переключатели, а также кнопки ввода строк, для нашего приложения нам нужны только кнопки ввода для целых величин и величин с плавающей точкой. Тип переменной (например величина с плавающей точкой или целое), определяется типом значения по умолчанию, передаваемого в *Draw.Create()*. Кнопка **ОК** будет автоматически отображена функцией *Draw.PupBlock()*. Эта функция берет список кортежей как аргумент, где каждый кортеж определяет кнопку для отображения. Каждый кортеж состоит из текста, отображаемого на кнопке, объекта кнопки, созданного функцией *Draw.Create()*, допустимых минимума и максимума величины, и текста подсказки (tooltip), появляющегося при наведении курсора на кнопку.

```
Draw = Blender.Draw
THORAXSEGMENTS = Draw.Create(3) # Сегментов в торсе
TAILSEGMENTS = Draw.Create(5)  # Сегментов в хвосте
LEGSEGMENTS = Draw.Create(2)   # Сегментов торса с
                                # ногами
WINGSEGMENTS = Draw.Create(2)  # Сегментов торса с
                                # крыльями
EYESIZE = Draw.Create(1.0)     # Размер глаз
TAILTAPER = Draw.Create(0.9)   # Конусность каждого
                                # сегмента хвоста

if not Draw.PupBlock('Add CreepyCrawly', [
    ('Thorax segments:' , THORAXSEGMENTS, 2, 50,
     'Number of thorax segments'),
    ('Tail segments:' , TAILSEGMENTS, 0, 50, 'Number of tail
     segments'),
    ('Leg segments:' , LEGSEGMENTS, 2, 10,
     'Number of thorax segments with legs'),
```

```
('Wing segments:' , WINGSEGMENTS, 0, 10,
 'Number of thorax segments with wings'),
('Eye size:' , EYESIZE, 0.1,10, 'Size of the eyes'),
('Tail taper:' , TAILTAPER, 0.1,10,
 'Taper fraction of each tail segment'),]),
return
```

Как Вы можете видеть, мы ограничиваем возможные величины наших кнопок ввода в разумном диапазоне (вплоть до 50 для сегментов торса и хвоста), чтобы исключить нежелательные результаты (огромные величины могут обрушить вашу систему, если память или процессорная мощность скудны).

## Запоминание выбора

Было бы очень удобно, если бы мы могли запоминать выбор пользователя, чтобы можно было выставить последние настройки, когда скрипт заработает снова, но в Блендере каждый скрипт запускается изолированно, и вся информация внутри скрипта теряется, как только он завершится. Следовательно, нам нужен некоторый механизм, сохраняющий информацию в постоянном режиме. С этой целью, API Блендера имеет модуль *Registry* (Реестра), который позволяет нам сохранять величины в памяти (а также на диске), индексируемые произвольным ключом.

Наш код инициализации GUI изменится немного по своей сути, если мы хотим добавить эту функциональность, но мы покажем код, извлекающий запомненные значения (если они существуют), и сопроводим код, сохраняющий выборы пользователя:

```
reg = Blender.Registry.GetKey('CreepyCrawlies',True)
try:
    nthorax=reg['ThoraxSegments']
except:
    nthorax=3
try:
    ntail=reg['TailSegments']
except:
    ntail=5
... <подобный код для остальных параметров> ...
```

```

Draw = Blender.Draw
THORAXSEGMENTS = Draw.Create(nthorax)
TAILSEGMENTS = Draw.Create(ntail)
LEGSEGMENTS = Draw.Create(nleg)
WINGSEGMENTS = Draw.Create(nwing)
EYESIZE = Draw.Create(eye)
TAILTAPER = Draw.Create(taper)

if not Draw.PupBlock('Add CreepyCrawly', [\
... <идентичный код, как в предыдущем примере> ...
return
reg={'ThoraxSegments':THORAXSEGMENTS.val,
'TailSegments':TAILSEGMENTS.val,
'LegSegments':LEGSEGMENTS.val,
'WingSegments':WINGSEGMENTS.val,
'EyeSize':EYESIZE.val,
'TailTaper':TAILTAPER.val}
Blender.Registry.SetKey('CreepyCrawlies',reg,True)

```

Фактически чтение и запись нашего ключа в реестре выделены. Аргумент *True* (Истина) указывает, что мы хотим извлечь наши данные с диска, если они не доступны в памяти, или записать их на диск также при сохранении, чтобы наш скрипт мог иметь доступ к этой сохраненной информации, даже если мы останавливали Блендер и перезапустили его позже. Фактически получаемый или записываемый ключ реестра - это словарь, который может содержать любые данные, которые нам нужны. Конечно, к настоящему времени ключа реестра может еще не существовать, в этом случае мы получим значение *None* (Ничто) - об этой ситуации заботится оператор *try ... except ...*.

## Вся мощь графики Блендера

Всплывающий диалог достаточен для многих применений, но если он не соответствует вашим требованиям, модуль Блендера *Draw* имеет множество строительных блоков для создания интерфейса пользователя, но эти строительные блоки требуют больше усилий, чтобы склеить их вместе в рабочем приложении.

Мы используем это построение из блоков, чтобы создать всплывающее сообщение об ошибке. Это всплывающее окно просто показывает сообщение на тревожном цветном фоне, но хорошо

иллюстрирует, как действия пользователя (например, нажатия клавиш или кнопок мыши) связаны с графическими элементами.

```

from Blender import Window, Draw, BGL

def event(evt, val):
    if evt == Draw.ESCKEY:
        Draw.Exit() # exit when user presses ESC
    return

def button_event(evt):
    if evt == 1:
        Draw.Exit()
    return

def msg(text):
    w = Draw.GetStringWidth(text)+20
    wb= Draw.GetStringWidth('Ok')+8
    BGL.glClearColor(0.6, 0.6, 0.6, 1.0)
    BGL.glClear(BGL.GL_COLOR_BUFFER_BIT)
    BGL.glColor3f(0.75, 0.75, 0.75)
    BGL.glRecti(3,30,w+wb,3)
    Draw.Button("Ok",1,4,4,wb,28)
    Draw.Label(text,4+wb,4,w,28)

```

```

def error(text):
    Draw.Register(lambda:msg(text), event, button_event)

```

В функции *error()* все начинается и заканчивается для пользователя; она сообщает Блендеру что рисовать, куда посылать события, такие, как щелчки по кнопке, куда послать нажатую клавишу, и начинает взаимодействие. Лямбда-функция необходима как функция, которую мы передаем в *Draw.Register()*, которая рисует, но не принимает аргументов, в то время как мы хотим передавать разные аргументы *text* каждый раз, когда мы вызываем *error()*. Функция *lambda* по существу определяет новую функцию без аргументов, но с вложенным текстом.

Функция *msg()* отвечает за отрисовку всех элементов на экране. Она рисует цветной фон с помощью функции *BGL.glRecti()*, сообщение с текстом для отображения (с *Draw.Label()*), и кнопку ОК, которой назначается событие номер 1 (с *Draw.Button()*). Когда пользователь щелкает по кнопке ОК, этот номер события посылается

в обработчик событий (event handler) - функцию `button_event()`, которую мы передали в `Draw.Register()`. Все, что обработчик событий делает, когда он вызывается с этим номером события 1 - завершает функцию `Draw.Register()` вызовом `Draw.Exit()`, так что наша функция `error()` может завершиться.

## Создание меш-объекта

Как только мы извлекли наши списки координат вершин и индексов граней из модуля `pytmesh`, нам нужен некоторый способ для создания нового меш-объекта в нашей сцене и добавления объектов `MVert` и `MFace` в этот меш. Это можно осуществить, например, так:

```
me=Blender.Mesh.New('Bug')
me.verts.extend(verts)
me.faces.extend(faces)
scn=Blender.Scene.GetCurrent()
ob=scn.objects.new(me,'Bug')
scn.objects.active=ob

me.remDoubles(0.001)
me.recalcNormals()
```

Первая строка создает новый меш-объект с именем *Bug* (Жук). Он не будет содержать никаких вершин, рёбер или граней, не будет вставлен в объект Блендера, и не будет подключен пока ни к какой Сцене. Если имя меша уже существует, к нему будет добавлен уникальный цифровой суффикс (например, *Bug.001*).

Следующие две строки действительно создают геометрию в меше. Атрибут `verts` – это место, куда ссылается наш список объектов `MVert`. У него есть метод `extend()`, который принимает список кортежей, каждый из которых содержит координаты *x*, *y*, и *z* создаваемых вершин. Точно так же метод `extend()` атрибута `faces` принимает список кортежей, каждый из которых содержит три или больше индексов, указывающих на вершины, которые вместе определяют грань. Порядок здесь важен: нам нужно сначала добавить новые вершины; в противном случае вновь созданные грани не смогут ссылаться на них. Нет необходимости определять какие-либо рёбра, так как добавление граней также неявно создаст рёбра, которые ещё не присутствуют.

Меш по своей сути еще не является объектом, которым может манипулировать пользователь, так что в следующих нескольких строках (выделено), мы извлекаем текущую сцену и добавляем в неё новый объект. Аргументы функции `new()` - меш-объект, который мы создали ранее, и имя, которое мы хотим дать объекту. Имя, даваемое объекту, может быть таким же, как и данное мешу, так как имена мешей и имена объектов существуют в различных пространствах имён. Как и с мешем, существующее имя будет сделано уникальным посредством добавления суффикса. Если имя опущено, новый объект получит в качестве имени по-умолчанию тип своего аргумента (*Mesh* в нашем случае).

Вновь созданный объект будет выбран, но не активен, так что мы исправим это, присвоив наш объект в `scene.objects.active`.

Когда мы собираем наш меш из различных наборов вершин, результат не может быть таким же чистым, как бы нам хотелось, и, следовательно, последние два действия позволяют убедиться, что у нас нет никаких пар вершин, которые занимают почти одинаковую позицию в пространстве, и что все нормали граней единообразно указывают наружу.

## Преобразование топологии меша

Создание существа из строительных блоков требует, чтобы мы применяли дублирование, масштабирование, и отражение к этим строительным блокам прежде, чем мы склеим их вместе. В Блендере 2.49, это означает, что мы должны определить некоторые вспомогательные функции (утилиты), чтобы выполнить эти действия, так как они не присутствуют в API. Мы определяем эти вспомогательные функции в модуле **Tools** (инструменты), но мы осветим некоторые из них здесь, так как они покажут несколько интересных методов.

Некоторые действия, как например, масштабирование вокруг средней точки или перемещение вершин просты, но присоединение группы вершин к другой сложнее, так как мы хотели бы предотвратить скрещивание рёбер друг с другом и сохранить грани плоскими и недеформированными. Мы не можем просто соединить два набора вершин (или краевых цикла) вместе. Но пробуя



различные отправные точки в рёберном цикле, и проверяя, если такой выбор минимизирует расстояние между всеми парами вершин, мы обеспечиваем, чтобы не было никаких рёберных пересечений, и искажения были минимальными (хотя мы не можем полностью предотвратить искажения граней, если рёберные циклы очень разнородные по форме).

## Схема кода сшивания рёберных циклов

В функции, которая создает новые грани, мы должны выполнить следующие шаги:

1. Удостовериться, что оба рёберных цикла имеют одинаковую и ненулевую длину.
2. Для каждого ребра в цикле 1:
  1. Найти ребро в цикле 2, которое ближе всего.
  2. Создать грань, соединяющую эти два ребра.

Функция, которая осуществляет эту довольно сложную на вид схему:

```
def bridge_edgeloops(e1,e2,verts):  
  
    e1 = e1[:]  
    e2 = e2[:]  
    faces=[]  
  
    if len(e1) == len(e2) and len(e1) > 0 :
```

Функция принимает аргументы: два списка рёбер и список вершин. Рёбра представлены в виде кортежей двух целых (индексы в списке вершин *verts*), а вершины - в виде кортежей координат *x*, *y*, и *z*.

Первая вещь, которую мы сделаем - создадим копии двух рёберных списков, поскольку мы не хотим испортить списки в их оригинальном контексте. Список граней, который мы будем строить, инициализируется в пустой список, и мы проверяем разумность и равенство длин обоих рёберных списков. Если это подтверждается, мы приступаем к следующему куску:

```
for a in e1:
```

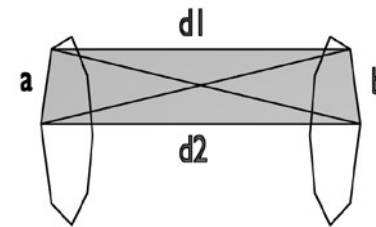
```
    distance = None # расстояние  
    best = None     # лучший  
    enot = []       # отвергнутые рёбра
```

Мы повторяем по каждому ребру в первом списке, ссылаясь на это ребро через *a*. параметр *distance* содержит расстояние до ближайшего ребра во втором рёберном списке, а *best* будет ссылкой на это ребро. *enot* - список, который копит все рёбра из второго списка, которые находятся на большем расстоянии, чем наилучшее.

В конце каждой итерации, *enot* будет содержать все рёбра из второго списка минус одно - которое мы считаем ближайшим. Затем мы переназначаем *enot* на второй список, таким образом второй список уменьшается на одно ребро с каждой итерацией. Мы заканчиваем, как только второй список рёбер будет исчерпан:

```
while len(e2):  
    b = e2.pop(0)
```

Текущее ребро из второго списка, которое мы рассматриваем, называется *b*. Для наших целей, мы определяем расстояние между *a* и *b* как сумму расстояний между соответствующими вершинами в *a* и *b*. Также мы проверяем, не окажется ли короче сумма расстояний до перевёрнутых вершин *b*. Если получилась такая ситуация, мы меняем вершины в ребре *b*. Это может казаться сложным способом действий, но суммированием двух расстояний мы гарантируем, что рёбра, которые сравнительно коллинеарны (параллельны) - привилегированы, тем самым уменьшая число неплоских граней, которые будут созданы. Проверка, не приведёт ли перевёрнутый второй край к более короткому расстоянию, мы предотвращаем образование искорёженного в виде галстука-бабочки четырёхугольника, как проиллюстрировано на следующем рисунке:



Реализация будет выглядеть похоже на предшествующий рисунок, где выделенные вектора - псевдонимы на объект *Mathutil.Vector*, преобразующий наши кортежи с координатами x, y, и z в соответствующие векторы, которые мы можем вычитать, складывать, и получать их длину.

Сначала мы вычисляем расстояние:

```
d1 = (vec(verts[a[0]]) - vec(verts[b[0]])).length + \
      (vec(verts[a[1]]) - vec(verts[b[1]])).length
```

Затем мы проверяем с перевернутым ребром b, будет ли в результате расстояние короче:

```
d2 = (vec(verts[a[0]]) - vec(verts[b[1]])).length + \
      (vec(verts[a[1]]) - vec(verts[b[0]])).length
if d2 < d1 :
    b = (b[1], b[0])
    d1 = d2
```

Если рассчитанное расстояние не самое короткое, мы откладываем ребро для следующей итерации, если оно не первое, с которым мы столкнулись:

```
if distance == None or d1 < distance :
    if best != None:
        enot.append(best)
    best = b
    distance = d1
else:
    enot.append(b)
```

Список отклонённых рёбер становится новым e2, затем мы заполняем список граней новой парой рёбер, и переходим к новой итерации по первому списку рёбер (a) – доп. пер.

```
e2 = enot
faces.append((a, best))
```

Наконец, мы преобразуем наш список граней, состоящий из кортежей двух рёбер, в список кортежей из четырех индексов:

```
return [(a[0], b[0], b[1], a[1]) for a, b in faces]
```

Есть много больше в этом скрипте, и мы вновь будем рассматривать *creepycrawlies.py* в следующей главе, где мы добавим модификаторы, группы вершин и арматуру к нашей модели. Иллюстрация показывает образцы бестиария, которые могут быть созданы скриптом.



## Ослепите вашего босса - гистограммы в стиле Блендер

Чтобы доказать, что Блендер адаптируется ко многим задачам помимо интерактивного создания 3D-графики, мы покажем Вам, как импортировать внешние данные (электронная таблица в формате CSV) и автоматизировать задачу создания и рендеринга представленной в 3D гистограммы.



Идея в том, чтобы запустить Блендер с аргументами, указывающими ему запустить скрипт, который читает .csv файл, рендерит изображение и сохраняет это изображение по окончании. Чтобы это было возможным, нам нужен способ вызывать Блендер с правильными параметрами. Мы дойдём скоро до этого скрипта, но сначала давайте увидим, как передавать параметры в Блендер, чтобы он запускал скрипт на Питоне:

```
blender -P /full/path/to/barchart.py
```

Также возможно вместо этого запустить скрипт из текстового буфера внутри *.blend* файла по имени этого текстового буфера. Обратите внимание на порядок параметров в этом случае - сначала ставится имя *.blend* файла:

```
blender barchart.blend -P barchart.py
```

В противоположность тому, что описано в документации API, в Питоне мы можем просто получить доступ к аргументам командной строки следующим образом:

```
import sys
print sys.argv
```

Последний фрагмент выведет все аргументы, включая имя программы Блендера первым. Наш скрипт должен пропускать любые аргументы, предназначенные для самого Блендера при использовании этого списка. Любые аргументы, предполагаемые только для нашего скрипта, которые не должны быть интерпретированы самим Блендером, должны находиться после **аргумента конца-опций** (end-of-options), двойного минуса (--).

Наконец, мы не хотим, чтобы Блендер появлялся и показывал графический интерфейс пользователя. Вместо этого, мы укажем ему работать в фоне и выйти по завершении. Это делается посредством прохождения опции -b. Задав всё это вместе, командная строка будет выглядеть похожей на это:

```
blender -b barchart.blend -P barchart.py -- data.csv
```

Если Блендер работает в фоновом режиме, Вы **должны** определить *.blend* файл, в противном случае Блендер разрушится. Если мы должны определить *.blend* файл, мы так же хорошо можем использовать внутренний текст для нашего скрипта на Питоне, иначе нам пришлось бы держать два файла одновременно вместо одного.

## Скрипт построения гистограммы

Здесь мы покажем важные части кода кусками (полный файл доступен как `barchart.blend`, который включает `barchart.py` как вложенный текст). Мы начинаем с создания нового объекта Мира и установки цветов его зенита и горизонта целиком в нейтральный белый (выделенная часть следующего кода):

```
if __name__ == '__main__':
    w=World.New('BarWorld')
    w.setHor([1,1,1])
    w.setZen([1,1,1])
```

Затем, мы извлекаем последний аргумент, переданный в Блендер и проверяем является ли расширение файла тем же самым .csv. Реальный промышленный код должен, конечно, иметь более серьёзную проверку на ошибки:

```
csv = sys.argv[-1]
if csv.endswith('.csv'):
```

Если у него правильное расширение, мы создаём новую Сцену с именем `BarScene` и присваиваем её атрибут `world` к нашему вновь созданному миру (Это было вдохновлено более сложным сценарием [jessethemid](http://blenderartists.org/forum/showthread.php?t=79285) на Blender Artists <http://blenderartists.org/forum/showthread.php?t=79285>). Фоновый режим не загружает никакого *.blend* файла по-умолчанию, так что сцена по-умолчанию не будет содержать никаких объектов. Тем не менее, просто, чтобы убедиться, мы создаем новую пустую сцену со значимым именем, которое будет содержать наши объекты:

```
sc=Scene.New('BarScene')
sc.world=w
sc.makeCurrent()
```

Затем, мы передаем имя файла в функцию, которая добавляет объекты *barchart* (гистограммы) на текущую сцену и возвращает центр диаграммы, чтобы наша функция *addcamera()* могла использовать его, чтобы направить туда камеру. Мы также добавляем лампу, чтобы сделать рендер возможным (в противном случае наш рендер будет весь черный).

```
center = barchart(sys.argv[-1])
addcamera(center)
addlamp()
```

Рендеринг самый простой (мы столкнемся с более сложными примерами в *Главе 8, Рендеринг и Обработка Изображения*). Мы извлекаем контекст рендеринга, который хранит всю информацию о рендеринге, например, номер кадра, какой выходной формат, размер изображения, и так далее. И, поскольку большинство атрибутов по умолчанию разумны, мы установим только выходной формат на PNG и запустим рендер.

```
context=sc.getRenderingContext()
context.setImageType(Scene.Render.PNG)
context.render()
```

Наконец, мы устанавливаем выходной каталог в пустую строку, чтобы сделать наш вывод в текущий каталог (каталог, в котором мы были, когда вызывали Блендер) и сохраняем наше визуализированное изображение. Изображение будет иметь то же базовое имя, как у .csv-файла, который мы приняли как первый аргумент, но будет иметь расширение .png. Мы проверили, что имя файла заканчивается на .csv, так что вполне безопасно тупо удалить последние четыре символа из имени файла и добавить .png

```
context.setRenderPath('')
context.saveRenderedImage(csv[:-4]+' .png')
```

Добавление лампы не значительно отличается от добавления любого другого объекта и очень подобно примеру "hello world". Мы создаём новый объект *Lamp*, добавляем его к текущей сцене и устанавливаем его позицию. Объект *Lamp* имеет, конечно, много настраиваемых параметров, но мы в этом примере довольствуемся не-направленной лампой по-умолчанию. Выделенный код показывает типичную идиому Питона: *loc* - кортеж из трех величин, но *setLocation()* принимает три отдельных аргумента, так что мы указываем, что хотим распаковать кортеж на отдельные значения с помощью \* нотации:

```
def addlamp(loc=(0.0,0.0,10.0)):
    sc = Scene.GetCurrent()
    la = Lamp.New('Lamp')
    ob = sc.objects.new(la)
    ob.setLocation(*loc)
```

Добавление камеры будет чуть-чуть сложнее, так как мы должны направить её на нашу гистограмму и убедиться, что угол обзора достаточно широкий, чтобы все видеть. Мы определяем здесь

перспективную камеру и устанавливаем довольно широкий угол. Поскольку камера по-умолчанию уже сориентирована вдоль оси z, мы не должны задавать никакого вращения, только установим позицию в 12 единиц от центра вдоль оси z, как выделено на второй строке следующего кода:

```
def addcamera(center):
    sc = Scene.GetCurrent()
    ca = Camera.New('persp', 'Camera')
    ca.angle=75.0
    ob = sc.objects.new(ca)
    ob.setLocation(center[0],center[1],center[2]+12.0)
    sc.objects.camera=ob
```

Сама функция *barchart* не такая уж большая неожиданность. Мы открываем файл с полученным именем и используем стандартный модуль *csv* из Питона, чтобы читать данные из файла. Мы загружаем все заголовки столбцов в *xlabel*, а остальные данные в *rows* (строки).

```
from csv import DictReader
```

```
def barchart(filename):
    csv = open(filename)
    data = DictReader(csv)
    xlabel = data.fieldnames[0]
    rows = [d for d in data]
```

Для того, чтобы масштабировать нашу гистограмму до разумных величин, мы должны определить пределы данных. Первый столбец каждой записи содержит значение по x (или метку), так что мы исключаем его из нашего вычисления. Так как каждая величина загружена в виде строки, мы должны преобразовать её в величину с плавающей точкой для сравнений.

```
maximum = max([float(r[n]) for n in data.fieldnames[1:]
               for r in rows])
minimum = min([float(r[n]) for n in data.fieldnames[1:]
               for r in rows])
```

Чтобы фактически создать столбики, мы проходим по всем строкам. Поскольку значение по x может быть текстовой меткой (как название месяца, например), мы сохраняем отдельно цифровое значение x для того, чтобы позиционировать столбики. Само значение x добавляется к сцене в виде объекта *Text3d* функцией *label()*, поскольку значения y визуализируются соответственно

масштабированными объектами *Cube* (Куб), добавляемыми функцией *bar()*. Функции *label()* и *bar()* не показаны здесь.

```
for x,row in enumerate(rows):
    lastx=x
    label(row[xlabel], (x,10,0))
    for y,ylabel in enumerate(data.fieldnames[1:]):
        bar(10.0*(float(row[ylabel])-minimum)/maximum,
            (x,0,y+1))
    x = lastx+1
```

Наконец, мы подписываем каждый столбец (то есть, каждый набор данных) своим собственным заголовком столбца как *label*. Мы сохранили число значений по *x*, так что мы можем вернуть центр нашей гистограммы деля его на два (*y*-компонент установлен на 5.0, так как мы масштабировали все значения по *y*, чтобы они лежали в пределах диапазона от 0 до 10).

```
for y,ylabel in enumerate(data.fieldnames[1:]):
    label(ylabel, (x,0,y+0.5), 'x')
return (lastx/2.0,5.0,0.0)
```

### Хитрость в Windows: SendTo (Отправить)

Как только у вас будет ваш *.blend* файл, содержащий корректный скрипт Питона и вы поймёте, как правильно вызвать его из командной строки, Вы можете интегрировать его более тесно с Windows XP, создав программу *SendTo*. Программа *SendTo* (в нашем случае *.BAT*-файл) - любая программа, которая принимает единственное имя файла как аргумент и что-либо делает с этим файлом. Он должен находиться в каталоге *SendTo*, который может быть расположен на разных местах в зависимости от вашей конфигурации системы. Его просто найти, щелкнув по кнопке **Пуск**, выбрав **Выполнить...**, и набрав **sendto** вместо команды. Откроется искомый каталог. В этот каталог Вы можете поместить *.BAT*-файл, в нашем случае он называется *BarChart.BAT*, и он будет содержать единственную команду:

```
/полный/путь/к/blender.exe /путь/к/barchart.blend
-P barchart.py -- %1
```

(заметьте знак процента). Теперь мы можем просто щелкать правой кнопкой мыши по любому *.csv*-файлу, с которым мы сталкиваемся, и затем выбирать *BarChart.BAT* в

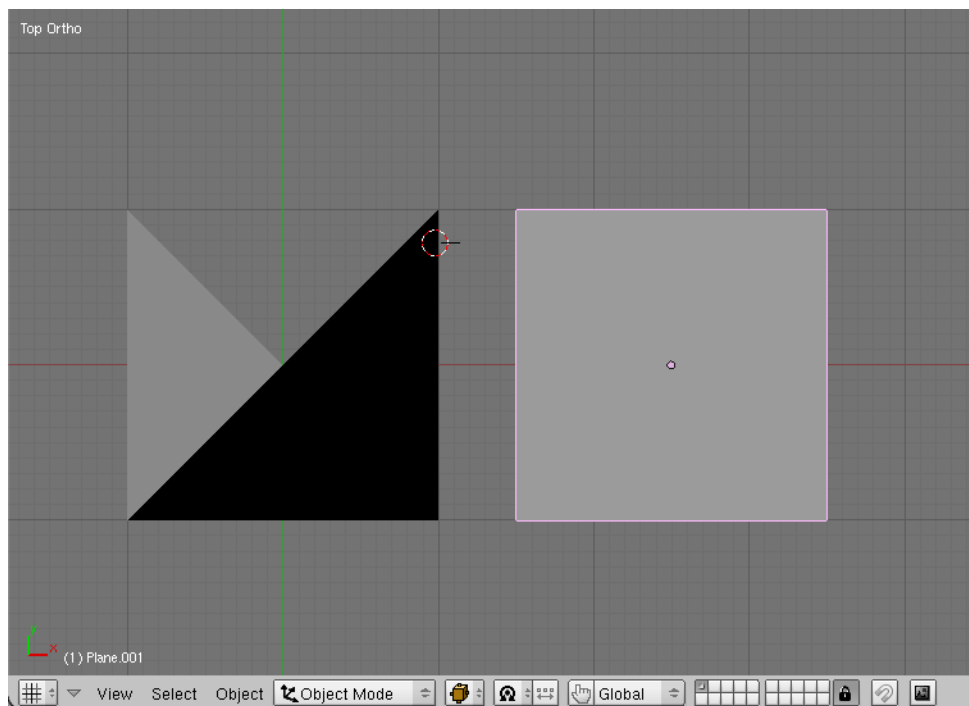
меню *Отправить*, и вуаля, *.png* файл появится рядом с нашим *.csv*.

## Таинственные грани - выбор и редактирование граней в мешах

Блендер уже предоставляет множество вариантов для выбора и манипулирования гранями, рёбрами и вершинами меша, или через встроенные методы, или через скрипты расширения Питона. Но если Вы хотите выбрать некоторые элементы, основываясь на ваших уникальных требованиях, этот раздел покажет, как это осуществить. Мы построим несколько небольших скриптов, которые иллюстрируют, как получить доступ к граням, рёбрам и вершинам, и как работать с различными свойствами этих объектов.

## Выбор искривлённых (не-планарных) четырёхугольников

**Искривлённые четырёхугольники** (Warped quads), также известные как **"а-ля галстук-бабочка"** (bow-tie quads), иногда формируются случайно при спутанном порядке вершин во время создания грани. В менее экстремальных случаях они могут быть созданы при перемещении одной вершины плоского четырёхугольника. Эта небольшая иллюстрация показывает, как они могут выглядеть в 3D-виде:



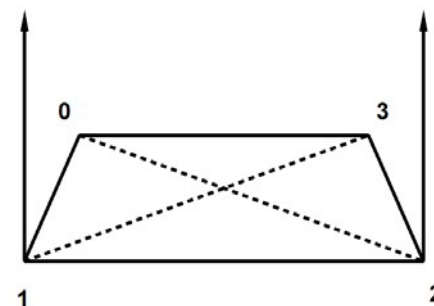
В 3D-виде, искривлённая грань справа не кажется необычной, но на рендере она не покажет однородного затенения:



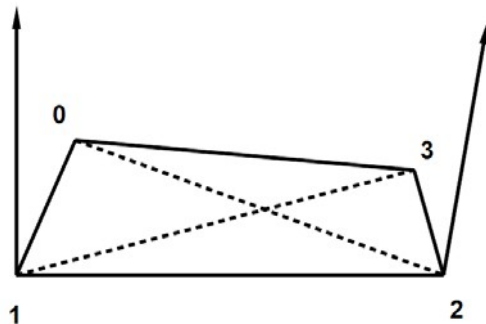
Оба объекта являются плоскостями (plane) и состоят из единственной грани с четырьмя вершинами. Тот, что слева - четырёхугольник галстук-бабочка. Его правый край перевернут на

полные 180 градусов, в результате появляется безобразный черный треугольник, где мы видим обратную сторону искривленной грани. Плоскость справа не показывает никакого заметного искажения в 3D-виде, хотя его правая верхняя вершина перемещена на значительное расстояние вдоль оси z (по линии нашего взгляда). При рендере, тем не менее, искажение правой плоскости ясно видно. Видимое искажение немного искривленного четырёхугольника можно преодолеть, включив атрибут *smooth* у грани, который интерполирует вершинные нормали вдоль грани, тогда вид результата будет плавнее. Немного искривленные четырёхугольники почти неизбежны при моделировании или деформации меша арматурой, а могут ли они привести к видимым проблемам, зависит от ситуации. Часто бывает полезно, если вы можете найти и выбрать их, чтобы вынести ваше собственное решение.

Искривлённый четырёхугольник можно идентифицировать, проверяя, что независимые нормали треугольников, которые формируют четырёхугольник, указывают в одинаковом направлении. Плоский четырёхугольник будет иметь нормали треугольников, направленные в одном и том же направлении, как показано на следующей картинке:



В то время как в искривлённом четырёхугольнике эти нормали не параллельны:



Эти нормали треугольников - не то же самое, что вершинные нормали: те определены как среднее всех нормалей граней, использующих вершину, так что мы должны вычислить самостоятельно эти нормали треугольников. Это можно сделать посредством вычисления векторного произведения рёберных векторов, то есть, векторов, определенных двумя вершинами в конце каждого ребра. В показанных у нас примерах есть левый треугольник, его нормаль формируется взятием векторного произведения рёберных векторов  $1 \rightarrow 0$  и  $1 \rightarrow 2$ , и треугольник справа, для него вычисляем векторное произведение рёберных векторов  $2 \rightarrow 1$  и  $2 \rightarrow 3$ .

Не имеет значения, просматриваем мы наши рёбра по часовой стрелке или против часовой стрелки, но мы должны быть осторожными, чтобы последовательно упорядочивать рёбра при расчете векторных произведений, поскольку знак может поменяться. Как только у нас будут наши нормали треугольников, мы можем проверить, указывают ли они в одном и том же направлении, удостоверившись, что все компоненты (x, y, и z) одного вектора масштабированы одинаково по сравнению с соответствующими компонентами второго вектора. Тем не менее, чтобы дать нам отчасти большую гибкость, мы хотели бы вычислять угол между нормальными векторами треугольников и выбирать грань, только если этот угол превышает некоторый минимум. Нам не нужно самим разрабатывать такую функцию, поскольку модуль *Blender.Mathutils* предоставляет функцию *AngleBetweenVecs()*.

Возможно построить четыре различных треугольника в четырёхугольнике, но не нужно сравнивать их все - нормалей любых

двух треугольников будет достаточно, поскольку перемещение одной вершины в четырёхугольнике изменит нормали трёх из четырех возможных треугольников.

## Схема и код выбора искривлённых граней

Вооружившись всей этой информацией, набросаем схему для нашего инструмента, она будет выглядеть так:

1. Показать всплывающий диалог для ввода минимального угла.
2. Проверить, что активный объект - это меш, и он в режиме *редактирования*.
3. Включить режим *выбора граней*
4. Для всех граней проверить, является ли она четырёхугольником, и если так:
  - Вычислить нормаль треугольника, определенного вершинами 0, 1, и 2
  - Вычислить нормаль треугольника, определенного вершинами 1, 2, и 3
  - Вычислить угол между нормальными векторами
  - Если угол > минимального угла, выбрать грань

Это транслируется в следующий код для фактического обнаружения и выбора (полный скрипт предоставлен как *warpselect.py*):

```
def warpselect(me,maxangle=5.0):
    for face in me.faces:
        if len(face.verts) == 4:
            n1 = (face.verts[0].co - \
                  face.verts[1].co ).cross(
                  face.verts[2].co - face.verts[1].co )
            n2 = ( face.verts[1].co - \
                  face.verts[2].co ).cross(
                  face.verts[3].co - face.verts[2].co )
            a = AngleBetweenVecs(n1,n2)
            if a > maxangle :
                face.sel = 1
```



Как Вы можете видеть, наша схема почти взаимно-однозначно соответствует коду. Заметьте, что *AngleBetweenVecs()* возвращает угол в градусах, так что мы можем непосредственно сравнить его с *maxangle*, который тоже выражен в градусах. Также, нет необходимости самостоятельно выполнять само векторное произведение двух векторов, так как класс *Vector* в Блендере хорошо снабжен всеми видами операторов. Прежде, чем мы сможем вызвать эту функцию, мы должны позаботиться о важной детали: для того, чтобы выбирать грани, должен быть включен режим *выбора граней*. Это можно сделать следующим образом:

```
selectmode = Blender.Mesh.Mode()  
Blender.Mesh.Mode(selectmode |  
Blender.Mesh.SelectModes.FACE)
```

Чтобы проиллюстрировать малоизвестный факт о том, что режимы выбора **не** являются взаимоисключающими, мы установили режим *выбора граней* дополнительно к любому уже выбранному режиму двоичным объединением величин или оператором (*|*). В конце скрипта мы восстанавливаем режим, который был активен.

## Выбор слишком острых граней

Существует много инструментов для выбора граней, с которыми в некоторых случаях громоздко работать. Блендер имеет встроенные инструменты, чтобы выбирать грани, которые имеют слишком маленькую площадь или которые имеют слишком короткий периметр. Тем не менее, этого недостаточно для выбора граней с рёбрами, которые формируют углы острее, чем некоторый предел. В некоторых задачах моделирования было бы очень удобно иметь возможность выбирать такие грани, так как они обычно трудны для манипуляций и могут вызывать безобразные артефакты при применении модификатора *subsurface* или при деформации меша.



Заметьте, что встроенный в Блендер инструмент *выбора острых рёбер* (sharp edges) (*Ctrl + Alt + Shift + S*) делает нечто другое, несмотря на свое название; он выбирает те рёбра, которые используются точно двумя гранями, и угол контакта между ними меньше, чем некоторая минимальная величина, или, другими словами, выбираются рёбра между гранями, которые сравнительно плоские.

Мы уже видели, что модуль Блендера *Mathutils* имеет функцию, вычисляющую угол, так что наш код является очень кратким, так как реальную работу делает единственная функция, показанная ниже. (Полный скрипт предоставлен как *sharpfaces.py*.)

```
def sharpfaces(me, minimum_angle):  
    for face in me.faces:  
        n = len(face.verts)  
        edges = [face.verts[(i+1)%n].co - face.verts[i].co  
                  for i in range(n)]  
        for i in range(n):  
            a = AngleBetweenVecs(-edges[i], edges[(i+1)%n])  
            if a < minimum_angle :  
                face.sel = 1  
                break
```

Заметьте, что мы не делаем различий между треугольными гранями и четырёхугольными, так как и те и другие могут иметь края, соединённые острым углом. Выделенная часть в предыдущем коде показывает одну тонкую деталь: всякий раз, когда мы вычисляем угол между нашими двумя рёберными векторами, мы инвертируем один из них, потому что для вычисления правильного угла оба вектора должны порождаться в одной вершине, а мы вычислили их все, последовательно указывая от одной вершины на другую.

Различие проиллюстрировано на следующем рисунке:





## Выбор вершин со множеством рёбер

В идеале меш должен содержать грани, которые состоят из только четырех вершин (эти грани обычно именуются **quads** — четырёхугольники), и у них должны быть относительно одинаковые размеры. Такая конфигурация оптимальна при деформации меша, что часто бывает необходимо в анимации. Конечно, нет ничего действительно ужасного в трехсторонних гранях (**tris**), но в общих чертах лучше избегать их, поскольку небольшие треугольные грани всё портят при применении модификатора **subsurface**, заставляя его показывать неприглядную рябь.



Но даже когда у вас есть меш, состоящий только из четырёхугольников, некоторые вершины являются центром более, чем четырех рёбер. Эти вершины иногда называют **полюсами**, отсюда и название скриптов в следующих разделах. Если количество рёбер чрезмерное, скажем шесть или больше (как показано в предыдущем скриншоте), такой участок может стать трудным для деформации, и трудным для манипуляций разработчиком модели. В большом и сложном меше эти вершины может быть сложно находить и, следовательно, нам нужно средство выбора таких вершин.

## Выбор полюсов

Для того чтобы выбрать вершины с определённого числа шагов, мы можем выполнить следующие шаги:

1. Независимо проверить, что активный объект - это *меш*.
2. Независимо убедиться, что мы - в режиме *объектов*.
3. Показать всплывающее меню для ввода минимального количества рёбер.
4. Для каждой вершины:
  - Итерация по всем рёбрам, подсчет вхождений вершины
  - Если счет - больше или равен минимуму, выбрать вершину

Этот метод - прямой и простой. Функция, которая ответственна за фактическую работу, показана ниже (полный скрипт называется

*poleselect1.py*). Она близко следует нашей схеме. Фактический выбор вершин осуществляется путем присвоения атрибуту вершины *sel*. Заметим также, что атрибуты *v1* и *v2* объекта ребра не являются индексами в атрибуте *verts* нашего меша, а ссылаются на объекты *MVert*. Вот почему нам нужно извлекать атрибуты *index* для сравнения.

```
def poleselect1(me,n=5):
    for v in me.verts:
        n_edges=0
        for e in me.edges:
            if e.v1.index == v.index or
               e.v2.index == v.index:
                n_edges+=1
            if n_edges >= n:
                v.sel = 1
                break
```

## Выбор полюсов, ещё раз

Вы вероятно обратили внимание, что мы повторяли обход списка рёбер по новой для каждой вершины (выделено в предыдущем коде). Это может быть дорого с точки зрения производительности и эта стоимость даже усложнена необходимостью сравнивать индексы, которые нужно извлекать снова и снова. Возможно ли написать более эффективный код, который, тем не менее, останется удобочитаемым? Да, если мы будем следовать этой стратегии:

1. Независимо проверить, что активный объект - это меш.
2. Независимо убедиться, что мы - в режиме *объектов*.
3. Показать всплывающее меню для ввода минимального количества рёбер.
4. Инициализировать словарь, проиндексированный индексом вершин, который будет содержать счетчики рёбер.
5. Повторять цикл по всем рёбрам (обновлять счет для обеих вершин, на которые ссылается ребро).
6. Повторять цикл по всем элементам словаря (если счет - больше или равен минимуму, выбираем вершину).

Используя эту стратегию, мы просто выполняем две, возможно длительных, итерации ценой памяти, требуемой на хранение словаря (ничто не бесплатно). Увеличение скорости незначительно для небольших мешей, но может быть серьёзным (Я отмечал 1,000-кратное повышение скорости у небольшого меша в 3,000 вершин) для больших мешей, и они - именно тот тип мешей, где кому-нибудь может понадобиться средство, подобное этому.

Наша переделанная функция выбора показана ниже (полный скрипт называется *poleselect.py*). Сначала отметьте оператор *import*. Словарь, который мы будем использовать, называется **словарь со значением по-умолчанию** (default dictionary) и предоставляется модулем Питона *collections*. словарь по-умолчанию является словарем, который инициализирует отсутствующие элементы при первой ссылке на них. Так как мы хотим увеличивать на 1 значение счетчика каждой вершины, на которую ссылается ребро, мы должны были бы или инициализировать наш словарь нулевыми величинами для каждой вершины в меше заблаговременно, или проверять каждый раз, проиндексирована ли уже вершина, для которой мы хотим увеличить счет, и если нет, инициализировать его. Словарь по умолчанию позволяет забыть о необходимости инициализировать все заранее, и является очень удобочитаемой идиомой.

Мы создаем наш словарь, вызывая функцию *defaultdictionary()* (функция, возвращающая новый объект, поведение которого настраивается некоторым аргументом, передаваемым в функцию, называется фабрикой в объектно-ориентированных кругах) с аргументом *int*. Аргумент должен быть функцией, не принимающей никаких аргументов. Встроенная функция *int()*, которую мы здесь используем, возвращает целую величину, равную нулю, когда вызывается без аргументов. Каждый раз, когда мы обращаемся к нашему словарю по несуществующему ключу, создаётся новый элемент, и его значением будет результат, возвращённый нашей функцией *int()*, то есть нуль. Существенные строки - те две, где мы увеличиваем счетчик рёбер (выделенная часть следующего кода). Мы могли бы написать это выражение немного другим способом, для иллюстрации, почему нам нужен словарь со значением по-умолчанию:

```
edgecount[edge.v1.index] = edgecount[edge.v1.index] + 1
```

Элемент словаря, на который мы ссылаемся с правой стороны выражения, не будет еще существовать всякий раз, когда мы ссылаемся на индекс вершины, с которой мы сталкиваемся впервые. Конечно, мы могли бы проверить это заранее, но это сделало бы код в целом гораздо менее читабельным.

```
from collections import defaultdict
```

```
def poleselect(me,n=5):
    n_edges = defaultdict(int)
    for e in me.edges:
        n_edges[e.v1.index]+=1
        n_edges[e.v2.index]+=1
    for v in (v for v,c in n_edges.items() if c>=n ):
        me.verts[v].sel=1
```

## Определение объема меша

Хотя Блендер не является на самом деле программой САПР (CAD), множество людей используют его для САПР-подобных задач, как например, архитектурная визуализация. Блендер способен импортировать множество типов файлов, включая файлы основных САПР-программ, так что включение технических моделей, сделанных с точными размерами, никогда не было проблемой.

Эти САПР-программы часто предлагают все типы инструментальных средств для измерения размеров вашей модели (её частей), тогда как Блендер, по своей природе, обеспечивает лишь очень малую часть этих инструментов. Возможно узнать размер и позицию объекта, нажав клавишу *N* в окне 3D-вида. В режиме *редактирования* Вы можете включить отображение длин рёбер, углов между рёбрами, и площадей граней (смотри панель **Mesh tools more** в контексте редактирования (*F9*) окна Кнопок), но это почти всё, что можно выяснить.

Питон может преодолеть эти ограничения в ситуациях, когда нам нужны какие-либо специфические измерения, но мы не можем экспортировать нашу модель в САПР-программу. Практическим примером является вычисление объема меша. В настоящее время, множество компаний предлагают возможность восстановить вашу цифровую модель в виде объекта реального мира посредством методов 3D-печати. Я должен сказать, что это - особенное чувство,

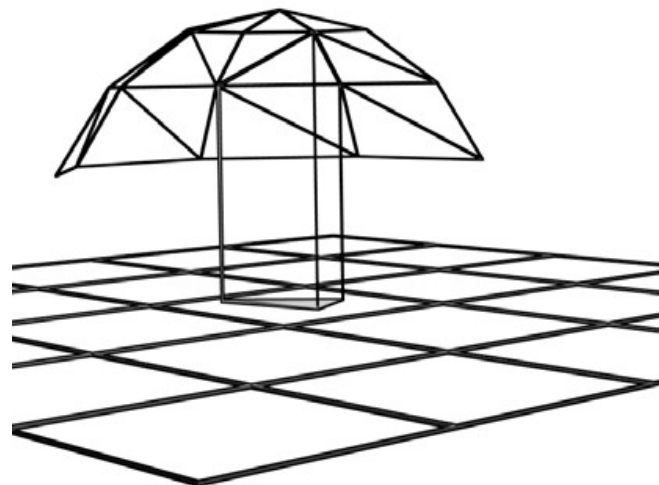
когда держишь пластиковую или даже металлическую копию вашей Блендер-модели в своих руках, это действительно добавляет целое новое измерение к 3D.

Сейчас основным компонентом цены 3D-печати модели является суммарный объем материала, который будет использован. Часто будет возможно разработать вашу модель как полый объект, который тратит меньше материала при производстве, но это очень неудобно - отправлять промежуточные версии вашей модели снова и снова, чтобы программное обеспечение изготовителя вычисляло объем и давало Вам ценовое предложение. Так что мы хотим иметь скрипт, который может вычислить объем меша достаточно точно.

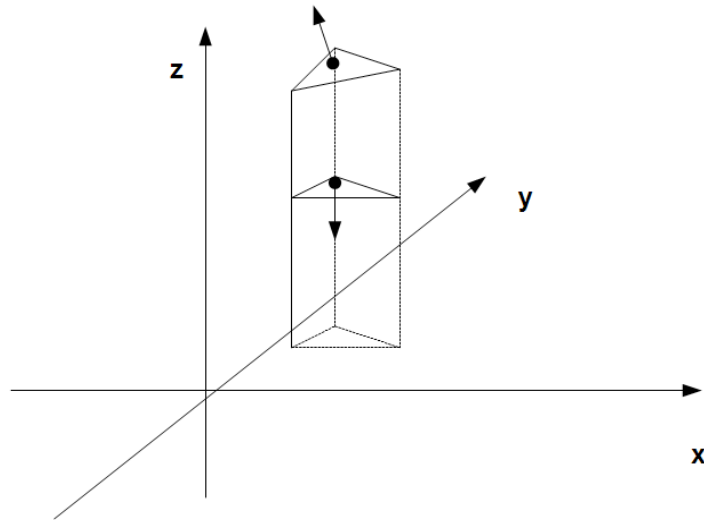
Общий метод вычисления объема меша иногда именуется **Формула Surveyor's** (землемера), так как он связан со способом землемеров для вычисления объема холма или горы триангуляцией их поверхности.

Основная мысль в том, чтобы разбить триангулированный меш на множество колонн, которые имеют основание на плоскости *xy*.

Площадь поверхности треугольника проецируется на плоскость *xy*, умножается на среднюю координату *z* трех вершин - это даёт объем такой колонны. Суммирование по всем этим объемам даст в результате объем полного меша (смотри следующий рисунок).



Есть пара вещей, которые должны быть приняты во внимание. Во-первых, меш может оказаться расширенным вниз от плоскости  $xy$ . Если мы создаем колонну от грани, которая лежит ниже плоскости  $xy$ , произведение спроецированной площади и средней координаты  $z$  будет отрицательным числом, так что мы должны вычесть эту величину, чтобы получить объем.



Во-вторых, меш может лежать полностью или частично выше плоскости  $xy$ . Если мы взглянем на пример на рисунке сверху, мы увидим что объект имеет два треугольника, которые дают вклад в объем объекта, верхний и нижний (вертикальные треугольники имеют нулевую спроецированную площадь, так что они не дают никакого вклада). Так как обе верхняя и нижняя грани лежат выше плоскости  $xy$ , мы должны вычесть объем колонны, создаваемой от нижней грани из объема, созданного верхней гранью. Если объект будет полностью ниже плоскости  $xy$ , он будет с другой стороны, и мы должны будем вычесть объем верхней колонны из объема нижней колонны.

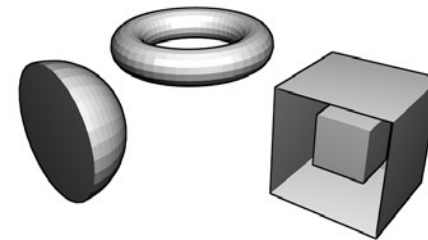
Мы можем сказать, что действие, которое нужно выполнить, определяется направлением нормалей наших треугольников. Если, например, треугольник - выше плоскости  $xy$ , но его нормаль

указывает вниз (она имеет отрицательный  $z$ -компонент), тогда мы должны вычесть рассчитанный объем. Следовательно важно, чтобы все нормали единообразно указывали наружу (в режиме редактирования выберите все грани и нажмите *Ctrl + N*).

Если мы принимаем во внимание все четыре возможности (нормаль грани направлена вверх или вниз, грань выше или ниже плоскости  $xy$ ), мы можем написать следующую схему для нашей функции:

1. Убедиться, что у всех граней нормали единообразно указывают наружу.
2. Для всех граней
  - Вычислить  $z$ -компоненту вектора нормали грани **Nz**
  - Вычислить произведение **P** среднего числа  $z$ -координат и площади спроецированной поверхности.
  - Если Nz положительно: прибавить P
  - Если Nz отрицательно: вычесть P

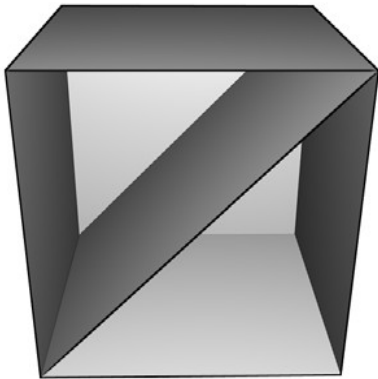
Этот отличный алгоритм работает для простых объектов без отверстий так же, как и для объектов, содержащих отверстия (например, тор), или даже полых (то есть, содержащих объект, полностью заключенный в другом объекте), примеры приведены на следующем скриншоте:



Поскольку мы допускаем, что произведение площади и координаты  $z$  может быть отрицательным, мы должны проверять только на направления нормали грани, чтобы охватить все ситуации.



Заметьте, что для меша необходимо быть закрытым и быть многогранником (manifold): Там не должно быть никаких отсутствующих граней, а также не должно быть никаких рёбер, которые не разделяют ровно двух граней, таких, как внутренние грани.



Важная часть кода показана здесь (полный скрипт называется *volume.py*):

```
def meshvolume(me):  
    volume = 0.0  
    for f in me.faces:  
        xy_area = Mathutils.TriangleArea(vec(f.v[0].co[:2]),  
                                           vec(f.v[1].co[:2]), vec(f.v[2].co[:2]))  
        Nz = f.no[2]  
        avg_z = sum([f.v[i].co[2] for i in range(3)])/3.0  
        partial_volume = avg_z * xy_area  
        if Nz < 0: volume -= partial_volume  
        if Nz > 0: volume += partial_volume  
    return volume
```

Выделенный код показывает, как мы вычисляем площадь треугольника, спроектированного на плоскость  $xy$ . *TriangleArea()* вычислит область двумерного треугольника, если ему передать двухмерные точки (точки на плоскости  $xy$ ). Итак, мы не передаем полные

координатные векторы вершин, но усекаем их (то есть, мы отбрасываем координату  $z$ ) до двух-компонентных векторов.

После прогона скрипта из текстового редактора или из меню **Scripts** в режиме объектов, появляется сообщение, показывающее объем в единицах Блендера. Прежде, чем выполнять скрипт, убедитесь, что все модификаторы применены, масштабирование и вращение применено (*Ctrl + A* в режиме *объектов*), меш полностью триангулирован (*Ctrl + T* в режиме *редактирования*), и, что меш является закрытым многогранником (manifold), проверив на non-manifold рёбра (*Ctrl + Alt + Shift + M* в режиме *выбора рёбер*). **Рёбра manifold** являются рёбрами, которые используются в точности двумя гранями. Также убедитесь, что все нормали указывают в правильном направлении. Применение модификаторов необходимо сделать, чтобы меш стал закрытым (если это - модификатор зеркальности mirror) и, чтобы сделать вычисление объема точным (если это - модификатор subsurface).

## Определение центра масс меша

При печати трехмерного объекта в пластмассе или металле, возможно, всплывёт невинный на вид вопрос, как только мы создадим нашу первую игрушку, основанную на созданном нами меше; где его центр масс? Если наша модель имеет ноги, и мы не хотим, чтобы она неожиданно упала, лучше бы центр масс находился где-нибудь над её ногами, и, по возможности внизу, чтобы она держалась стабильно. Схематически это показано на картинке:



Как только мы узнали, как определять объем меша, мы можем взять оттуда и по-новой использовать многие концепции для разработки скрипта, определяющего центр масс. Нам необходимо знать два дополнительных момента для вычисления позиции центра масс:

- Центры массы спроецированных объемов мы построим при расчете объема меша
- Как складывать рассчитанные центры масс всех этих индивидуальных объемов

Все это допускает, что твердые части нашего меша имеют однородную плотность. Меш может иметь любую форму или даже быть полым, но для твердых частей принимается, что их плотность однородна. Это разумное предположение для материалов, осаждаемых 3D-принтерами.

Первый вопрос заключает в себе немного геометрии: спроецированный объем - по существу треугольная колонна (или треугольная призма), закрытая, возможно, наклонной треугольной гранью. Расчет центра масс можно сделать следующим образом: координаты  $x$  и  $y$  центра масс являются координатами  $x$  и  $y$  центра спроецированного треугольника на плоскость  $xy$  - это просто среднее арифметическое координат  $x$  и  $y$  соответственно трех точек задающих треугольную грань. Координата  $z$  центра масс - это половина средней высоты нашей спроецированной колонны. Это - среднее арифметическое  $z$ -координат трех точек треугольной грани, поделенное на два.

*К сожалению, такой простой расчет средних значений координат не даст точного положения центра масс, в отличие от расчета объема в предыдущем разделе. Этот вопрос заключает в себе не «немного геометрии», а много матана. Точный расчет включает в себя, как минимум, вычисление двойного интеграла по площади спроецированного треугольника (для расчета ЦМ произвольного трёхмерного тела необходим тройной интеграл). Формулы расчета координат ЦМ есть, например, в этой книге: <http://www.toroid.ru/zaporojecGI.html>, стр. 281, пример расчета для похожей призмы (правда, с квадратным основанием) на стр. 285. Совпадение результатов применяемого автором метода с*

*точным возможно только в случае горизонтальности треугольной грани, в остальных случаях будет присутствовать погрешность. Конечно, если площадь треугольника мала, а высота столба много больше любой из сторон этого треугольника (обычная ситуация в высокополигональном меше), то погрешность будет небольшой, однако она всё равно во много раз больше, чем погрешности, обсуждаемые в следующем разделе. — наглая отсебятина переводчика.*

Второй вопрос заключен главным образом в здравом смысле: даны две массы  $m_1$  и  $m_2$  с их центрами масс в  $v_1$  и  $v_2$  соответственно, их комбинированный центр масс является средне взвешенным. То есть, центр масс - пропорционально ближе к центру масс тяжелого компонента.



Конечно, это у нас теперь здравый смысл, но кто-то вроде Архимеда должен был увидеть, где здесь действительно здравый смысл. После нахождения этого 'закона рычагов' (как он назвал это), он не кричал "эврика!" и не бегал голым, так что потребовалось отчасти больше времени для привлечения внимания.

Давайте поместим всю эту информацию в рецепт, которому мы можем последовать:

1. Убедиться, что у всех граней нормали единообразно указывают наружу.
2. Для всех граней:
  - Вычислить  $z$ -компоненту вектора нормали грани  $N_z$
  - Вычислить произведение  $P$  среднего числа  $z$ -координат и площади спроецированной поверхности.
  - Вычислить  $ЦМ(x, y, z)$  с  $x, y$ , как среднее от спроецированных координат  $x, y$ , и  $z$  как (среднее число  $z$ -координат грани)/2
  - Если  $N_z$  положителен: прибавить  $P$ , умноженное на  $ЦМ$
  - Если  $N_z$  отрицателен: отнять  $P$ , умноженное на  $ЦМ$

Из схемы выше ясно, что расчет центра масс идет рука об руку с вычислением частичных объемов, так что имеет смысл переопределить функцию `meshvolume()` в следующей:

```
def meshvolume(me):
    volume = 0.0
    cm = vec((0,0,0))
    for f in me.faces:
        xy_area = Mathutils.TriangleArea(vec(f.v[0].co[:2]),
                                           vec(f.v[1].co[:2]),vec(f.v[2].co[:2]))
        Nz = f.no[2]
        avg_z = sum([f.v[i].co[2] for i in range(3)])/3.0
        partial_volume = avg_z * xy_area
        if Nz < 0: volume -= partial_volume
        if Nz > 0: volume += partial_volume
        avg_x = sum([f.v[i].co[0] for i in range(3)])/3.0
        avg_y = sum([f.v[i].co[1] for i in range(3)])/3.0
        centroid = vec((avg_x,avg_y,avg_z/2))
        if Nz < 0: cm -= partial_volume * centroid
        if Nz > 0: cm += partial_volume * centroid
    return volume,cm/volume
```

Добавленные или изменённые строки выделены

## Некоторые замечания о точности

Хотя большинство из нас - художники, а не инженеры, мы все еще можем спросить, насколько точным является число, которое мы вычисляем для нашего объема или центра масс меша. Есть два вопроса для рассмотрения - существенная точность и вычислительная точность нашего алгоритма.

**Существенная точность (Intrinsic accuracy)** - это когда мы ссылаемся на рассмотрение того факта, что наша модель сделана из небольших многоугольников, которые аппроксимируют некоторую представляемую форму. При выполнении моделирования органических объектов это не имеет особого значения, если наша модель выглядит хорошо, она хорошая. Тем не менее, если мы пытаемся аппроксимировать некоторую идеальную форму, например сферу, полигональной моделью (скажем uv-сферой, или iso-сферой) найдется различие между рассчитанным объемом и известным объемом идеальной сферы. Мы можем улучшить эту

аппроксимацию, увеличивая количество разбиений (или, что тоже самое, уменьшая размер полигонов), но мы никогда не сможем полностью устранить это различие, и используемый алгоритм для вычисления объема не сможет изменить это.

**Вычислительная точность (Computational accuracy)** имеет несколько аспектов. Во-первых, есть точность чисел, при расчетах с ними. На большинстве платформ, на которых Блендер работает, вычисления выполняются с использованием чисел с плавающей точкой двойной точности. Это соответствует приблизительно 17 цифрам точности и мы ничего не можем сделать, чтобы улучшить это. К счастью, это более чем достаточная точность для работы.

Затем есть точность нашего алгоритма. Если вы посмотрите на код, вы увидите, что мы складываем и умножаем потенциально огромное количество величин, а типичная модель с высоким разрешением может содержать более 100 тысяч граней или даже миллион. Для каждой грани мы вычисляем объем спроецированного столба, и все эти объемы складываются (или вычитаются) вместе. Проблема в том, что эти объемы могут значительно отличаться по величине, не только потому, что площади граней могут отличаться, но особенно потому, что площади проекций вблизи вертикальной плоскости очень малы по сравнению теми, что близки к горизонтальной плоскости.

Теперь, если мы складываем очень большое и очень маленькое число с ограниченной точностью вычислений, мы потеряем маленькое число. Например, если наша точность должна быть ограничена тремя значимыми цифрами, при сложении 0.001 и 0.0001 мы должны получить 0.001, теряя эффект от маленького числа. Реально наша точность намного лучше (около 17 цифр), но мы и складываем намного больше, чем два числа. Однако, если мы осуществляем функцию `volume()`, используя один из приведенных алгоритмов, разница никогда не вырастет более чем до 1 миллиона, так что пока мы не начнём заниматься ядерной физикой в Блендере, нет необходимости беспокоиться. (Для тех кто всё-таки беспокоится, альтернатива приведена в скрипте как функция `volume2()`. Тем не менее, проследите за тем, чтобы Вы знаете, что Вы делаете).

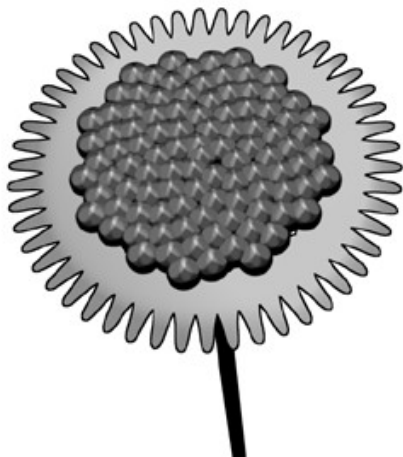




Питон способен работать с числами потенциально бесконечного размера и точности, но это значительно медленнее, чем выполнение нормальных вычислений с плавающей точкой. Функции и классы, предоставляемые в Mathutils, первоначально закодированы в C для скорости и ограничены числами с плавающей точкой двойной точности. Смотри <http://code.activestate.com/recipes/393090/>, <http://code.activestate.com/recipes/298339/> или Раздел 18.5 Поваренной книги Питона, 2-е издания, O'Reilly для некоторых других методов и математической подготовки.

## Растущий подсолнечник - присвоение родителей и группирование объектов

Создание сложнособранных объектов автоматизируется достаточно легко, но мы хотели бы обеспечить конечного пользователя способами выбирать все эти связанные объекты, чтобы затем перемещать их вместе. Этот раздел показывает, как мы можем этого достичь, создавая группы и назначая родителей объектам. В результате в конце у Вас будет связка милых подсолнухов.



## Группы

Группы разработаны, чтобы облегчить выбор или манипуляцию более, чем одним объектом одновременно. Иногда такое поведение является частью большей схемы. Арматура, например - набор костей, но зато такие наборы имеют очень специфические отношения (у костей в арматуре точно определены отношения между собой).

Есть много ситуаций, где мы хотели бы идентифицировать связку объектов, как причастных друг к другу без специфических отношений. Блендер предоставляет два типа групп, чтобы помочь нам определить их свободные отношения: **группы объектов** (или просто группы) для именованных наборов объектов, и **группы вершин** для именованных наборов вершин внутри меш-объектов.

Группы объектов позволяют нам выбирать иначе не связанный набор объектов, которые мы добавили к группе (мы могли бы сгруппировать меш, арматуру, и несколько пустышек вместе, например). Групповое отношение отличается от отношений родитель-ребенок. Группы просто позволяют нам выбирать объекты, а объекты, имеющие родителей, перемещаются вслед за их родителем, если его перемещают. Функциональность определения и манипулирования группами предоставлена в модуле Group и его идентично названным классом (группа - это просто тоже тип объекта в Блендере, но он содержит список ссылок на другие объекты, но не на другие группы, к несчастью). Вы можете, например, добавить группу из внешнего .blend файла точно так же, как *Лампу* или *Меш*. Следующая таблица включает некоторые часто используемые операции с группами (смотри модуль *Blender.Group* в документации API Блендера для дополнительной функциональности):

Операция	Действие
<code>group=Group.New(name='aGroupName')</code>	Создаёт новую группу
<code>group=Group.Get(name='aGroupName')</code>	Получить ссылку на группу по имени

Группы вершин являются удобным путем идентифицировать группы связанных вершин (как, например, ухо или нога в модели игрушки), но они имеют свою область применения за рамками



простого выбора. Их можно использовать для определения влияния деформаций костями или для идентификации по имени регионов эмиттеров для каждой из нескольких систем частиц. На группах вершин мы сфокусируемся в следующей главе.

## Отношения родитель-потомок

Воспитание детей в реальной жизни может быть труднее в разы, но в Блендере это довольно легко, хотя он иногда удивляет разнообразием вариантов на выбор. Родителем объекта можно назначить другой объект, единственную кость в арматуре, или, одну или три вершины в меш-объекте. Следующая таблица показывает важные методы (Посмотрите *Blender.Object* в API Блендера для дополнительной функциональности):

Оператор	Действие
<code>parent.makeParent([child1, child2, child3])</code>	Присвоение родителя-объекта объектам child
<code>parentmesh.makeParentVertex([child1, child2,child3], vertexindex1)</code>	Присвоение родителя-вершины объектам child
<code>parentmesh.makeParentVertex([child1, child2,child3],vertexindex1,vertexindex2,vertexindex3)</code>	Присвоение родителей - 3 вершин объектам child

## Выращивание подсолнуха из семечка

Мы можем поместить всю эту информацию для эффективного использования, когда мы пишем скрипт, который будет создавать модель подсолнуха (Ван Гог, вероятно, снова отрезал бы себе другое ухо, если бы он увидел этот "подсолнух", но с другой стороны, это был другой способ смотреть в целом). Единственный подсолнух, который мы создадим, состоит из стебля и головки цветка. Головка

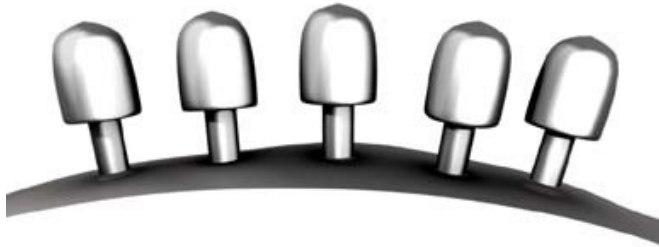
подсолнуха состоит из небольших цветков, которые станут семенами после оплодотворения, и обода с большими лепестками. (Я знаю, любой ботаник съезжится от моего языка. Маленькие цветки называются "disc florets" но floret (цветочек) - просто "маленький цветок", не так ли? А те на краю - "ray florets".) Наша головка будет иметь семена и каждое семечко является отдельным меш-объектом, который будет потомком вершины нашего главного меша.

Мы хотим, чтобы наши семена не просто двигались вместе с нашей семенной головкой, а следовали за любым локальным изгибом и самостоятельно ориентировались перпендикулярно поверхности головки, так чтобы мы могли, например, искривить главный меш с помощью пропорционального редактирования, и все подключенные к нему семена следовали за ним. Для достижения этого мы используем трёх-вершинный вариант родителя.

При присвоении объекту родителя в виде трех различных вершин меша, этот объект последует за позицией этих вершин и ориентирует себя относительно нормали (смотри следующие иллюстрации):



Нам не нужно соединять все эти триплеты вершин, так как главный меш сам не рендерится (он будет полностью закрыт семенами). Все-же мы определим грань в каждом триплете вершин; в противном случае для разработчика модели будет трудно увидеть главный меш в режиме редактирования.



Лепестки являются отдельными объектами, главный меш будет их родителем как объект, так как они не должны следовать за кривизной меша головки, только за позицией и вращением. Головка, в свою очередь, будет потомком стебля, так что мы можем перемещать всю сборку, перемещая стебель.

Наконец, мы назначаем все индивидуальные объекты в единую группу. Таким образом, можно будет легко выбрать всё за один раз, и это позволит нам ссылаться или добавлять один или больше подсолнухов из внешнего файла как единую сущность.

## Дублирование против связанной копии

Мы сказали, что все наши семена и лепестки - отдельные объекты, но имеет больше смысла сделать взамен него экземпляр (в Блендере называется **создать связанную копию**). Так как все семена и все лепестки, которые мы смоделировали, идентичны, мы можем ссылаться на те же самые меш-данные и просто изменять позицию, вращение, или масштаб объекта как нужно, сохраняя приличное количество памяти. При использовании Блендера интерактивно, мы можем создать экземпляр объекта, нажимая *Alt + D* (вместо *Shift + D* для обычной копии). В нашем скрипте, мы просто определяем новый объект и указываем его на тот же меш-объект, передавая ссылку на тот же меш при вызове *Object.New()*.

## Выращивание подсолнуха

Давайте посмотрим на основную часть скрипта, который создаёт подсолнух (полный скрипт доступен как *sunflower.py*). Первый шаг должен вычислить позицию семян:

```
def sunflower(scene, nseeds=100, npetals=50):
    pos = kernelpositions(nseeds)
```

Исходя из этих позиций мы создаем головку, у которой вершины и грани мы можем сделать родителем зёрен и собрать их в меш головки (выделенная часть следующего кода):

```
headverts=pos2verts(pos)
faces=[(v,v+1,v+2) for v in range(0,len(headverts),3)]
head=Tools.addmeshobject(scene,headverts,
                           faces,name='head')
```

Следующий шаг должен создать базовый меш для зерна и создать объекты, которые ссылаются на этот меш (выделенная часть следующего кода):

```
kernelverts,kernelfaces=kernel(radius=1.5,
                                scale=(1.0,1.0,0.3))
kernelmesh = Tools.newmesh(kernelverts,
                             kernelfaces,name='kernel')
kernels = [Tools.addmeshduplicate(scene,kernelmesh,
                                   name='kernel')

            for i in range(nseeds)]
```

Каждому зерну затем назначается пригодная позиция и родитель - подходящие вершины в меше головки цветка (выделенная часть следующего кода):

```
for i in range(nseeds):
    loc = Tools.center(head.data.verts[i*3:(i+1)*3])
    kernels[i].setLocation(loc)
    head.makeParentVertex([kernels[i]],
                           tuple([v.index for v in
                                   head.data.verts[i*3:(i+1)*3]]))
```

Затем мы создаем меш лепестка и размещаем дубликаты этого меша вдоль обода головки цветка (выделенная часть следующего кода):

```

    petalverts,petalfaces=petal((2.0,1.0,1.0))
    petalmesh =
Tools.newmesh(petalverts,petalfaces,name='petal')
    r = sqrt(nseeds)
    petals =
[Tools.addmeshduplicate(scene,petalmesh,name='petal')
    for i in range(npetals)]

```

Каждый лепесток позиционируется, поворачивается вдоль обода и назначается потомком головки (выделенная часть следующего кода):

```

for i,p in enumerate(petals):
    a=float(i)*2*pi/npetals
    p.setLocation(r*cos(a),r*sin(a),0)
    e=p.getEuler('localspace')
    e.z=a
    p.setEuler(e)
    head.makeParent(petals)

```

Наконец, мы создаём меш и объект стебля, и назначаем стебель родителем головки. Таким образом, весь цветок может перемещаться при перемещении стебля:

```

# добавление стебля (stalk) (head назначается потомком
stalk)
stalkverts,stalkfaces=stalk()
stalkob =
Tools.addmeshobject(scene,stalkverts,stalkfaces,
                    name='stalk')
stalkob.makeParent([head])

```

Все, что осталось сделать - нужно сгруппировать зерна и лепестки в отдельных группах (выделено), и затем все части подсолнуха в целом группируются, чтобы было легко ссылаться на него:

```

kernelgroup = Blender.Group.New('kernels')
kernelgroup.objects=kernels
petalgroup = Blender.Group.New('petals')
petalgroup.objects=petals
all = Blender.Group.New('sunflower')
all.objects=sum([kernels,petals],[head,stalkob])

```

Функция *addmeshduplicate()*, используемая в коде, объявлена в модуле *Tools* следующим способом:

```

def addmeshduplicate(scn,me,name=None):
    ob=scn.objects.new(me)
    if name : ob.setName(name)
    scn.objects.active=ob
    me.remDoubles(0.001)
    me.recalcNormals()
    for f in me.faces: f.smooth = 1
    me.update()
    Blender.Window.RedrawAll()
    return ob

```

Принимая сцену, меш, и имя (необязательное) для объекта, она добавляет новый объект в сцену. Меш-объект принимается как аргумент, и может использоваться снова и снова для создания новых объектов, которые ссылаются на этот же меш.

Вновь созданные объекты становятся автоматически выбранными, но не делаются активными, так что следующий шаг должен сделать вновь-созданный объект активным (выделено в предыдущем коде). Не необходимы, но, возможно, удобны пользователю следующие два действия: обеспечение того, чтобы все нормали граней были единообразно направлены наружу, и удаление всех вершин, которые слились слишком близко вместе. Эти последние два действия можно выполнить только в меше, который вставлен в объект.

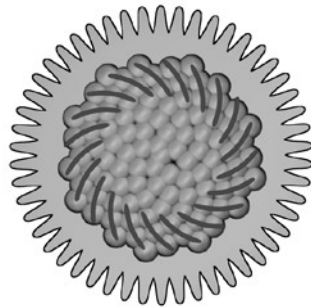
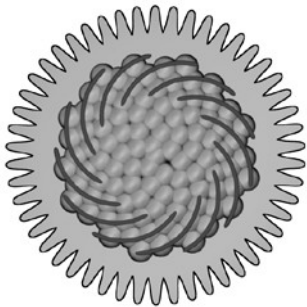
Также, для удобства, мы установили атрибут *smooth* (плавно) для всех граней, чтобы получить более гладкие изображения при рендере. Наконец, мы обновляем (update) список отображения для этого меша и уведомляем всё окно Блендера, что имеется изменение.

### Небольшое отступление, или почему кролики связаны с подсолнухами.



Одна из вещей, которую Вы можете заметить - то, что мы разместили семена в специфической спирали. Этот тип спирали, где последующие позиции вдоль спирали расположены идущими с так называемым *Золотым сечением*, называется **спираль Ферма** (Fermat's spiral). Такой спиралью получается естественным образом во многих семенных головках, когда цветочки или семена формируются в середине и выталкиваются наружу, в результате получается очень рациональная (плотная) упаковка.

Когда мы увидели, размещение семян также кажется, следует за обоими левым и правым поворотами кривых. Количество этих кривых обычно является парой из *последовательности Фибоначчи* [ 1 1 2 3 5 8 13 21 ] и отношение такой пары чисел стремится сойтись в Золотом сечении, когда они становятся больше. (В двух иллюстрациях нашей семенной головы внизу мы можем различить 13 спиралей против часовой стрелки и 21 спираль по часовой стрелке.) Фибоначчи изобрёл эту серию в попытке моделирования роста населения кроликов. Больше о подсолнухах (и, возможно, кроликах), можно обнаружить [здесь: http://en.wikipedia.org/wiki/Sunflower](http://en.wikipedia.org/wiki/Sunflower).



## Итог

В этой главе мы увидели, как создавать сложные объекты, и как сделать задачу конфигурирования этих объектов легкой для конечного пользователя, предоставив графический интерфейс, который помнит предыдущие настройки. Мы также увидели, что можно привлечь Блендер, как инструмент командной строки, для автоматизации часто выполняемых задач.

Мы также узнали, как создавать отношение родитель-потомок между объектами и сделали первый шаг в редактировании мешей. В подробностях, мы увидели как:

- Создать конфигурируемый меш-объект
- Разработать графический интерфейс пользователя
- Заставить ваш скрипт сохранять выборы пользователя для многократного использования впоследствии
- Выбирать вершины и грани в меше
- Делать родителем объекта другой объект
- Создавать группы
- Модифицировать меш
- Запускать Блендер с командной строки и рендерить в фоновом режиме
- Обрабатывать параметры командной строки

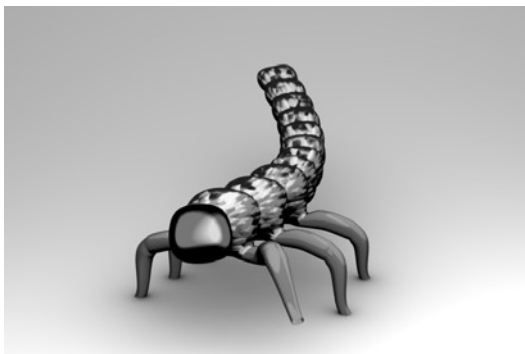
В следующей главе мы увидим, как можно назначать группы вершин и материалы нашим мешам.

## Группы вершин и материалы

Сложными мешами бывает трудно манипулировать, когда вершин очень много. В этой главе мы рассмотрим, как мы можем облегчить жизнь конечного пользователя, определяя группы вершины, чтобы пометить их наборы. Мы также изучим несколько видов использования групп вершин, включая их использование в арматурах и модификаторах, и мы взглянем на методы применения различных материалов к различным частям меша.

В этой главе мы изучим:

- Определение групп вершин
- Назначение вершин в группу
- Назначение материалов граням
- Назначение вершинных цветов вершинам
- Установка свойств рёбер
- Добавление модификаторов
- Покрытие костей кожей (оснастка меша)



### Группы вершин

**Группы Вершин** (Vertex groups) являются способом организации наборов вершин в пределах меша. Меш может иметь любое количество групп вершин, и любая вершина в пределах меша может быть членом более, чем одной группы вершин, или может не принадлежать никакой группе вершин совсем. Вновь созданный *Меш*-объект не содержит никаких определённых групп вершин.

В их основной форме, группы вершин являются ценным средством в идентификации определённых частей сложного меша. Назначая вершины в вершинные группы, разработчик модели в конечном счете обеспечивает людей, например, риггеров или текстурщиков, средствами для легкой идентификации и выбора частей модели, с которой они хотят работать.

Все же использование групп вершин идет гораздо дальше простой идентификации. Множество модификаторов мешей ограничивают своё влияние до определенной группы вершин, а арматуру можно сконфигурировать так, чтобы при деформации меша влияние каждой кости было привязано к единственной группе вершин. Мы увидим примеры этого позже.

Группа вершин является не просто набором вершин. С каждой вершиной в вершинной группе можно связать *вес* (*weight*, от нуля до единицы), который используется множеством модификаторов для более точной настройки их влияния. Вершина может иметь различные, связанные с ней, веса в каждой вершинной группе, которой принадлежит.

Жуки, которых мы создаем с помощью *creepycrawlies.py* - отличный пример довольно сложного меша со многими отчётливыми частями, для которых было бы очень полезно определить группы вершин. Не только для того, чтобы упростить выбор части по имени, например головы (*head*), но также, чтобы облегчить себе жизнь, если

мы хотим оснастить (rig) модель.

Наши основные инструменты в создании групп вершин - методы *Меш*-объектов, перечисленные в следующей таблице:

Метод	Действие	Замечания
<code>addVertGroup (group)</code>	Добавляет новую пустую группу вершин.	
<code>assignVertsToGroup (group, vertices, weight, mode)</code>	Добавляет список индексов вершин к существующей группе вершин с данным весом.	<i>Mode</i> (режим) определяет что делать, когда вершина уже является членом группы вершин. Смотри основной текст относительно деталей.
<code>getVertsFromGroup (group, weightsFlag=0, vertices)</code>	Возвращает список индексов вершин (по умолчанию) или список (индекс, вес) кортежей (если <code>weightsFlag = 1</code> ). Если список <i>vertices</i> определён, возвращаются только вершины, присутствующие в этом списке и в группе.	
<code>removeVertsFromGroup (group, vertices)</code>	Удаляет список вершин <i>vertices</i> из существующей группы вершины. Если список не определен, то удаляются все вершины.	
<code>renameVertGroup (groupName, newName)</code>	Переименовывает группу вершин	
<code>getVertGroupNames ()</code>	Возвращает список всех имен групп вершин.	
<code>removeVertGroup (group)</code>	Удаляет группу вершин	НЕ удаляет реальные вершины.

Важно понимать, что создание группы вершин и назначение вершин в неё - это два различных действия. Создание новой пустой группы вершин выполняется посредством вызова метода *addVertGroup()* вашего *Меш*-объекта. Он принимает единственную строку в качестве аргумента и она будет именем группы вершин. Если уже есть группа вершин с таким именем, к имени будет добавлен цифровой суффикс, чтобы предотвратить совпадение имён, например: *TailSegment* может стать *TailSegment.001*.

Добавление вершин в существующую группу вершин производится посредством вызова метода *assignVertsToGroup()* вашего меша. Этот метод принимает четыре обязательных аргумента - имя группы вершин, которой назначаются вершины, список

индексов вершин, вес, и *режим назначения*. Если группа вершин не существует, или один из индексов вершины указывает на несуществующую вершину, вызывается исключение.

Вес должен быть величиной между 0.0 и 1.0; любой вес больше, чем 1.0 отсекается до 1.0. Вес меньший или равный 0.0 удалит вершину из группы вершин. Если Вы хотите назначить различный вес вершинам в одной и той же группе вершин, Вы должны назначать их в группу с помощью отдельных вызовов метода *assignVertsToGroup()*.

*Режим назначения* (mode) бывает трёх видов: *ADD*, *REPLACE*, и *SUBTRACT*. *ADD* добавит новые вершины к группе вершин и свяжет с ними нужный вес. Если какие-нибудь из вершин в списке уже присутствуют, вес к ним будет добавлен. *REPLACE* заменит вес,

связанный с индексами в списке, если они входят в вершинную группу, или ничего не сделает в противном случае. *SUBTRACT* попытается вычесть вес у вершин в списке и снова ничего не сделает, если они не входят в группу вершин. Чаще всего при добавлении полностью новых групп вершин в меш Вы будете использовать режим *ADD*.

## Весомый вопрос

Для нашего первого примера мы добавим две новых группы вершин к существующему меш-объекту - одна будет содержать все вершины, которые имеют положительную x-координату, а другая будет содержать вершины с отрицательной x-координатой. Мы назовем эти группы **Right** и **Left** соответственно.

К тому же, мы дадим каждой вершине в этих группах вес в зависимости от их расстояния от центра объекта, с большим весом для вершин, которые находятся дальше от центра.

### Схема кода: *leftright.py*

Схематически мы предпримем следующие шаги:

1. Получить активный объект.
2. Проверить, что это - меш и получить меш-данные.
3. Добавить две новых группы вершин к объекту - *Left* и *Right*.
4. Для всех вершин в меше:
  1. Посчитать вес
  2. Если x-координата > 0:
  3. Добавить индекс вершины и вес в группу вершин *right*
  4. Если x-координата < 0:
  5. Добавить индекс вершины и вес в группу вершин *left*

Для того, чтобы убедиться, что новая группа вершин пуста, мы проверяем, существует ли уже эта группа, и в этом случае удаляем из неё вершины. Эта проверка выделена в коде:

```
def leftright(me,maximum=1.0):
    center=vec(0,0,0)
    left=[]
    right=[]
    for v in me.verts:
        weight=(v.co-center).length/maximum
        if v.co.x>0.0:
            right.append((v.index,weight))
        elif v.co.x<0.0:
            left.append((v.index,weight))
    return left,right

if __name__=="__main__":
    try:
        ob=Blender.Scene.GetCurrent().objects.active
        me=ob.getData(mesh=True)

        vgroups=me.getVertGroupNames()
        if 'Left' in vgroups:
            me.removeVertsFromGroup('Left')
        else:
            me.addVertGroup('Left')
        if 'Right' in vgroups:
            me.removeVertsFromGroup('Right')
        else:
            me.addVertGroup('Right')

        left,right=leftright(me,vec(ob.getSize()).length)

        for v,w in left:
            me.assignVertsToGroup('Left',[v],
                                  w,Blender.Mesh.AssignModes.ADD)
        for v,w in right:
            me.assignVertsToGroup('Right',[v],w,
                                  Blender.Mesh.AssignModes.ADD)

        Blender.Window.Redraw()

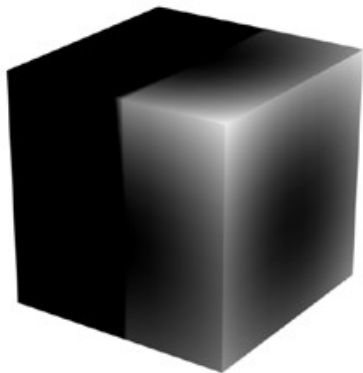
    except Exception as e:
        Blender.Draw.PupMenu('Error%t|'+str(e)[:80])
```

Полный скрипт доступен в файле *leftright.py*. Формуле,

вычисляющей вес, возможно, нужно некоторое объяснение: для того, чтобы назначить максимальный вес 1.0 в точке, лежащей на наибольшем расстоянии от центра объекта, мы должны масштабировать максимально возможным расстоянием. Мы могли бы пройтись циклом по всем вершинам, чтобы сначала определить максимум, но здесь мы решаем аппроксимировать этот максимум корнем от суммы квадратов размеров. Это заведомо больше максимального расстояния, так что максимальный вес, назначаемый любой из вершин, вероятно, будет меньше чем 1.0. Тем не менее, получение размера - значительно быстрее, чем расчет точного максимума для больших мешей. Также заметьте, что мы вычисляем расстояние до центра (центр объекта в режиме просмотра вершин в меше - всегда в (0, 0, 0)).

Он может отличаться от того, что пользователь может воспринимать как центр меша. (Центр объекта отображается как розовая точка в Блендере и может быть изменён, чтобы лежать в средней позиции всех вершин, с помощью **Object | Transform | Center new.**)

Результирующий вес для меша может выглядеть похожим на это:



## Модификаторы

Модификаторы - это инструменты, которые изменяют меш неразрушающим способом, и могут корректироваться интерактивно.

Другие объекты также могут иметь модификаторы: 3d-текст, Метаболлы и Кривые, например. Эти объекты могут быть представлены как сетки, так что их тоже можно модифицировать. Всё же не все модификаторы могут быть связаны с этими объектами. При желании, эффекты модификаторов можно сделать постоянными, применив их (apply). Блендер обеспечивает целый ряд модификаторов от subsurface до всех видов деформирующих модификаторов. Таблица показывает список доступных модификаторов:

Модификатор	Влияние групп вершин	Примечание
<i>displacement</i> (смещение)	да	
<i>curve</i> (кривая)	да	
<i>explode</i> (взрыв)	да	
<i>lattice</i> (решетка)	да	
<i>mask</i> (маскирование)	да	
<i>meshdeform</i> (деформация мешем)	да	
<i>shrinkwrap</i> (усаживающаяся упаковка)	да	
<i>simpledeform</i> (простая деформация)	да	
<i>smooth</i> (смягчение)	да	
<i>wave</i> (волна)	да	
<i>array</i> (массив)	нет	
<i>bevel</i> (скос или фаска)	нет	
<i>boolean</i> (объединение/пересечение/вычитание объектов)	нет	
<i>build</i> (построение)	нет	



<i>cast</i> (бросать)	нет	
<i>decimate</i> (уменьшение количества вершин)	нет	
<i>edgesplit</i> (разделение рёбер)	нет	
<i>mirror</i> (зеркально)	нет	
<i>subsurface</i> (подразделение поверхности)	нет	
<i>uvproject</i> (UV-проецирование)	нет	
<i>Particle system</i> (Система частиц)	да	На многие параметры можно воздействовать различными группами вершин
<i>armature</i> (арматура)	да	Влияние каждой кости может быть ограничено единственной группой вершин

Много модификаторов возможно настроить так, чтобы ограничить их влияние специфичной группой вершин, и есть несколько специальных модификаторов. Система частиц считается модификатором, хотя обычно системы частиц управляются через свой собственный набор инструментов. Также, связь с группами вершин у неё в некотором смысле обратная: вместо ограничивающего влияния на вершины в пределах группы вершин, веса вершин группы могут влиять на все типы параметров системы частицы, как например, плотность эмиссии и скорости частиц. Мы увидим пример этого в секции *Полёт искр*.

Модификаторы Арматуры также немного специфичны, так как они не ограничивают свое влияние единственной группой вершин. Тем не менее, их можно сконфигурировать так, чтобы ограничить влияние каждой отдельной кости на специфичную группу вершин, как мы изучим в секции *Кости*.

С точки зрения программиста на Питоне, список модификаторов

является свойством объекта (то есть, *НЕ* меша, лежащего в его основе). Объекты, ссылающиеся на один и тот же меш, могут иметь различные модификаторы. Этот список содержит объекты *Модификаторов*, и их можно добавлять к нему и удалять из него, а отдельные модификаторы можно перемещать вверх или вниз по списку. Порядок модификаторов в некоторых случаях важен. Например, при добавлении модификатора *subsurface* после зеркального модификатора результат может выглядеть отличающимся от того, что получится при добавлении зеркального модификатора перед модификатором *subsurface*.

Объект *модификатора* имеет тип и имя (первоначально представленное типом, но оно может быть установлено во что-то более подходящее). Тип - один из типов в списке констант в *Modifier.Types*. Каждый объект модификатора может иметь множество настроек, которые индексируются ключами, определёнными в *Modifier.Settings*. Не все настройки подходят для всех типов.

Если у нас было два объекта, меш-объект с именем *Target* (Цель) и объект-решетка (lattice) с именем *Deformer*, и мы хотели бы ассоциировать объект *Deformer* как модификатор решетки на объект *Target*, следующий кусок кода поможет достичь эту цель:

```
import Blender
from Blender import Modifier

target = Blender.Object.Get('Target')
deformer= Blender.Object.Get('Deformer')

mod = target.modifiers.append(Modifier.Types.LATTICE)
mod[Modifier.Settings.OBJECT] = deformer
target.makeDisplayList()
Blender.Window.RedrawAll()
```

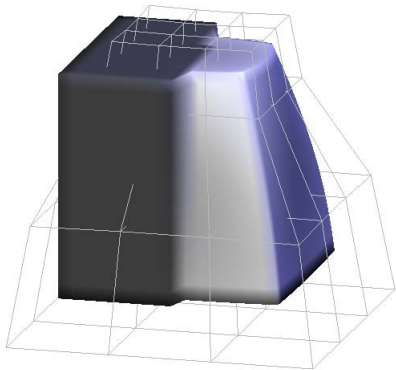
Если объект *Target* имел группу вершин с именем *Right*, состоящую из вершин в правой половине объекта, мы могли бы ограничить влияние модификатора, изменив атрибут VERTGROUP. Наш кусок кода должен измениться таким образом (дополнительная строка выделена):

```
import Blender
from Blender import Modifier

target = Blender.Object.Get('Target')
deformer= Blender.Object.Get('Deformer')

mod = target.modifiers.append(Modifier.Types.LATTICE)
mod[Modifier.Settings.OBJECT] = deformer
mod[Modifier.Settings.VERTGROUP] = 'Right'

target.makeDisplayList()
Blender.Window.RedrawAll()
```



## Гравировка

Рассмотрим следующую проблему: дан некоторый текст, мы хотим отрендерить этот текст в виде утопленных бороздок на поверхности, подобно тому, как если бы они были выгравированы. Это не так уж просто, как кажется. Конечно, достаточно просто создать текстовый объект, но для того, чтобы манипулировать этим текстом, мы хотели бы преобразовать этот текстовый объект в меш. Интерфейс Блендера предлагает эту возможность в меню объектов, но как ни странно, API Блендера не предоставляет эквивалентной функции. Так, нашим первым барьером будет преобразование текстового объекта в меш.

Вторая проблема, которую мы должны решить - как выдавить набор вершин или рёбер на нужную глубину от поверхности. Опять

же, в API Блендера нет функции для этого, так что мы должны добавить её к нашему пакету инструментов сами.

Последняя проблема более тонкая. Если нам каким-нибудь образом удалось создать несколько утопленных канавок, мы могли бы захотеть сделать краями чуть менее острыми, так как в действительности ничто не имеет абсолютно острых краёв. Существуют различные способы добиться этого, но многие из них включают добавление модификатора в наш меш. **Модификатора скоса bevel** может быть достаточно, чтобы убрать только острые края, но вполне вероятно, мы хотели бы добавить модификатор subsurface к нашему мешу целиком. Здесь у нас есть проблема: при заполнении промежутков между символами нашего текста, весьма вероятно, что мы столкнёмся со множеством узких треугольников. Эти треугольники испортят внешний вид результата нашего модификатора subsurface, как можно увидеть на следующем рисунке:



Две вещи могли бы помочь смягчить эту проблему. Одна - это добавить вес crease (складки) к рёбрам нашего выгравированного текста, этим самым взвешивая эти края сильнее, чем при расчете subsurface по умолчанию. Это может помочь, но может также отодвинуть нас от цели применения модификатора, так как это делает эти края острыми. Следующий рисунок показывает результат: лучше, но все еще не выглядит идеально.



Другим подходом будет добавить дополнительный рёберный цикл ровно за пределами выгравированного текста. Это добавит кольцо четырёхугольных граней вокруг текста, заставляя subsurface вокруг текста вести себя намного лучше, как это можно увидеть ниже. В нашей конечной реализации мы применяем оба решения, но сначала мы примемся за каждую задачу поочередно.



### Конвертация объекта Text3d в меш

Объект *Text3d* базируется на кривой с несколькими дополнительными параметрами. Блок данных, на который он ссылается - объект Кривой Блендера (*Curve*), и как только мы узнаем, как получить доступ к индивидуальным частям кривой, которые составляют каждый символ в нашем тексте, мы можем преобразовать эти кривые в вершины и рёбра. Все соответствующие функциональные возможности могут быть найдены в модулях *Blender.Curve* и *Blender.Geometry*.



В Блендере, отношение между объектом *Text3d* и объектом *Curve* (Кривой) более тонкое и запутанное, чем описано в основном тексте. Объект *Text3d* - специализированная версия объекта *Curve*, подобно подклассу на объектно-ориентированном языке. Тем не менее, в API Блендера объект *Text3d* не является подклассом *Curve*, как и нет у него дополнительных атрибутов, доступных на том же экземпляре объекта. Звучит запутанно? Так и есть. Как же тогда Вы извлечете все атрибуты? Весь фокус в том, что вы можете использовать имя объекта *Text3d*, чтобы получить доступ к связанному с ним объекту *Curve*, как показывает этот маленький пример:

```
txt = ob.getData()  
curve = Blender.Curve.Get(txt.getName())
```

Теперь мы можем использовать *txt*, чтобы иметь доступ к *Text3d*-специфичной информации (например, *txt.setText('foo')*) и *curve*, чтобы иметь доступ к *Curve*-специфичной информации (например, *curve.getNumCurves()*).

Объект *Curve* Блендера состоит из множества объектов *CurNurb*, которые представляют сегменты кривой. Единственный текстовый символ обычно состоит из одного или двух сегментов кривой. Маленькая буква *e*, например, состоит из внешнего сегмента и небольшого внутреннего сегмента кривой. Объекты *CurNurb*, в свою очередь, состоят из множества **узлов** или **управляющих точек**, которые задают сегмент кривой. В случае объектов *Text3d* эти узлы всегда являются объектами *BezTriple*, и модуль *Geometry* Блендера предоставляет нам функцию *BezierInterp()*, которая возвращает список координат, интерполированных между двумя точками. Эти точки и направляющие кривой в этих точках (часто называемые **handle**, рукоять), можно взять из объектов *BezTriple*. Результирующий код выглядит так (полный код является частью нашего пакета разработчика в *Tools.py*) (Эта и последующие функции этого раздела, несмотря на заверения автора, отсутствуют в файле *Tools.py*, прилагаемом ко 2-й главе, найти их можно только в файле *engrave.py* — прим. пер.):

```
import Blender  
from Blender.Geometry import BezierInterp as interpolate  
from Blender.Mathutils import Vector as vec  
  
def curve2mesh(c):  
    vlists=[]  
    for cn in c:  
        npoints = len(cn)  
  
        points=[]  
        first=True  
        for segment in range(npoints-1):  
            a=cn[segment].vec  
            b=cn[segment+1].vec  
            lastpoints = interpolate(vec(a[1]),vec(a[2]),
```

```

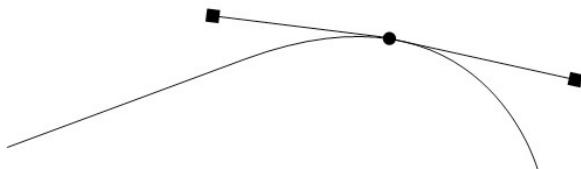
        vec(b[0]), vec(b[1]), 6)
    if first:
        first = False
        points.append(lastpoints[0])
        points.extend(lastpoints[1:])
    if cn.isCyclic():
        a=cn[-1].vec
        b=cn[0].vec
        lastpoints=interpolate(vec(a[1]), vec(a[2]),
                               vec(b[0]), vec(b[1]), 6)
        points.extend(lastpoints[:-2])

    vlists.append(points)

return vlists

```

Выделенные строки показывают два важных аспекта. Первая показывает фактическую интерполяцию. Мы переименовали довольно неуклюжее имя функции *BezierInterp()* в *interpolate()*, и она принимает пять аргументов. Первые четыре берутся от двух объектов *BezTriple*, между которыми мы интерполируем. В каждом объекте *BezTriple* можно получить доступ к списку из трех векторов: входящая рукоять, позиция точки, и исходящая рукоять (смотри следующий рисунок). Мы передаем позицию первой точки и исходящей рукояти и позицию второй точки и входящей рукояти. Пятый аргумент является количеством точек, которые мы хотим получить на выходе функции *interpolate()*.



Вторая выделенная строка заботится о **замкнутых кривых** - кривых, в которых их первые и последние точки связаны. Это является случаем всех кривых, которые формируют символы в тексте. Функция возвращает список списков. Каждый список содержит все интерполированные точки (кортежи из x, y, z координат)

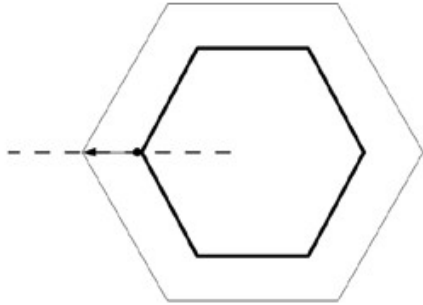
для каждой кривой. Заметьте, что некоторые символы состоят из более, чем одной кривой. Например, небольшая буква *e* во многих шрифтах, или буква *o* состоит из двух кривых, одна задаёт внешнюю границу буквы и одна внутреннюю. Объект *Text3d*, содержащий текст *Foo*, например, возвращает список из пяти списков - первый будет содержать вершины, определяющие большую букву *F*, а второй и третий будут содержать вершины для двух кривых, которые создают маленькую букву *o*, так же будет с четвертым и пятым.

### Выдавливание рёберного цикла

**Выдавливание (Extrusion)** является процессом, где мы дублируем вершины (и, возможно, соединяющие их рёбра) и перемещаем их в некотором направлении, после чего мы соединяем эти дубликаты вершин с их оригиналами новыми рёбрами, и заканчиваем операцию, создавая новую грань между старыми и новыми вершинами. Нам нужно это для того, чтобы утопить очертание нашего текста, чтобы создать бороздку с вертикальными стенками. Функция *extrude\_selected\_edges()* в *Tools.py* принимает меш и вектор как аргументы, и выдавит вершины на выбранных рёбрах в меше в направлении вектора, добавляя все необходимые новые рёбра и грани. Поскольку эта техника является расширением того, что мы уже видели раньше, код не показан здесь.

### Расширение (Expanding) рёберного цикла

Если у нас есть список рёбер, формирующих замкнутую кривую (или более одного), определяющий символ, мы хотели бы окружить эти рёбра дополнительным рёберным циклом, чтобы создать лучшее "выполнение" любого модификатора *subsurface*, который конечный пользователь может связать с нашим мешем. Это был бы довольно сложный процесс, если мы должны были бы вычислять это в 3D, но, к счастью, наши преобразованные символы имеют все свои вершины на плоскости *xy* (дело в том, что все символы в новых экземплярах *Text3d* объекта лежат на плоскости *xy*)..



Всего лишь два измерения - это вполне податливая проблема. Для каждой точки в нашем рёберном цикле мы определяем направление вершинной нормали. **Вершинная нормаль** является линией, разрезающей пополам угол между двумя рёбрами, которые делят рассматриваемую нами точку. Если два ребра коллинеарны (или почти так), мы берем за вершинную нормаль линию, перпендикулярную одному из рёбер. Позиция точки, создаваемой в новом рёберном цикле, будет где-нибудь на этой нормали. Для того, чтобы определиться, должны ли мы перемещать наружу или внутрь вдоль этой нормали, мы просто пробуем одно направление и проверяем новую позицию - находится ли она внутри границ нашего символа. Если это так, мы берём обратное направление.

Один вопрос по-прежнему нуждается в решении: символ может состоять из более, чем одной кривой. Если мы хотим сделать дополнительные рёберные циклы вокруг такого символа, такой рёберный цикл должен быть снаружи внешней границы символа, но внутри любой внутренней кривой. Другими словами, если мы создаем новый рёберный цикл, мы должны знать, лежит ли кривая внутри другой кривой. Если это так, то она не является внешней границей, и новый рёберный цикл должен быть создан лежащим внутри кривой. Следовательно, наша функция *expand()* (показанная в следующем куске кода, полный код является частью *Tools.py*. На самом деле эта и все вызываемые ею функции находятся в файле *expand.py* — прим. пер.), берет дополнительный опциональный аргумент *plist*, который является списком списков, содержащих объекты *MVert*, определяющие дополнительные полигоны, чтобы

сверяться с ними. Если первая точка кривой, которую мы хотим расширить, лежит в пределах любой из этих дополнительных кривых, мы принимаем, что кривая, которую мы расширяем, является **внутренней кривой**. (Это будет неверным предположением, если внутренняя кривая будет пересекать внешнюю кривую в некоторой точке, но для кривых, определяющих символ в шрифте, такого никогда не происходит.)

```
def expand(me, loop, offset=0.05, plist=[]):
```

```
    ov = [me.verts[i] for i in verts_from_edgeloop(loop)]
```

```
    inside=False
```

```
    for polygon in plist:
```

```
        if in_polygon(loop[0].v1.co, polygon):
```

```
            inside=True
```

```
            break    # мы не имеем дел с несколькими
```

```
включениями
```

```
    n=len(ov)
```

```
    points=[]
```

```
    for i in range(n):
```

```
        va = (ov[i].co-ov[(i+1)%n].co).normalize()
```

```
        vb = (ov[i].co-ov[(i-1)%n].co).normalize()
```

```
        cosa=abs(vec(va).dot(vb))
```

```
        if cosa>0.99999 :    # почти коллинеарны
```

```
            c = vec(va[1],va[0],va[2])
```

```
        else:
```

```
            c = va+vb
```

```
        l = offset/c.length
```

```
        p = ov[i].co+l*c
```

```
        if in_polygon(p,ov) != inside:
```

```
            p = ov[i].co-l*c
```

```
        print i,ov[i].co,va,vb,c,l,cosa,p
```

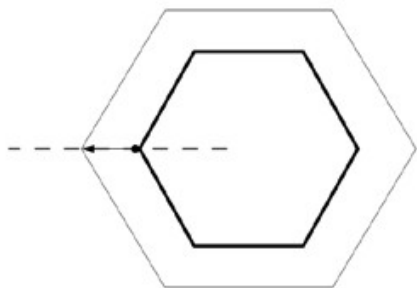
```
        points.append(p)
```

```
    return points
```

Выделенный код вызывает функцию (приведенную в *Tools.py*), которая принимает список рёбер, формирующих рёберный цикл, и возвращает отсортированный список вершин. Это необходимо, поскольку наша функция *in\_polygon()* принимает список вершин, а не рёбер, и предполагает, что этот список отсортирован, то есть

смежные вершины формируют рёбра, которые не пересекаются.

Чтобы определить, находится ли точка внутри замкнутого многоугольника, определяемого списком вершин, мы считаем количество рёбер, которые пересекаются линией (часто называемой **лучом**), которая начинается в данной точке и распространяется до бесконечности. Если количество пересекаемых рёбер нечетное, точка лежит внутри многоугольника; если четное, она лежит снаружи многоугольника. Следующий рисунок иллюстрирует концепцию:



Функция `in_polygon()`, показанная здесь - часть `Tools.py`. Она принимает точку (*Вектор*) и список вершин (объекты `MVert`) и возвращает или *Истину* или *Ложь*. Заметьте, что любая z-координата у точки или у вершины в многоугольнике игнорируются.

```
from Blender.Geometry import LineIntersect2D
from Blender.Mathutils import Vector as vec
```

```
def in_polygon(p, polygon):
    intersections = 0
    n = len(polygon)
    if n < 3 : return False
    for i in range(n):
        if LineIntersect2D (p,vec(1.0,0.0,0.0),polygon[i].
co,polygon[(i+1)%n].co):
            intersections+=1
    return intersections % 2 == 1
```

Трудная задача выполняется на выделенной строке функцией `LineIntersect2D()`, доступной в модуле `Blender.Geometry`. Действие *деление по модулю (%)* в операторе `return` - способ определить,

нечетное ли количество пересечений.

### **Собираем всё вместе: *Engrave.py***

Вооруженные всеми вспомогательными функциями, разработанными в предыдущих секциях, мы можем сделать список шагов, которые мы должны предпринять для того, чтобы выгравировать текст:

1. Показать всплывающее меню для ввода строки, которую надо гравировать.
2. Проверить, что активный объект - меш, и выбраны грани.
3. Создать объект `Text3d`.

(на самом деле скрипт `engrave.py` требует, чтобы объект `Text3d` уже был создан и выбран как активный, так что первые 3 пункта не полностью соответствуют действительности — прим. пер.)

4. Преобразовать его в меш, с подходящими группами вершин.
5. Добавить дополнительные рёберные циклы к символам.
6. Выдавить оригинальные символы вниз.
7. Заполнить низ выдавленных символов.
8. Добавить "cartouche" (прямоугольник) вокруг текста.
9. Заполнить пространство между cartouche и символами.
10. Добавить модификатор `subsurface`.
11. Установить величину `crease` (складки) на рёбрах, содержащихся в группах вершин `TextTop` и `TextBottom`.

Наш окончательный скрипт следует за этой схемой почти в точности и использует инструменты, которые мы разработали раньше в этой главе. Мы покажем здесь наиболее важные секции (полный скрипт доступен как `engrave.py`). Мы начинаем с преобразования объекта `Text3d` (с в следующем коде) в список, содержащий список позиций вершин для каждого сегмента кривой в тексте, и мы добавляем новый пустой *Меш*-объект в сцену с несколькими пустыми группами вершин:

```
vlist = curve2mesh(c)

me = Blender.Mesh.New('Mesh')
ob = Blender.Scene.GetCurrent().objects.new(me, 'Mesh')

me.addVertGroup('TextTop')
me.addVertGroup('TextBottom')
me.addVertGroup('Outline')
```

Следующий шаг должен добавить эти вершины в меш и создать соединяющие рёбра. Так как все сегменты кривой в символе замкнуты, мы должны позаботиться о добавлении дополнительного ребра, чтобы соединить мостом промежуток между последней и первой вершиной, как показано на выделенной строке. На всякий случай, мы удаляем любые задвоения, которые могут присутствовать в интерполированном сегменте кривой. Мы добавляем вершины к группе вершин *TextTop* и сохраняем ссылку на список новых рёбер для будущего использования.

```
loop=[]
for v in vlist:
    offset=len(me.verts)
    me.verts.extend(v)
    edgeoffset=len(me.edges)
    me.edges.extend([(i+offset,i+offset+1)
                     for i in range(len(v)-1)])
    me.edges.extend([(len(v)-1+offset,offset)])
    me.remDoubles(0.001)

    me.assignVertsToGroup('TextTop',
                          range(offset,len(me.verts)),
                          1.0,
                          Blender.Mesh.AssignModes.ADD)
    loop.append([me.edges[i] for i in range(edgeoffset,
                                             len(me.edges) )])
```

Для каждого рёберного цикла, который мы сохранили в предыдущей части, мы создаем новый, и немного больший, рёберный цикл вокруг него и добавляем эти новые вершины и рёбра к нашему мешу. Мы также хотим создать грани между этими рёберными циклами, и это действие начинается на выделенной строке: здесь мы используем встроенную функцию Питона *zip()*, чтобы получить пары рёбер двух рёберных циклов. Каждый

рёберный цикл упорядочен вспомогательной функцией (доступной в *Tools.py*), которая сортирует рёбра, чтобы они лежали в порядке, в котором они соединены друг с другом. Для каждой пары рёбер мы создаем две возможных организации индексов вершин и вычисляем, какая из них формирует нескрученную грань. Это вычисление производится посредством функции *least\_warped()* (код не показан), которая основана на сравнении периметров граней, заданных двумя различными порядками вершин. Нескрученная грань будет иметь самый короткий периметр, именно её мы затем добавляем к мешу.

```
for l in range(len(loop)):
    points = expand.expand(me,loop[l],
                           0.02,loop[:l]+loop[l+1:])

    offset=len(me.verts)
    me.verts.extend(points)
    edgeoffset=len(me.edges)
    me.edges.extend([(i+offset,i+offset+1)
                     for i in range(len(points)-1)])
    me.edges.extend([(len(points)-1+offset,offset)])
    eloop=[me.edges[i] for i in
           range(edgeoffset,len(me.edges))]
    me.assignVertsToGroup('Outline',
                          range(offset,len(me.verts)),
                          1.0,
                          Blender.Mesh.AssignModes.ADD)

    faces=[]
    for e1,e2 in zip( expand.ordered_edgeloop(loop[l]),
                     expand.ordered_edgeloop(eloop)):
        f1=(e1.v1.index,e1.v2.index,
             e2.v2.index,e2.v1.index)
        f2=(e1.v2.index,e1.v1.index,
             e2.v2.index,e2.v1.index)
        faces.append(least_warped(me,f1,f2))
    me.faces.extend(faces)
```

Мы опустили код выдавливания рёберной петли символа, но следующие строки содержательны, так как они показывают, как заполняется рёберный цикл. Сначала мы выбираем все важные рёбра, используя две вспомогательные функции (это - выдавленные рёбра символов). Затем, мы вызываем метод *fill()*. Этот метод будет заполнять любой набор замкнутых рёберных циклов до тех пор, пока они лежат в одной плоскости. Он даже позаботится об отверстиях



(подобно небольшому острову в букве е):

```
deselect_all_edges(me)
select_edges(me, 'TextBottom')
me.fill()
```

Дополнение cartouche - просто вопрос добавления прямоугольного рёберного цикла вокруг наших символов. Если этот рёберный цикл выбрать вместе с вершинами в группе вершин *Outline*, можно снова использовать метод *fill()* для заполнения этого cartouche. Это не показано здесь. Несколько заключительных штрихов: мы по возможности преобразуем треугольники в нашем меше в четырехугольники, используя метод *triangleToQuad()*, затем подразделяем меш. Мы также добавляем модификатор *subsurface*, устанавливаем атрибут сглаживания (*smooth*) на всех гранях и пересчитываем нормали всех граней, чтобы они согласованно указывали наружу.

```
me.triangleToQuad()
me.subdivide()

mod = ob.modifiers.append(
    Blender.Modifier.Types.SUBSURF)
mod[Blender.Modifier.Settings.LEVELS]=2

select_all_faces(me)
set_smooth(me)
select_all_edges(me)
me.recalcNormals()
```

### Скрытый модификатор Захвата:



Мы видели, что модификаторы, доступные в Блендере, можно добавлять к объекту в Питоне. Есть, тем не менее, один модификатор, который может быть добавлен, но создаётся впечатление, что он не имеет эквивалента в графическом интерфейсе Блендера. Это - так называемый **модификатор Hook** (Захват). Захват в Блендере - способ сделать родителем вершин объект (так что это противоположно *vertex parenting*, где мы родителем объекта назначаем вершины), и в приложении самостоятельно может быть доступно через меню **Mesh | Vertex | Add Hook** в режиме редактирования. После добавления он появится в списке модификаторов. С точки зрения программиста,

модификатор Захвата никак не отличается из других модификаторов, но увы, ни его тип, ни параметры, не документированы в API.

### Добавление переводчика к разделу Гравировка:

*К сожалению, опробованная мной программа engrave.py (с необходимым ей модулем expand.py), скачанная с сайта издательства, работала далеко не так красиво, как это описано в тексте. В очередной раз придётся набраться наглости и указать на недоработки автора.*

1. Простая ошибка в программе: ближе к концу есть такие строки:

```
me.subdivide()
me.triangleToQuad()
me.subdivide()
```

*перед преобразованием в четырёхугольники, и тем более, перед подразделением необходимо было выделить все вершины, а к этому моменту они выделены все, кроме основного контура букв. В результате, меш подразделяется на устрашающее количество лишних треугольников. Я заменил первое подразделение на выбор всех рёбер.*

```
select_all_edges(me)
me.triangleToQuad()
me.subdivide()
```

2. Расширение или окантовка некоторых символов происходила *внутрь*, а не наружу, как положено, т.е. проверка на то, является ли контур внутренним, не всегда срабатывала правильно. На мой взгляд, проблема состоит в этой строке функции *in\_polygon()* модуля *expand*:



```
if cross(p,vec(1.0,0.0,0.0),polygon[i].co,  
        polygon[(i+1)%n].co):
```

*Насколько я понял, второй конец проверяемого луча **вес(1.0,0.0,0.0)** взят произвольно, и это вызывает накладку в отдельных случаях. Для себя я просто поставил более удалённый вектор **вес(1000.0,0.0,0.0)**, и программа в моём тестовом случае перестала ошибаться. В общем же случае программа должна сама по некоторому алгоритму вычислять этот вектор так, чтобы наверняка исключить возможность ошибок.*

*3. Самое страшное: заполнение пространства между буквами выполняется пресловутой функцией **fill()** (её аналог в интерфейсе Блендера - Shift-F), результатом которой и являются множество треугольников с очень острыми углами. С тем же успехом можно было сразу применить булеановское вычитание и не мучиться. Возможно, добавлением специальной функции красивого заполнения проблему можно решить, но, думаю, такая функция вряд-ли будет простой.*

## Полет искр

Искры и все яркие эффекты подобного рода легко можно создать добавлением подходящей системы частиц к объекту. Множеством параметров систем частиц можно управлять с помощью весов в группе вершин, включая локальную плотность испускаемых частиц.

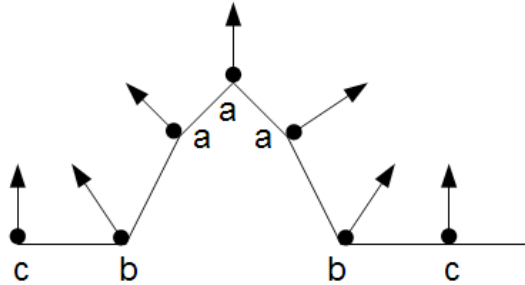
В этом примере мы хотели бы имитировать поведение электрического феномена, называемого "Огни святого Эльма". Это такой эффект, когда при определенных обстоятельствах, особенно в начале грозы, некоторые объекты начинают светиться. Это свечение называется **коронный разряд** (см., например, [http://ru.wikipedia.org/wiki/Огни\\_святого\\_Эльма](http://ru.wikipedia.org/wiki/Огни_святого_Эльма)), и наиболее заметно на острых и выступающих частях более крупных структур, например, на радиоантеннах или громоотводах, где электрическое поле, которое вызывает этот эффект, наиболее сильное.

Для того, чтобы правдоподобно влиять на количество частиц, испускаемых мешем, нам нужно вычислять величину, называемую локальная кривизна, и хранить эту кривизну, нужным образом

отмасштабированную, как вес в группе вершин. Затем, мы можем применить эту группу вершин к параметру плотности на **дополнительной** панели контекста частиц, чтобы управлять эмиссией.

Меш может иметь любую форму, и в большинстве случаев нет хорошей формулы, которая бы аппроксимировала его форму. Следовательно, мы аппроксимируем локальную кривизну неизбежно грубым способом (если нужна дополнительная информация и немного тяжелой математики, смотрите [http://en.wikipedia.org/wiki/Mean\\_curvature](http://en.wikipedia.org/wiki/Mean_curvature)), вычисляя среднюю рёберную кривизну всех связанных с вершиной рёбер для каждой вершины в меше. Здесь мы определяем **рёберную кривизну** как скалярное произведение нормализованной вершинной нормали и вектора ребра (то есть, вектор, формируемый от вершины к её соседке). Это произведение будет отрицательным, если ребро изгибается вниз относительно нормали, и положительным, если оно изгибается вверх. Мы обратим этот знак, так как нам более привычно понятие положительной кривизны для пиков, а не для впадин. По-другому можно посмотреть на это так: в областях положительной кривизны угол между вершинной нормалью и ребром, начинающемся в той же вершине, больше 90°.

Следующий рисунок иллюстрирует концепцию - он изображает серию вершин, связанных рёбрами. У каждой вершины показана связанная с ней вершинная нормаль (стрелками). Вершины, обозначенные как **a**, имеют положительную кривизну, те, что обозначены **b** - отрицательную кривизну. Две из показанных вершин помечены буквой **c**, они находятся в области нулевой кривизны - в этих местах поверхность плоская, и вершинная нормаль перпендикулярна рёбрам.



## Расчет локальной кривизны

Функцию, которая вычисляет локальную кривизну для каждой вершины в меше, и возвращает список нормализованных весов, можно осуществить следующим образом:

```
from collections import defaultdict
def localcurvature(me, positive=False):

    end=defaultdict(list)
    for e in me.edges:
        end[e.v1.index].append(e.v2)
        end[e.v2.index].append(e.v1)

    weights=[]
    for v1 in me.verts:
        dvpn = []
        for v2 in end[v1.index]:
            dv = v1.co-v2.co
            dvpn.append(dv.dot(v1.no.normalize()))
        weights.append((v1.index, sum(dvpn)/max(len(dvpn),
            1.0)))

    if positive:
        weights = [(v, max(0.0, w)) for v, w in weights]

    minimum = min(w for v, w in weights)
    maximum = max(w for v, w in weights)
    span = maximum - minimum
```

```
if span > 1e-9:
    return [(v, (w-minimum)/span) for v, w in weights]
return weights
```

Функция *localcurvature()* принимает меш и один опциональный аргумент, и возвращает список кортежей с индексом вершины и её весом. Если дополнительный аргумент - *Истина*, любой рассчитанный отрицательный вес отвергается.

Сложная работа выполняется на выделенных строках. Здесь мы проходим циклом над всеми вершинами, и затем, во внутреннем цикле, проверяем каждое связанное с текущей вершиной ребро, чтобы извлечь вершину на другом конце из предварительно рассчитанного словаря. Затем мы вычисляем *dv* как рёберный вектор и добавляем скалярное произведение этого рёберного вектора и нормализованной вершинной нормали в список *dvpn*.

```
weights.append((v1.index, sum(dvpn)/max(len(dvpn), 1.0)))
```

Предшествующая строка может выглядеть странно, но она добавляет кортеж, состоящий из индекса вершины и средней кривизны, где среднее число получается вычислением суммы всех величин кривизны по каждому ребру из списка, и деления её на количество величин в списке. Поскольку список может быть пустым (это случается, когда меш содержит не связанные вершины), мы предохраняемся от ошибки деления на 0, деля её на длину списка или на единицу, в зависимости от того, что больше. Таким образом, мы сохраняем наш код более удобочитаемый, избегая оператора *if*.

## Схема кода: *curvature.py*

С функцией *localcurvature()* в нашем распоряжении, сам скрипт вычисления кривизны становится совсем кратким (полный скрипт доступен как *curvature.py*):

```
if __name__ == "__main__":
    try:
        choice = Blender.Draw.PupMenu("Normalization\tOnly
                                         positive|Full range")

        if choice>0:
            ob = Blender.Scene.GetCurrent().objects.active
            me = ob.getData(mesh=True)
```

```

try:
    me.removeVertGroup('Curvature')
except AttributeError:
    pass

me.addVertGroup('Curvature')

for v,w in localcurvature(me,
                           positive=(choice==1)):
    me.assignVertsToGroup('Curvature',[v],w,
                          Blender.Mesh.AssignModes.ADD)

Blender.Window.Redraw()

except Exception as e:
    Blender.Draw.PupMenu('Error%t|'+str(e)[:80])

```

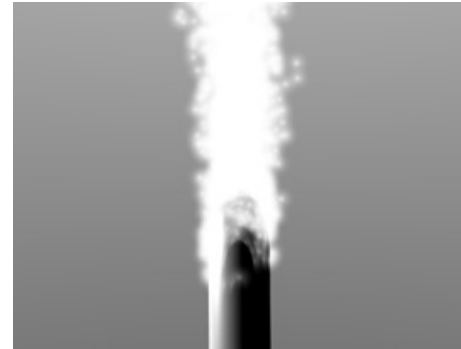
Выделенные строки показывают, что мы удаляем возможно существующую группу вершин *Curvature* из Меш-объекта внутри блока *try*, и отлавливаем исключение *AttributeError*, которое будет вызвано, если группа отсутствует. Затем, мы снова добавляем группу с тем же именем, так что она будет полностью пустая. Последняя выделенная строка показывает, как мы добавляем отдельно каждую вершину, поскольку любая вершина может иметь отличающийся от других вес.



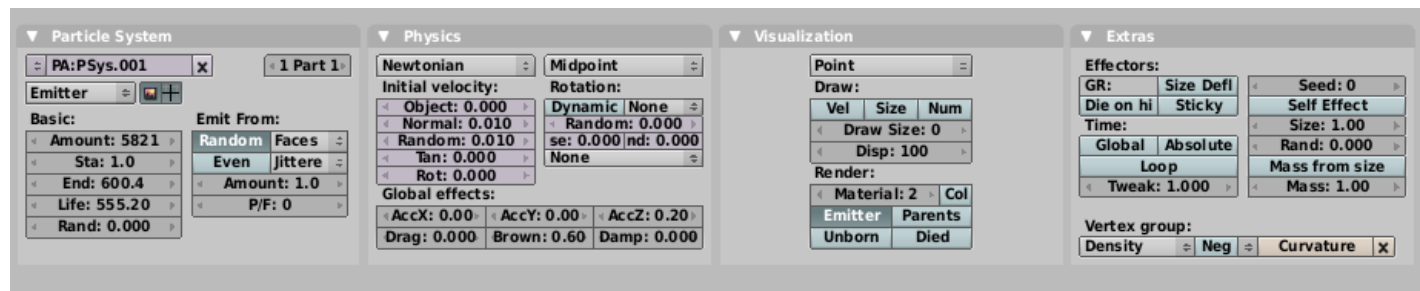
Все действия окружены конструкцией *try ... except*, которая поймает любые исключения, и они появятся во всплывающем информационном сообщении, если произойдет что-то необычное. Наиболее вероятно, это будет в ситуациях, когда пользователь забудет выбрать Меш-объект.

## Собираем всё это вместе: Огни святого Эльма

Иллюстрация испускания из заостренного стержня была сделана моделированием простого объекта стержня вручную, и, затем, вычислением кривизны с помощью *curvature.py*.



Затем, была добавлена система частиц и параметр плотности (*density*) в панели Extra был настроен на группу вершин *Curvature*. Стержню и системе частиц были даны отдельные материалы: простой серый и белое Хало соответственно. Частицы были симулированы для 250 кадров, и для иллюстрации представлен кадр 250.



## Кости

Арматура может считаться основой анимации, поскольку деформирует меш управляемым способом, который можно задавать ключами в данных кадрах, необходима для аниматоров, чтобы придавать позы их персонажам удобно контролируемым способом.

Реализация арматуры Блендера обеспечивает риггера и аниматора подавляюще большим количеством возможностей, но в конце концов арматура в первую очередь набор связанных костей, где каждая кость деформирует часть меша. Перемещения этих костей друг относительно друга могут быть обусловлены несколькими различными ограничениями.

Хотя кости можно конфигурировать для работы так, чтобы они влияли через **envelope** (конверт), тем самым деформируя любую вершину целевого меша в пределах определенного радиуса, их можно также сконфигурировать, чтобы деформировать только те вершины, которые принадлежат группе вершин с именем, совпадающим с именем этой кости. Такая деформация в дальнейшем управляется весом вершины в группе вершин, давая нам возможность точной настройки влияния кости.

## Тик-так

Чтобы проиллюстрировать основные возможности арматуры, мы создадим риг простой модели часов. Часы - это единый меш, состоящий из трех отдельных, не соединённых между собой субмешей - body (тело), little hand (маленькая рука), и big hand (большая рука). *(Здесь автор, типа, пошутил. В английском языке стрелки часов почему-то называются «hand», что одновременно означает «ладонь» или «рука». Ну в статье стрелки и выполнили в виде реальных рук. Я долго не мог решить, как же лучше перевести эти little hand и big hand. - прим. пер.)* Вершины каждой руки часов принадлежат двум отдельным вершинным группам - одна половина часовой руки (*Arm*), подключена к центру часов, и для конца руки (или ладони, *Hand*) отдельно. Эта настройка позволяет создать мультяшную анимацию наподобие карикатуры, где мы, например, можем сделать след конца руки фактическим движением.

### Схема кода: clock.py

Мы должны предпринять следующие шаги, чтобы оснастить наши часы предлагаемым способом:

1. Импортировать данные меша
2. Создать меш часов

3. Создать вершинные группы
4. Создать объект арматуры
5. Создать кости в составе арматуры.
6. Связать модификатор с арматурой

Перевод из схемы в код - почти один в один, только нужно повторить множество инструкций для каждой из костей (полный код доступен как clock.py):

```
me=Blender.Mesh.New('Clock')
me.verts.extend(clockmesh.Clock_verts)
me.faces.extend(clockmesh.Clock_faces)
```

```
scn=Blender.Scene.GetCurrent()
ob=scn.objects.new(me)
scn.objects.active=ob
```

```
me.addVertGroup('BigHand')
me.assignVertsToGroup('BigHand',
    clockmesh.Clock_vertexgroup_BigHand,
    1.0, Blender.Mesh.AssignModes.ADD)
```

... <аналогичный код для вершинных групп LittleHand, BigArm и LittleArm опущен> ...

```
ar = Blender.Armature.New('ClockBones')
ar.envelopes=False
ar.vertexGroups=False
obbones = scn.objects.new(ar)
```

```
mod = ob.modifiers.append(Blender.Modifier.Types.ARMATURE)
mod[Blender.Modifier.Settings.OBJECT]=obbones
mod[Blender.Modifier.Settings.ENVELOPES]=False
mod[Blender.Modifier.Settings.VGROUPS]=True
```

```
ar.makeEditable()
bigarm = Blender.Armature.Editbone()
bigarm.head = vec(0.0,0.0 ,0.57)
bigarm.tail = vec(0.0,0.75,0.57)
ar.bones['BigArm'] = bigarm
bighand = Blender.Armature.Editbone()
    bighand.head = bigarm.tail
```

```
bighand.tail = vec(0.0,1.50,0.57)
bighand.parent = bigarm
ar.bones['BigHand'] = bighand
```

... <аналогичный код для маленькой руки опущен> ...

```
ar.update()
```

```
obbones.makeParent([ob])
```

Важные моменты выделены. Сначала, мы отключаем *envelopes* и свойства *vertexGroups* у *объекта* арматуры. Это может показаться странным, но эти свойства являются остатками от того времени, когда арматура не была модификатором, приложенным к мешу, а работала через родительское (parented) влияние на Меш-объект (по крайней мере, насколько Я могу судить, доступная документация немного невнятна в этом месте). Мы определяем, какое влияние использовать, устанавливая свойства в *модификаторе арматуры*.

После связывания арматурного модификатора с нашим Меш-объектом, мы создадим нашу арматуру кость за костью. Прежде, чем мы добавим какие-либо кости в арматуру, мы должны вызвать её метод *makeEditable()*. Заметьте, что этот **режим редактирования для арматур** отличен от режима редактирования для других объектов, которые можно задавать с помощью функции *Blender.Window.editMode()*! Как только мы закончим, мы возвращаемся в *нормальный* режим снова, вызывая метод *update()*.

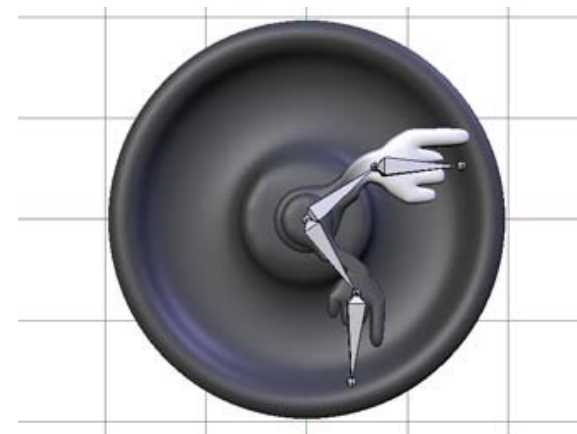
Вы можете обратить внимание, что при создании нашей арматуры мы создаём экземпляры объектов *Editbone*. Вне режима редактирования эти те же кости ссылаются на объекты типа *Bone*. Оба ссылаются на одну и ту же кость, но предлагают различную функциональность и атрибуты, подходящие для *режима редактирования* или для *режима объектов*. Для того, чтобы снабдить нас тем же подходом, Блендер также предоставляет объекты *PoseBone* для манипуляции костями в **режиме Позы**.

Кости позиционируются в арматуре определением позиций их головы и хвоста (тупой и острый концы соответственно, в представлении кости как восьмиугольника). Для соединения костей не достаточно сделать позицию хвоста одной кости равной позиции головы другой. Для того, чтобы кость следовала за перемещениями

другой кости, она должна быть дочерней к ней. Отношения родитель-потомок осуществляются установкой в атрибут *parent* потомка ссылки на объект родительской кости. В нашем примере, у нас каждая кость ладони является потомком своей соответствующей кости руки.

Кости в составе арматуры индексируются их именем. Если свойство модификатора арматуры *VGROUPS* установлено, имя кости должно быть идентично имени группы вершин, на которую она влияет.

Последняя строка кода нашего примера также важна; необходимо сделать арматуру родителем Меш-объекта. Это может показаться излишним в ситуациях, где арматура и меш остаются в одном и том же месте, и перемешаются только отдельные кости в арматуре; но если не сделать этого, это приведёт к неустойчивому отображению меша при интерактивном изменении позы (Вы должны переводить меш в режим редактирования и обратно, например, чтобы видеть эффект от позы на арматуре, который полностью непригоден для работы). Результат нашей оснастки будет выглядеть похожим на это (мы установили режим отображения арматуры в *x-ray*, чтобы сделать её видимой через меш):



Отрендеренный результат выглядит так:



Мы могли бы захотеть ограничить движение отдельных костей до точных вращений вокруг оси z, и это можно сделать добавлением ограничений (*constraints*). Мы столкнемся с ограничениями в следующем разделе.

## Get a bit of backbone boy!

Всё, что мы узнали уже об остнастке, может быть применено также к *creepycrawlies.py*. Если мы хотим расширить функциональность сгенерированной модели, мы можем соединить модификатор арматуры со сгенерированным мешем. Мы также создадим объект арматуры с подходящим набором костей.

Наша задача уже облегчена, потому что мы уже сгруппировали вершины частей тела в модуле *mymesh*, так что связывание их с группой вершин и соответствующей костью тривиально. Не таким тривиальным будет создание самих костей, так как их может быть много, и их нужно разместить и соединить правильным способом.

Давайте посмотрим на то, как могли бы быть осуществлены некоторые существенные элементы (полный код смотрите в *creepycrawlies.py*). Сначала мы должны создать арматуру и сделать её редактируемой для добавления костей:

```
ar = Blender.Armature.New('BugBones')
ar.autoIK = True
obbones = scn.objects.new(ar)
```

```
ar.makeEditable()
```

Мы можем также задать любые атрибуты, которые изменяют поведение арматуры или способ её отображения. Здесь мы просто включаем свойство *autoIK*, так как это сделает манипуляции хвостом нашего создания, возможно очень длинным, намного проще для аниматора.

Следующий шаг - это создание костей для каждого набора вершин. Список *vgroup* в следующем коде содержит кортежи (*vg,vlist,parent,connected*), где *vg* - имя группы вершин а *vlist* - список индексов вершин, принадлежащих этой группе. Каждая кость, которую мы создаем, может иметь родителя и может физически быть соединена с родителем. Эти условия задаются частями кортежа *parent* (родитель) и *connected* (соединён):

```
for vg,vlist,parent,connected in vgroup:
```

```
    bone = Blender.Armature.Editbone()
    bb = bounding_box([verts[i] for i in vlist])
```

Для каждой кости, которую мы создаем, мы вычисляем габаритный ящик (bounding box) всех вершин в группе, на которые эта кость будет влиять. Дальше мы должны разместить кость. При способе, которым мы настраивали наше создание, все сегменты его тела вытягивались вдоль оси y, за исключением крыльев (*wing*) и ног (*leg*). Они вытягивались вдоль оси x. Мы сначала проверяем этот факт, и соответственно устанавливаем переменную *axis* (ось):

```
    axis=1
    if vg.startswith('wing') or vg.startswith('leg'):
        axis = 0
```

Кости в составе арматуры индексируются по имени и позиции концов костей, сохраненных в их атрибутах *head* (голова) и *tail* (конец) соответственно. Так, если у нас есть родительская кость, и мы хотим определить её среднее значение координаты y, мы можем вычислить это следующим способом:

```
    if parent != None :
        parenty = (ar.bones[parent].head[1] +
                   ar.bones[parent].tail[1])/2.0
```

Мы вычисляем эту позицию, поскольку такие части как, например, ноги и крылья имеют родительские кости (то есть, они



перемещаются вместе с родительской костью), но не подсоединены головой к хвосту. Мы разместим эти кости, начиная в центре родительской кости, и для этого нам нужна позиция родителя по у. Кости сегментов, лежащих вдоль оси у, сами позиционируются вдоль оси у, и, таким образом, имеют нулевые координаты х и z. Координаты х и z ног и сегментов крыльев берутся из их габаритных ящиков. Если кость подсоединена (*connected*), мы просто устанавливаем позицию её головы в копию позиции хвоста родителя (выделено ниже).



Класс Блендера *Vector* предоставляет функцию *copy()*, но как ни странно, нет функции *\_\_copy\_\_()*, так что он не будет играть по правилам с функциями из модуля Питона *copy*.

```
if connected:
    bone.head = ar.bones[parent].tail.copy()
else:
    if axis==1:
        bone.head=Blender.Mathutils.Vector(0,
                                              bb[1][0],0)
    else:
        bone.head=Blender.Mathutils.Vector(bb[0][1],
                                              parenty,bb[2][1])
```

Положение хвоста кости рассчитывается аналогичным образом:

```
if axis==1:
    bone.tail=Blender.Mathutils.Vector(0,bb[1][1],0)
else:
    bone.tail=Blender.Mathutils.Vector(bb[0][0],
                                        parenty, bb[2][0])
```

Последние шаги в создании кости - это добавление её к арматуре и установка специфичных для костей опций и всех родительских связей.

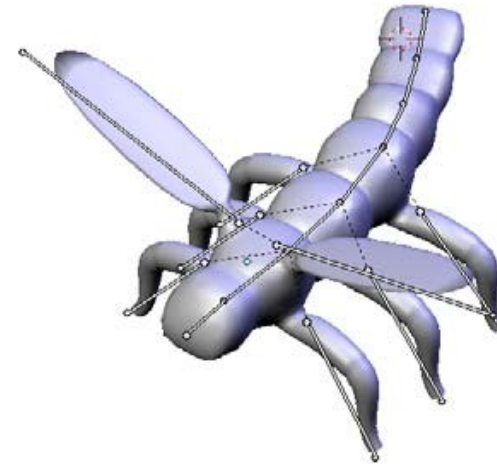
```
ar.bones[vg] = bone

if parent != None :
    bone.parent=ar.bones[parent]
else:
    bone.clearParent()
```

```
if connected:
    bone.options=Blender.Armature.CONNECTED
```

Заметьте, что в предыдущем коде важен порядок действий: атрибут *parent* может быть присвоен или очищен только у костей, которые добавлены к арматуре, а опция *CONNECTED* может быть установлена только у кости, имеющей родителя.

Кроме того, мы должны остерегаться здесь некоторой специфичности Блендера. Родитель может быть установлен у кости назначением в её атрибут *parent*. Если у него нет родителя, этот атрибут возвращает *None*. Тем не менее, мы не можем *назначить None* в этот атрибут, мы должны использовать функцию *clearParent()*, чтобы удалить родительские отношения.



## Материалы

Материалы — это то, что дает объекту внешнее проявление. В Блендере, материалы чрезвычайно разносторонни, и из-за этого довольно сложны. Почти любым аспектом того, как ведёт себя луч света при отражении от объекта, можно управлять, и не только простыми параметрами, но также картами изображений и нодовыми сетями.

Вплоть до 16 материалов может быть связано с объектом и, в пределах объекта, индивидуальные его части могут ссылаться на один из этих 16 материалов. На объектах *Text3d*, каждый индивидуальный символ может ссылаться на различный материал, и для кривых это так же для каждой управляющей точки.

С точки зрения разработчика, назначение материала объектам - процесс из двух шагов. Сначала, мы должны определить новый материал, и затем мы должны назначить материал или материалы на объект. Первый шаг может быть опущен, если мы можем сослаться на уже существующие материалы.

Если у объекта, подобному мешу, уже определены грани, тогда мы все еще должны назначать материал для каждой грани. Вновь создаваемые грани будут иметь назначенным активный материал, если активный материал задан.

Небольшой кусок кода иллюстрирует, как мы можем назначать материалы на Меш-объект. Здесь мы назначаем материал с белым рассеянным цветом для всех граней с чётным номером, и с черным рассеянным цветом для всех граней с нечетным номером на Меш-объекте в переменной *ob*.

```
me=ob.getData(mesh=1)
```

```
mats=[ Blender.Material.New(), Blender.Material.New() ]
mats[0].rgbCol=[1.0,1.0,1.0]
mats[1].rgbCol=[0.0,0.0,0.0]
```

```
ob.setMaterials(mats)
ob.colbits=3
```

```
for f in me.faces:
    if f.index%2 == 0 :
        f.mat=0
    else:
        f.mat=1
```

Выделенная строка гарантирует, что индексы материалов, используемые для каждой грани, относятся к материалам, назначенным на объект. (Также возможно связать материалы с меш-данными, как мы увидим в следующей секции.)

## Материалы Объекта против материалов ObData

В Блендере и Меш-объект и объект Блендера верхнего уровня, содержащий Меш-объект, могут иметь свой собственный список из 16 материалов. Это удобно, если нам нужно множество экземпляров копии одного и того же меша, но с приложенными различными материалами. Тем не менее, в некоторых ситуациях, мы можем захотеть приложить некоторые или все материалы к Мешу, а не к объекту. Это управляется атрибутом объекта *colbits*. Этот атрибут состоит из 16 битов, и каждый из них указывает использовать материал от Объекта или от Меша. Мы уже видели пример с этим атрибутом в предыдущей секции.

Объект Кривой (*Curve*) также может иметь собственный набор материалов, и выбор фактического материала подчиняется тем же правилам, что и для Меш-объекта. Метаболлы также имеют свой собственный набор материалов, и переключение между комплектами материалов производится так же, но в отличие от многих типов объектов, которые состоят из частей (смотри следующую секцию), нет способа соединять различные материалы с различными элементами в пределах Метаболла (это истинно также и в графическом интерфейсе пользователя: кнопки на панели Links and Materials контекста Редактирования существуют, чтобы назначать индексы материалов индивидуальным элементам метаболла, но они не дают эффекта). Используется только первый слот списка материалов.

Заметьте, что объекты, которые не визуализируются сами, как например, арматуры и решетки, не имеют связанных материалов (таким образом, любые материалы, связанные с Объектом верхнего уровня, содержащим арматуру или решетку, будут проигнорированы). Некоторые объекты, которые не имеют связанных материалов, могут иметь связанные с ними текстуры. Объекты Мира и Лампы, например, можно связать с текстурами, чтобы управлять их цветами.

## Назначение материалов частям Объекта

В пределах меша каждая грань может иметь собственный, связанный с ней материал. Этот материал идентифицируется своим индексом в списке материалов и сохраняется в атрибуте *mat*. В



пределах объекта *Text3d*, каждый символ может иметь собственный материал, опять же идентифицируемый своим индексом в списке материалов. На этот раз, этот индекс не хранится непосредственно в атрибуте, но может быть установлен или извлечен методами *accessor*, которые принимают индекс символа в тексте в качестве аргумента.

Секциям внутри Кривой (объекты *CurNurb*), можно назначить индекс материала их методом *setMatIndex()*. Индекс мог быть извлечен из них соответствующим методом *getMatIndex()*. Заметьте, что связь материала с кривыми, которые состоят из единственной линии без настроенной выдавленной ширины или связанного объекта скоса, не будет иметь видимых эффектов, так как эти кривые не рендерятся.

Следующий кусок кода показывает как назначать различные материалы различным символам в пределах объекта *Text3d*. Сам код прост, но как Вы можете заметить, мы определяем список из трех материалов, но используем только один. Это расточительно, но необходимо, чтобы обойти специфику в функции *setMaterial()*. Её аргумент индекса материала должен быть смещён на один, например, индекс 2 имеет отношение ко второму материалу в списке, тем не менее, самый большой индекс может пройти не смещённым на единицу. Так если мы хотели бы использовать два материала, мы должны бы использовать индексы 1 и 2, чтобы иметь доступ к материалам 0 и 1, но фактический список материалов должен содержать три материала, в противном случае мы не сможем передать 2 в качестве аргумента в *setMaterial()*.

```
mats=[Material.New(),Material.New(),Material.New()]
mats[0].rgbCol=[1.0,1.0,1.0]
mats[1].rgbCol=[0.0,0.0,0.0]
mats[2].rgbCol=[1.0,0.0,0.0]
ob.setMaterials(mats)
ob.colbits=3
txt=ob.getData()
```

```
for i in range(len(txt.getText())):
    txt.setMaterial(i,1+i%2)
```

Выделенный код показывает коррекцию на 1. Полный код представлен как *TextColors.py*.

## Вершинные цвета против материалов граней

Один важный аспект работы с материалами, с которым мы пока не имели дела - **цвета вершин**. В мешах каждая вершина может иметь собственный цвет вершины. Цвет вершины отличается от материала, но будут ли цвета вершин вызывать какие-то видимые эффекты, контролируется флагами режима материала. Чтобы использовать любые цвета вершины в материале, должен быть установлен бит *VColPaint* вызовом метода *setMode()*. Когда используется этот режим, цвета вершин определяют диффузный (рассеянный) цвет материала, тогда как все обычные атрибуты материалов управляют способом, которым этот диффузный цвет будет рендериться. Обычное использование для цветов вершин - это *запекание* дорогих в вычислительном отношении эффектов, как например, *ambient occlusion*. Поскольку цвета вершин можно рендерить очень быстро, *ambient occlusion* может быть аппроксимирована этим способом, даже в настройке реального времени, как например, в игровом движке. (Аппроксимирована, поскольку при этом не будет такой же реакции на изменения в освещении.)

Цвета вершин сохраняются как объекты *Mesh.MCol* (основаны на кортежах RGBA) в атрибуте грани *col*. Атрибут *col* является списком, содержащим ссылку на объект *MCol* для каждой вершины в грани. У такого размещения проявится смысл, когда Вы поймёте, что фактически материалы связаны с гранями, а не с вершинами. Когда цвета вершин различны, они линейно интерполируются через грань.

Присваивать атрибуту грани *col* возможно, если только у меша был установлен его атрибут *vertexColors* в Истину.

Следующий пример показывает, как мы можем установить цвета вершин меша. Мы выбираем градации серого в зависимости от координаты z вершин (выделено).

```
import Blender
ob=Blender.Scene.getCurrent().objects.active
me=ob.getData(mesh=1)

me.vertexColors=True
for f in me.faces:
    for i,v in enumerate(f.verts):
```

```
g = int(max(0.0,min(1.0,v.co.z))*255)
f.col[i].r=g
f.col[i].g=g
f.col[i].b=g
```

```
mats=[Blender.Material.New()]
mats[0].setMode(Blender.Material.Modes['VCOL_PAINT'])
ob.setMaterials(mats)
ob.colbits=1
ob.makeDisplayList()
```

```
Blender.Window.RedrawAll()
```

Полный код доступен как *VertexColors.py*.

## Добавление материалов в нашу гравюру

Как последний штрих в нашей деятельности с гравюрой, мы добавим два материала. Один индекс материала назначим вершинам на поверхности, и другой вершинам в выточенных канавках. Этим методом мы можем, например, создать проявление вновь созданной надписи на куске выветренного камня.

Так как мы ранее определили несколько удобных групп вершин, назначение индексов материала будет вопросом итерации над всеми гранями и назначения в каждую вершину грани подходящего индекса материала в зависимости от того, членом какой вершинной группы является вершина. Функция, показанная ниже, принимает чуть более общий подход, так как она принимает меш и список регулярных выражений, и назначает индекс материала на каждую грань в зависимости от принадлежности к группе вершин, которая имеет имя, соответствующее одному из регулярных выражений.

Эти функции делают очень легким назначение одинакового индекса материала во все группы вершин, которые имеют аналогичные имена, например все хвосты и сегменты грудной клетки меша, создаваемого *creepycrawlies.py* (они все имеют такие имена как, например, *tail.0*, *tail.1*, , и так далее).

Функция доступна в *Tools.py*. Она зависит от функции Питона *re.search()*, которая сопоставляет регулярное выражение со строкой. Выделенная строка показывает, что мы вставляем строку

регулярного выражения в так называемые якоря (^ и \$). Этим путём регулярное выражение, такое как например, *aaaa*, сопоставится только с группой вершин с именем *aaaa*, а не с именем *aaaa.0*, так что мы сможем различить их, если мы хотим. Если же мы хотим соответствия всем именам групп вершин, которые начинаются с *tail*, мы могли бы, например, передать регулярное выражение *tail.\**.



Регулярные выражения являются чрезвычайно мощным способом сопоставления строк. Если Вы незнакомы с ними, Вы должны обратиться к документации по модулю Питона *re* (<http://docs.python.org/library/re.html>). Можно начать, например, с <http://wiki.intuit.ru>.

Другая вещь, которую нужно отметить в этой функции — использование операций с множествами. Они немного ускорят процесс, так как операции с множествами в Питоне чрезвычайно быстрые. Мы используем их здесь, чтобы проверять множество вершин (или, скорее, их индексов), которые составляют грань, на то, что все они входят в множество индексов вершин, находящихся в некоторой группе вершин. Мы заранее вычисляем оба множества индексов вершин, те, которые принадлежат группе вершин и индексы вершин каждой грани, и храним их в словарях для легкого доступа. Таким образом, мы создаем эти множества только однажды, для каждой группы вершин и для каждой грани соответственно, вместо воссоздания каждого множества всякий раз, когда мы сопоставляем регулярное выражение. Для больших мешей это потенциально сохранит много времени (за счет памяти).

```
import re
```

```
def matindex2vertgroups(me,matgroups):
    if len(matgroups)>16:
        raise ArgumentError("number of groups larger than
                               number of materials possible (16)")

    groupnames = me.getVertGroupNames()

    vertexgroupset={}
    for name in groupnames:
        vertexgroupset[name]=set(me.getVertsFromGroup(name))
        print name,len(vertexgroupset[name])
```

Далее мы выйдем за пределы статики, и увидим как управлять перемещением объектов.

```
faceset={}
for f in me.faces:
    faceset[f.index]=set([v.index for v in f.verts])

for i,matgroup in enumerate(matgroups):
    for name in groupnames:
        if re.search('^'+matgroup+'$',name):
            for f,vset in faceset.items():
                if vset.issubset(vertexgroupset[name]) :
                    me.faces[f].mat = i
            break
```



## Итог

В этой главе, мы видели как сделать жизнь легче для наших конечных пользователей, определяя вершинные группы у мешей, чтобы упростить выбор определенных характеристик. Мы также видели, как назначать материалы вершинам, и как создавать новые материалы, если нужно. Первые шаги были предприняты, чтобы оснастить (rig) меш. В частности, мы узнали:

- Как определять вершинные группы
- Как назначать вершины в вершинные группы
- Как назначать материалы граням
- Как назначать вершинам вершинные цвета
- Как устанавливать свойства рёбер
- Как добавлять модификатор
- Как покрывать кожей кости

## PyDrivers и Constraints (Управляющие объекты и Ограничения)

Перевод: S.Lavik, Striver

Проектируя сложные объекты с подвижными частями, мы хотели бы управлять перемещением этих частей относительно друг друга. Иногда, для достижения цели мы можем использовать физические движки, например, такие как Bullet physics, но часто этого не достаточно для точного воспроизведения анимации, физический движок не всегда предоставляет необходимый контроль над сценой. Большую часть времени умное использование многократных **ограничений** будет вполне достаточным, но иногда взаимосвязи между объектами (другими словами хорошая анимация) не могут быть выражены с точки зрения простых **ограничений** и ключевой анимации. В таких случаях мы можем расширить возможности Блендера, определив собственные **ограничения** или **отношения** между анимируемыми объектами, используя Питон.

В этой главе мы увидим, как можно связать встроенные **ограничения** с объектами Блендера и как определить сложные **отношения** между анимированными объектами используя так называемые **pydrivers**. Мы также определим новые сложные **ограничения**, которые могут использоваться точно так же, как и встроенные ограничения. Мы пока не будем изучать такие определения, как ключевые кадры (key frames), поскольку мы столкнемся с ними в более поздних главах.

В этой главе мы узнаем:

- Как управлять одним **IPO** из другого в выражениях Питона
- Как работать с некоторыми ограничениями, присущими **pydrivers**
- Как управлять движением объектов и костей, добавляя **ограничения**
- Как написать **ограничение** в Питоне, которое привяжет один объект к ближайшей к нему вершине на другом объекте

Для начала давайте познакомимся с некоторыми определениями, чтобы получить ясное представление о том, с чем мы имеем дело.

### *Акцентируем внимание на свойствах анимации*

Блендер универсален, но достаточно сложен. Прежде, чем мы сможем манипулировать анимацией объектов с помощью Питона, необходимо, чтобы мы разобрались с основными понятиями.

### **IPO**

В Блендере почти любой объект может быть анимирован. Обычно это делается, с помощью фиксации некоторых параметров, таких как положение в пространстве некоего объекта в определенных ключевых кадрах и интерполяция этих параметров для остальных промежуточных кадров. В Блендере группы объектов, задействованные в анимации собираются в так называемые кривые IPO. Например, все пространственные параметры, такие как местоположение, вращение, и масштаб сгруппированы как тип объекта IPO и могут быть связаны со многими объектами Блендера: мешем, камерой, или лампой. Большинство свойств материалов в Блендере также могут быть сгруппированы в соответствующем IPO. Получается, что "Материальный" тип IPO может быть связан с любым объектом, которому присвоен материал. Аналогично, тип IPO Лампы должен быть связан с объектом Лампы.



**IPO** это аббревиатура, но что она обозначает кажется немного неясным. Wiki Блендера заявляет, что она происходит от слова **InterPOlation**, то есть от математической функции (*почитайте Wiki там есть интересная информация на этот счет - прим. пер*), но в Блендере мы столкнемся с интерполяцией как с объектом. И большую часть времени будем использовать IPO как существительное, однако, это обсуждение становится немного академическим.

Каждый IPO может быть связан с более чем одним объектом. Например, возможно анимировать вращение нескольких объектов, объединив их с одним объектом IPO. В Блендер API кривые IPO представлены объектами IPO. Объект IPO может быть связан с другим объектом посредством метода `setIpo()`. Следующая таблица дает краткий обзор типов IPO, IPO-каналов, и список объектов с которыми они могут взаимодействовать. Обратитесь к API документации о модуле `Blender.IPO` за подробной информацией.

(<http://www.blender.org/documentation/249PythonDoc/index.html>).

Тип IPO	IPO каналы (некоторые примеры, см. полный список в API документации)	Соответствующие объекты в Блендере
<i>Object</i>	LocX, LocY, LocZ (перемещение) RotX, RotY, RotZ (вращение) ScaleX, ScaleY, ScaleZ (масштаб)	Все объекты Блендера, которые можно перемещать: Меш, Лампа, Камера и др.
<i>Pose</i>	RotX, RotY, RotZ (вращение)	Кости (Bone)

<i>Material</i>	R,G,B (рассеянный цвет)	Любые объекты, использующие материалы
<i>Texture</i>	Contrast (контрастность)	Любые объекты, использующие текстуры, например: Меш, Лампа, Мир и др.
<i>Curve</i>	Speed (скорость)	Кривые (Curve)
<i>Lamp</i>	Energ (энергия) R,G,B (цвет)	Лампы
<i>World</i>	HorR,HorG,HorB (цвет горизонта)	Мир (World)
<i>Constraint</i>	Inf (влияние)	Ограничения
<i>Sequence</i>	Fac (фактор, например громкость звуковой дорожки) Обратитесь к API документации для Blender.IPO module за подробной информацией	Последовательности

## IPO-каналы и IPO-кривые

Кривые IPO, перечисленные в таблице, содержат целую коллекцию связанных между собой анимационных параметров. Каждый из этих параметров упоминается как канал. Примером **канала** IPO-объекта является `LocX` (x-компонент местоположения) и `RotY` (вращение вокруг оси Y). Каждый **канал** представлен объектом `IPOCurve`, который реализует необходимую функциональность для

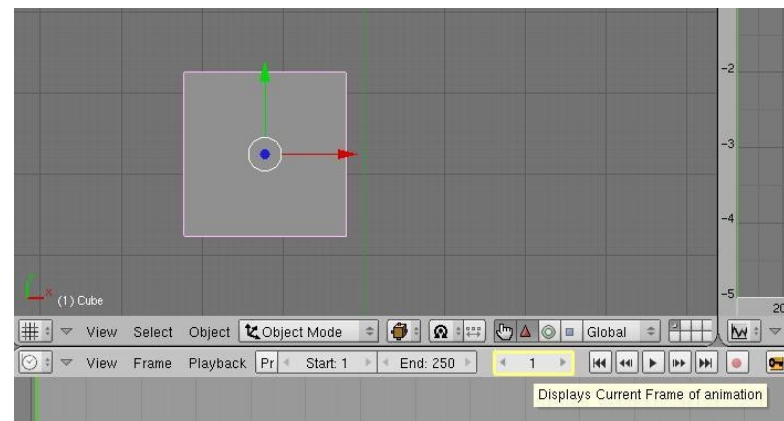
возвращения значений, интерполированных между ключевыми кадрами анимации.

Примером канала в IPO материала (Material) является SpecB – синий компонент зеркального цвета (specular color).

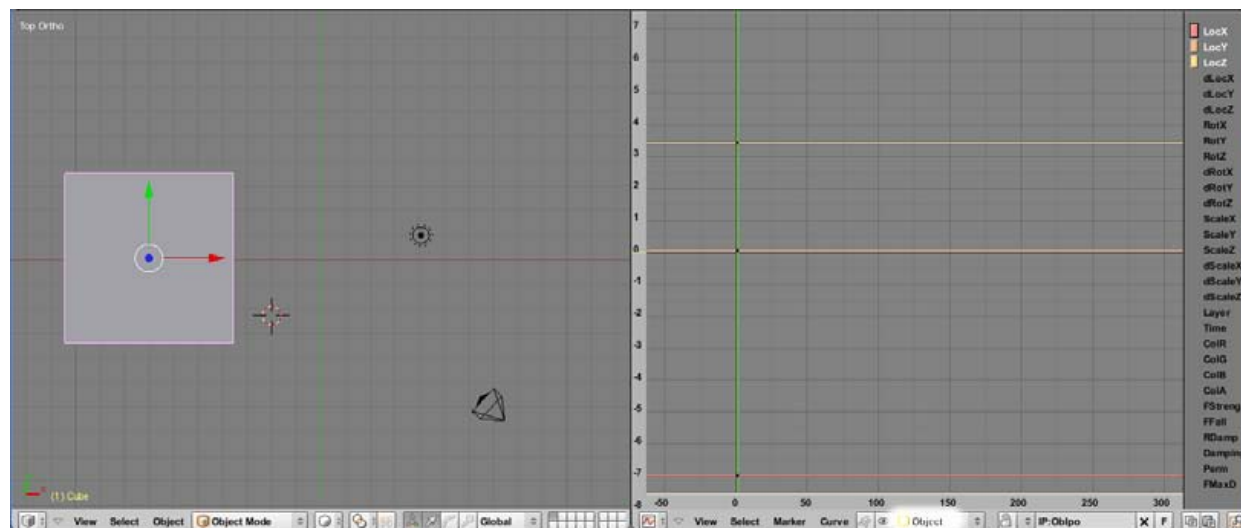
IPOCurve-объекты доступны как атрибуты приведенных в таблице IPO, например, `myipo.LocX` обратится к `LocX` IPOCurve, если `myipo` будет IPO - объектом.

Чтобы проиллюстрировать эти понятия предположим, что мы хотим анимировать движение простого куба вдоль оси X. Мы начнем движение с 1 кадра и закончим его в кадре номер 25. В Блендере выполним следующие шаги:

1. Добавьте простой Куб, выбрав в меню **Add | Mesh | Cube** и удостоверьтесь, что Вы находитесь в объектном режиме (object mode).
2. Перейдите к первому кадру анимации (чтобы выбрать необходимый кадр, просто введите число в виджет, показанный на скриншоте).



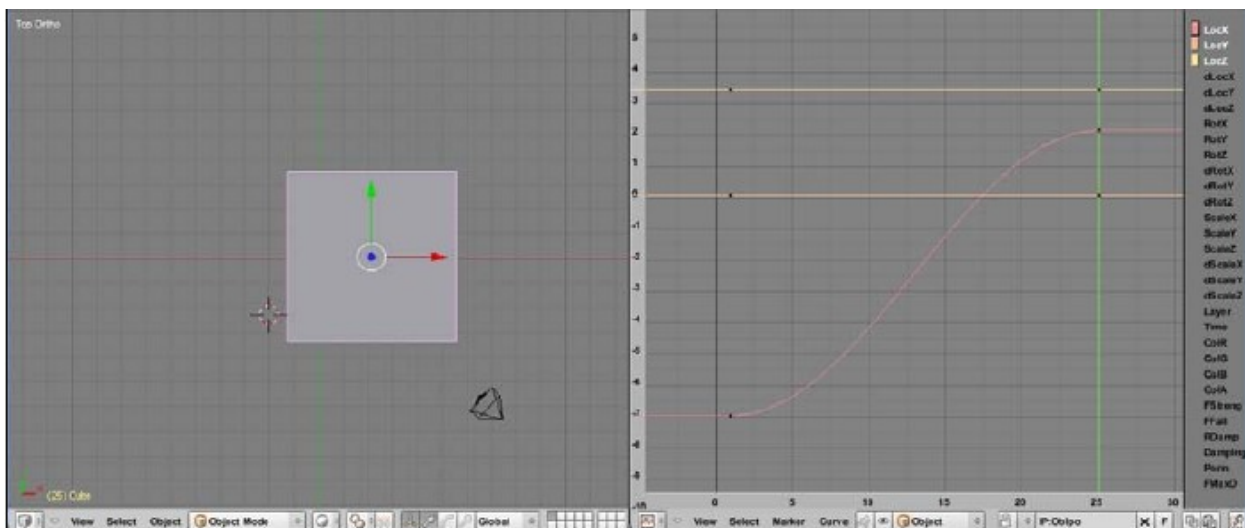
3. Добавьте ключевой кадр, выбрав **Object | Insert keyframe | Loc**. В окне редактора IPO добавленный ключевой кадр расположения нашего куба в пространстве обнаружится как IPO типа *Object* (см. скриншот).





Текущий кадр отображается в виде зеленой вертикальной линии. Расположение IPO зафиксировано тремя каналами (для положения куба вдоль оси X - LocX, вдоль осей Y и Z - LocY и LocZ соответственно). Каналы представлены в виде графиков различных цветов (они могут перекрывать друг на друга). Этими линиями можно управлять прямо в редакторе IPO Кривых, но пока мы только добавим второй ключевой кадр.

1. В окне Timeline выберите 25 кадр.
2. Выберите Куб и переместите его вправо вдоль оси X.
3. Добавьте второй ключевой кадр, выбрав **Object | Insert keyframe | Loc** (или просто нажав *I* - прим. пер.). Теперь мы видим что каждый из графов, представляющих три IPO-канала (направления по осям X, Y, Z) получили вторые точки-пересечения с зеленой линией. Поскольку мы изменили местоположение куба только вдоль оси X, графы других каналов остались плоскими, но линия канала LocX изменилась вслед с изменением положения куба по оси X.



Добавляя больше ключевых кадров, мы можем сделать любое движение настолько сложным, насколько нужно, но задача становится более тяжелой, если мы например захотим заставить наш объект следовать по предварительно вычисленному точному

пути. Позже в этой главе мы увидим, как можно управлять объектами IPOCurve, которые представляют IPO-каналы с помощью программирования.

## Ограничения

**Ограничения** в Блендере связаны с объектами Блендера верхнего уровня или Bone-объектами и представлены в виде объекта Constraint. У Объектов Блендера и Bone-объектов есть атрибут `constraint`, с помощью которого осуществляется **последовательность ограничений**. Также выше перечисленные объекты имеют методы, для добавления, удаления, и изменения **ограничений** в этой последовательности.

Когда **ограничение** связано с объектом, результатом будет объединение параметров ограничений и расчетных параметров объекта. Атрибут `influence` (влияние) определяет, насколько сильно параметры **ограничения** будут влиять на объект анимации.

## Различия между управляющими объектами (drivers) и ограничениями

Управляющие объекты и ограничения похожи тем, что они влияют на изменение свойств пути (*речь идет о параметрах анимации — прим. пер.*), но в тоже время они очень разные: **ограничения** действуют непосредственно на объекты, в то время как **управляющие объекты** определяют то, как IPO-кривая будет изменяться относительно изменений других IPO-кривых в процессе анимации. **Ограничения** влияют только на пространственные свойства объекта, такие как положение, масштаб или вращение, а с помощью **управляющих объектов** любой кривой IPO можно управлять с помощью другой кривой IPO. Это означает, что даже параметры материалов, такие как цвет, или параметр лампы, такой как энергия, может управляться другим IPO. Однако есть ограничение: IPO-кривые, управляющие другими IPO-кривыми должны в настоящее время обладать специальными свойствами объекта, таким образом, Вы можете управлять цветом материала,



вращая некоторый объект, но Вы не можете изменить цвет объекта энергией лампы. Кроме того, факт, что **ограничения** могут затронуть только пространственные свойства, означают, что нет никакого способа, которым Вы можете ограничить, например, рассеянный цвет (diffuse color) материала. Следующая таблица показывает некоторые **ограничения** и их соответствующие атрибуты. Обратитесь к документации API по модулю `Blender.Constraint` за подробной информацией.

Типы Ограничений	Стандартные атрибуты
TrackTo	Target (target object) Track (axis to track)
Floor	Target (target object)
StretchTo	Target (target object)
CopyLocation	Copy (выбор компонента(тов) для копирования )

Заметьте, что возможно анимировать влияние ограничения (параметр `influence`), когда с Объектом связано IPO типа `constraint`.

### Программирование ограничений

Блендер имеет много **ограничений**, которые Вы можете применить к объекту. Некоторые из них похожи на **управляющие объекты** (`drivers`), в том смысле, что они не ограничивают движение объекта, но могут копировать некоторые параметры, такие как вращение или расположение (`location`). С точки зрения разработчика, каждому объекту Блендера присущ атрибут `constraints`, который является последовательностью объектов ограничений. В эту последовательность можно добавлять элементы и удалять их из неё. Также можно менять порядок элементов.

Метод	Действие	Пример
<code>append(<i>type</i>)</code>	Добавляет новое ограничение к объекту и возвращает ограничение	<code>ob.constraints.append(Constraint.Type.TRACKTO)</code>
<code>remove(<i>constraint</i>)</code>	Удаляет ограничение с объекта	<code>ob.constraints.remove(ob.constraints[0])</code>
<code>moveUp(<i>constraint</i>)</code> <code>moveDown(<i>constraint</i>)</code>	Изменяет позицию ограничения в списке ограничений	<code>ob.constraints.moveDown(ob.constraints[0])</code>
<code>[]</code>	Доступ к атрибутам ограничений	<code>Con = ob.constraints[0]</code> <code>Con[Constraint.Settings.TARGET] = other</code>

Новые **Ограничения** не становятся экземплярами объектов посредством конструктора, но посредством вызова метода `append()` атрибута `constraints` вместе с переданным ему типом ограничения. на выходе `append()` мы получаем новое **Ограничение**, параметры настроек которого уже можно изменять.

### Программирование кривых IPO

IPO-каналы управляются из скриптов так же, как и **ограничения**, но они по своей сути более разнообразны, чем **ограничения**, поскольку существует много различных типов IPO-каналов, и некоторые из них, особенно текстурные каналы и ключи формы, нуждаются в специальной обработке. Про них существует отдельная глава (Глава 6: Ключи формы, IPO, и Позы), но различные варианты использования Питона для IPO - каналов будут показаны ниже.

### Управляющие объекты (PyDrivers)

Есть много случаев, где мы хотели бы изменять некоторые свойства, относительно других свойств анимируемых объектов, но

не всегда возможно зафиксировать эти «взаимоотношения», управляя одним IPO-каналом через другой. Так происходит потому, что такое отношение не всегда оказывается простой линейной зависимостью, например, движение поршня управляется круговым движением. Другой случай когда отношение не постоянно, например, свет, включающийся только тогда, когда выключатель находится в определенном положении.

В этих случаях отношения между объектами могут быть определены Питон-выражением или так называемым **pydriver**. **Управляющий объект** принимает IPO-канал другого объекта как входной параметр и возвращает результат, управляющий IPO-каналом на текущем объекте. Поскольку эти выражения на Питоне имеют доступ к полному API Блендера, взаимоотношения могут быть действительно очень сложными.

## Ограничения на Питоне (PyConstraints)

Там, где **управляющие объекты** могут использоваться, чтобы обходить пределы встроенных возможностей Блендера по управлению IPO-каналами, **PyConstraints** позволяют нам преодолеть трудности в ситуациях, где встроенные **ограничения** не достаточны. Например, невозможно ограничить положение одного объекта на поверхности другого, если в нем есть отверстия. Встроенные **ограничения** предлагают способы ограничивать расположение объекта не ниже чем расположен другой объект (ограничение `floor`). Но если мы хотели бы, чтобы была возможность изменять позицию объекта ниже поверхности другого объекта в местах, где есть отверстия, мы должны запрограммировать такое **ограничение** самостоятельно. Как мы увидим, PyConstraints позволяют нам сделать точно это.

Поскольку все вступительные замечания позади, мы наконец снова можем вернуться к программированию в следующем абзаце.

## Установка времени - один управляет всеми

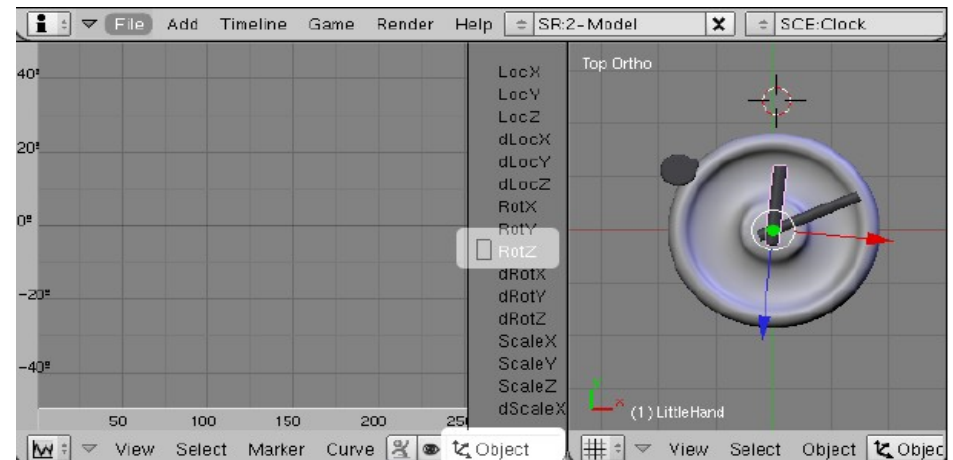
Как использовать часы, если невозможно установить время удобным способом? Вместо перемещения каждой стрелки часов

отдельно, мы хотели бы поворачивать единственную кнопку, чтобы перемещать обе стрелки — большую (минутную) и маленькую (часовую), причем (очевидно) часовая стрелка должна перемещаться в двенадцать раз медленнее минутной.

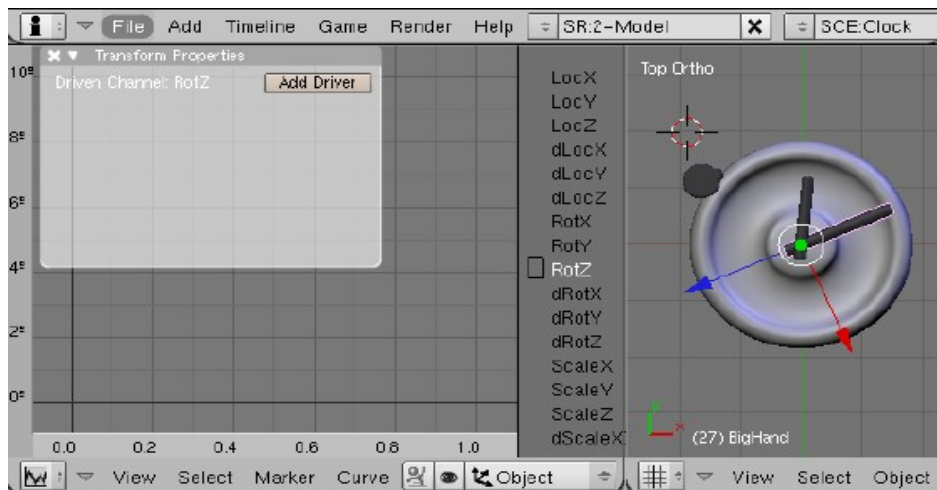
Поэтому, мы должны определить объект-кнопку (которую мы скорее всего не будем визуализировать), для управления вращением костей в стрелках часов.

Чтобы настроить ведомые каналы, выполним шаги:

1. В окне 3D View, выберите объект `bighand` (большая стрелка).
2. В окне редактора IPO-Кривых удостоверьтесь что выбран тип IPO – object. Справа вы увидите список IPO-каналов. Выберите `RotZ`, щелкнув на нем левой кнопкой мышки.



3. Выберите **Curve | Transform Properties**. В появившемся окне нажмите на кнопку **Add Driver**.



4. Не закрывая Transform Properties, выберите **Curve | Insert 1:1 mapping** и затем щелкните по **Default one-to-one mapping**. В редакторе IPO появится прямая светло-голубая линия.



5. В окне Transform Properties, нажмите на изображение светло-зеленого питона. Изображение станет темно-зеленым, и теперь

возможно редактировать выражение **pydriver** в смежной текстовой области. Введите туда следующий код:

```
ob('Knob').RotZ*(360/(2*m.pi))/10
```

Вот и все! Теперь, если вращать кнопку вокруг оси Z, большая стрелка следует примеру. Все же **pydriver**-выражение действительно нуждается в некотором разъяснении. Выделенная часть является движком (driver) - канал объекта (object channel), обеспечивающий входные данные для управления IPO-каналом. `ob('Knob')` является укороченной записью (стенографией), позволенной в **pydriver**-выражениях для Блендера. `Object.Get('Knob')` и атрибут `RotZ` дают нам вращение вокруг оси Z. Это вращение задано в радианах, тогда как результат **pydriver**-выражения для канала вращения (`RotZ`) должен быть в градусах, поэтому мы умножаем на 360 и делим на удвоенное число пи ( $m.pi = 3.14$ ). Наконец, мы делим полученное число в градусах на десять, потому что по некоторой неясной причине, Блендер не принимает градусы, не поделенные на 10! (Заметьте, что это "делить на десять" действительно нужно только для каналов вращения по осям, но не для любого из других каналов!)

### 1-on-1 mappings



Вы можете задаться вопросом, почему мы должны были сначала вставить кривую 1:1. Отношение между ведомым каналом и его **управляющим объектом** содержит еще один слой и это - кривая, транслирующая значение на выходе **управляющего объекта** (pydriver) в финальное значение. Эту кривую можно изменять вручную, но обычно мы делаем всю точную настройку в нашем **pydriver** и просто вставляем кривую 1-к-1. Такой вариант работы настолько распространен, что Блендер обеспечивает специальный интерфейс для этой ситуации, так как весьма утомительно создавать необходимые кривые снова и снова для каждого управляемого канала.

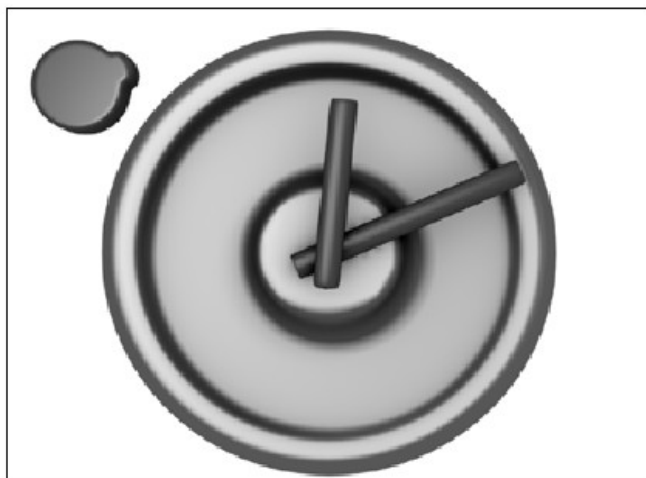
Конечно, мы, возможно, достигли бы того же самого результата, ведя вращение напрямую через канал вращения объекта `knob`, или даже с помощью копии **ограничения** вращения. Это спасло бы нас

от странных проблем преобразования, но цель этого абзаца показать основы.

Часовая стрелка из примера, - вот где использование **pydriver** действительно является правильным решением. (Хотя, изменяя непосредственно IPO-кривую, мы могли бы изменить темп изменения управляющего канала, но это было бы не столь же ясно, как простое выражение, и почти невозможно для более сложных отношений между объектами). Мы повторяем список действий, показанных ранее, но теперь для маленькой (часовой) стрелки и введем следующее **pydriver**-выражение:

```
ob('Knob').RotZ*(360/(2*m.pi))/10/12
```

Поскольку часовая стрелка в двенадцать раз медленней, чем минутная, мы используем то же самое **pydriver**-выражение что и для минутной стрелки, но разделим результат на двенадцать. Теперь, когда мы вращаем объект `knob` (кнопку) по ее оси Z, минутная стрелка будет следовать как и раньше, а часовая соответственно в 12 раз медленнее. Вместо того, чтобы вручную вращать кнопку, также возможно анимировать вращение кнопки, для анимации обеих стрелок часов. Полный результат доступен как `clock-pydriver.blend`, изображение часов с кнопкой, показано на следующем скриншоте:



## Сокращения

В пределах **pydriver**-выражений можно использовать некоторые полезные сокращения, чтобы экономить на печатании. В пошаговом примере мы уже использовали сокращение `ob('<name>')` — это обращение к объектам Блендера по имени, аналогично, возможно получить доступ к Меш-объектам и материалам посредством `me('<name>')` и `ma('<name>')` соответственно. Кроме того, модуль `blender` доступен как `b`, модуль `Blender.Noise` как `n`, и модуль Питона `math` как `m`. Он позволяет выражениям использовать тригонометрические функции, такие как синус, например. Этих возможностей достаточно, чтобы покрыть много проблем, но их все равно не хватит если мы захотим, например, импортировать внешние модули. Есть путь избежать этих трудностей, мы его увидим в следующем абзаце.

### Преодоление ограничений: **pydrivers.py**

Поле ввода для **pydrivers** ограничено 125 символами, и даже при том, что сокращения позволяют получить доступ к модулю Питона `math` и к некоторым из модулей Блендера, с помощью сокращённых выражений, предоставленного места достаточно мало. Кроме того, поскольку **pydrivers** должны быть **выражениями** Питона, весьма трудно отлаживать их (например, потому что Вы не можете вставить функцию `print`) или добавить нечто похожее на функциональность `if/then`. Последний пример до некоторой степени может быть преодолен хитрыми уловками, основанными на том факте, что Истина (`True`) и Ложь (`False`) в Питоне преобразуются в, соответственно, 1 и 0 внутри числового выражения, таким образом утверждение:

```
if a>b:
    c=14
else:
    c=109
эквивалентно:
```

```
c = (a>b)*14 + (a<=b)*109
```

Однако чувствуется неуклюжесть выражения, ведь мы оцениваем условие дважды. К счастью, и проблему пространства и

ограничение единственного выражения можно преодолеть при использовании текстового блока с именем `pydrivers.py`. Если такой текстовый блок присутствует, его содержание доступно в виде модуля с именем `p`. Так, например, если мы определяем функцию `clamp()` (зажим) в `pydrivers.py` таким образом:

```
def clamp(a, low, high):  
    if a < low: a = low  
    if a > high: a = high  
    return a
```

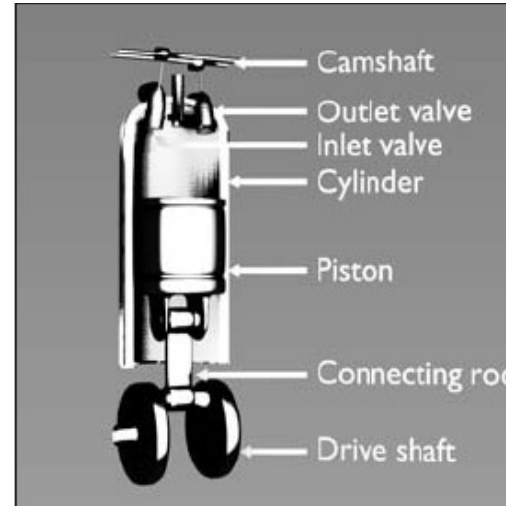
Мы можем вызвать эту функцию в нашем **pydriver**-выражении как `p.clamp(a, 14, 109)`.

Мы будем использовать `pydrivers.py` в следующих примерах, не только потому, что это позволит применять более сложные выражения, но также и потому что ширина **pydriver** области еще меньше чем ее длина, что делает такое выражение очень трудным к прочтению, поскольку Вы должны постоянно пользоваться прокруткой для доступа ко всем частям выражения.

## Внутреннее сгорание — корреляция сложных изменений

Предположим, что мы хотим продемонстрировать, как работает четырехтактный двигатель внутреннего сгорания. У такого двигателя есть множество движущихся частей, и многие из них связаны сложным образом.

Чтобы увидеть отношения между частями двигателя, будет полезно взглянуть на следующую иллюстрацию. На скриншоте перечислены названия, которые мы будем использовать, когда обратимся к различным частям мотора. (Я не автомобильный инженер и не механик, таким образом названия, возможно, не точны, но по крайней мере мы будем говорить об одних вещах. За дополнительной информацией Вы можете обратиться сюда [http://en.wikipedia.org/wiki/Four-stroke\\_cycle](http://en.wikipedia.org/wiki/Four-stroke_cycle).)



*camshaft* – распределительный вал

*outlet valve* – выпускной клапан

*inlet valve* – впускной клапан

*cylinder* – цилиндр

*piston* – поршень

*connecting rod* – шатун

*drive shaft* – ведущий или коленчатый вал

Прежде, чем мы начнем формировать части, чтобы использовать их вращение и положение, для управления другими частями, нужно условиться: в реальности поршни в цилиндрах двигаются за счет расширения воспламененного топлива, они толкают ведущий вал (или коленчатый вал) с соединенным маховым колесом и распределительным валом (или в нашем случае с некоторыми механизмами, которые не показаны здесь), движение возвращается к распределительному валу, который управляет движением выпускных и впускных клапанов. Очевидно, что мы не можем следовать этой концепции непосредственно, поскольку нет никакого топлива как объекта, который стимулирует двигаться другие объекты, таким образом имеет смысл полностью изменить цепь отношений. В нашей установке маховое колесо будет вращать ведущий вал и различные механизмы, а ведущий вал, в свою очередь, будет вести большинство других объектов, включая поршень и его шатун. Мы будем также управлять энергией лампы, помещенной в наконечник свечи зажигания, вращая ведущий вал.

Ведущий вал просто будет следовать за вращением махового колеса, как более медленный механизм (это можно осуществить с помощью **ограничения** *copy rotation* объекта, но здесь мы всё хотим



осуществить через **pydrivers**). Соответствующий **pydrivers** для канала RotX будет похож на это:

```
ob('Flywheel').RotX/(2*m.pi)*36
```

Это может выглядеть неуклюжим, но необходимо помнить - вращения сохраняются в радианах, в то время как **pydriver**-выражения должны возвращать вращение в градусах, деленных на 10.

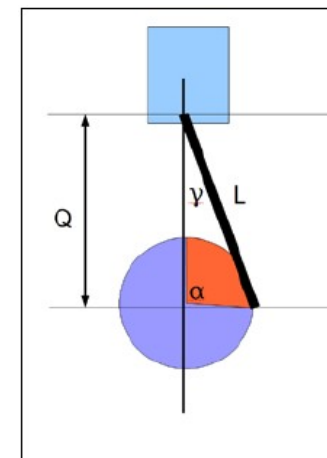
Высшая передача и оба распределительных вала будут также следовать за вращением махового колеса, но со скоростью, уменьшенной в два раза и с противоположным направлением вращения:

```
m.degrees(ob('Flywheel').RotX*-0.5)/10.0
```

Чтобы проиллюстрировать, как получить доступ к функциям в математическом модуле Питона **math**, мы не стали преобразовывать в градусы самостоятельно, а воспользовались функцией **degrees()**, поставляемой с модулем **math**.

Мы смоделировали распределительный вал с кулачком, указывающим точно вниз. Если мы хотим управлять вращением по оси X распределительного вала на входе посредством вращения ведущего вала, мы должны принять во внимание, что он движется на половинной скорости. Кроме того, его задержки вращения немного отстают, чтобы соответствовать циклу воспламенения цилиндра, поскольку он открывает входной клапан на начальном движении вниз и закрывает клапан как раз перед искрой воспламенения:

```
ob('DriveShaftPart').RotX/(2*m.pi)*18+9
```



Выражение для распределительного вала на выходе почти идентично за исключением времени запаздывания (здесь 24, но настройка этого двигателя не совсем соответствует реальной механике):

```
ob('DriveShaftPart').RotX/(2*m.pi)*18+24
```

Движение поршня ограничено только по вертикали, но его точное движение более сложно для вычисления. Нас интересует длина отрезка **Q** — смотрите предыдущий рисунок — и расстояние между центром ведущего вала и точкой, где шатун (**L** на диаграмме) соединяется с поршнем. Поскольку длина шатуна постоянна, изменение **Q** будет функцией от угла поворота  $\alpha$  ведущего вала. Расстояние от центра ведущего вала, до точки, где шатун связан с ведущим валом, фиксировано. Мы назовем это расстояние **R**. Теперь у нас есть треугольник со сторонами **Q**, **L**, и **R** и известен угол  $\alpha$ . Поскольку три из этих данных (**L**, **R**, и  $\alpha$ ) известны, мы можем вычислить **Q**, при использовании теоремы косинусов ([http://ru.wikipedia.org/wiki/Теорема\\_косинусов](http://ru.wikipedia.org/wiki/Теорема_косинусов)). Поэтому мы определяем функцию **q()** в файле **pydrivers.py**, которая возвращает длину **Q**, при заданных **L**, **R**, и  $\alpha$ :

```
def q(l,r,a): return r*cos(a)+sqrt(l**2-(r*sin(a))**2)
```

Выражение для канала поршня **LocZ** просто обращается к этой функции с соответствующими значениями аргументов:

```
p.q(1.542,0.655,ob('DriveShaftPart').RotX)
```

Точные значения для **L** и **R** были взяты из меша, используя координаты соответствующих вершин шатуна и ведущего вала в окне Transform Properties. (кнопка *N* в окне 3D-вида)

Для самого шатуна можно использовать то же выражение для LocZ-канала, но нужно так тщательно сделать соединение поршня и шатуна, чтобы они точно совпадали.

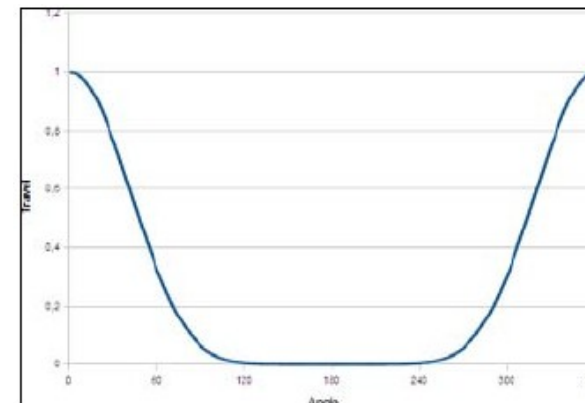
Однако, движение шатуна не ограничено только перемещением по оси Z, так как он вращается вокруг оси X с центром в точке, соединяющей шатун с поршнем. Угол вращения ( $\gamma$  на диаграмме) можно вывести из значений **L**, **R**, и  $\alpha$ :

```
def topa(l,r,a):  
    Q=q(l,r,a)  
    ac=acos((Q**2+l**2-r**2)/(2*Q*l))  
    if a%(2*pi)>pi : ac = -ac  
    return -ac
```

**Pydriver** выражение для RotX будет выглядеть вот так:

```
m.degrees(p.topa(1.542,0.655,ob('DriveShaftPart').RotX))/1  
0.0
```

Впускной и выпускной клапаны управляются вращением их соответствующих распределительных валов. Очертание кулачка очень сложно, так что здесь мы используем не фактическую форму его контура, а аппроксимируем ее, она выглядит достаточно хорошо (то есть, открытый клапан в функции еще оживленное движение в правильном моменте). Следующая картинка показывает движение клапана как функцию от угла вращения:



Наконец, в pydrivers.py мы определяем функцию spike(), которая принимает угол поворота распределительного вала как аргумент и возвращает значение между 0.0 и 1.0 которое резко возрастает в районе нулевого угла:

```
def spike(angle):  
    t = (cos(angle)+1.0)/2.0  
    return t**4
```

Сейчас клапан движется линейно, но линия, по которой он следует, наклонена на 10 градусов (вперед для впускного клапана, назад для выпускного клапана), теперь нам придется управлять двумя каналами, LocZ и LocY, каждый нужно умножить на правильное значение для создания наклонного движения. Поэтому мы определим две функции в pydrivers.py:

```
def valveZ(angle,tilt,travel,offset):  
    return cos(radians(tilt))*spike(angle)*travel+offset  
def valveY(angle,tilt,travel,offset):  
    return sin(radians(tilt))*spike(angle)*travel+offset
```

Обе функции возвращают расстояние в зависимости от угла поворота управляющего объекта. Tilt (наклон) - наклон клапана (в градусах), travel — максимальная длина пути, по которому проходит клапан вдоль наклонной линии, а offset (компенсация) - значение, которое позволяет регулировать позицию клапана. Соответствующие **pydriver**-выражения для LocZ и LocY-каналов впускного клапана:



```
p.valveZ(ob('CamInlet').RotX+m.pi,-10.0,-0.1,6.55)
и
p.valveY(ob('CamInlet').RotX+m.pi,-10.0,-0.1,-0.03)
```

(Выражения для выпускного клапана аналогичны, но с положительным углом tilt.)

До сих пор, все IPO-каналы были каналами объекта, такими как расположение и вращение. Но также возможно управлять другими каналами, ведь нам нужно изменять энергию лампы, помещенной в свечу зажигания. В `pydrivers.py` мы для начала определим вспомогательную функцию `topi()`, которая, в качестве аргументов, кроме угла вращения движущегося объекта принимает угол `h` (в радианах) и интенсивность `i`. `topi()` возвращает эту интенсивность, если угол двигающегося объекта находится между 0 и `h`, и ноль, если угол выйдет за пределы этого ряда. Поскольку угол на входе функции, возможно больше, чем  $2\pi$  (когда двигающийся объект пройдет больше чем полный круг), мы исправляем это выделенной операцией деления по модулю:

```
def topi(a,h,i):
    m = a%(2*pi)
    r=0.0
    if m<h: r=i
    return r
```

**pydriver**-выражение для канала энергии (называемый "Enerг" в редакторе Кривых IPO), может быть выражено следующим образом:

```
p.topi(ob('DriveShaftPart').RotX/2+m.pi,0.3,0.5)
```

Как видно, это выражение запустит «огонь» в свече зажигания при первых 17 градусах (0.3 радиан), установив энергию для этого цикла в 0.5 .

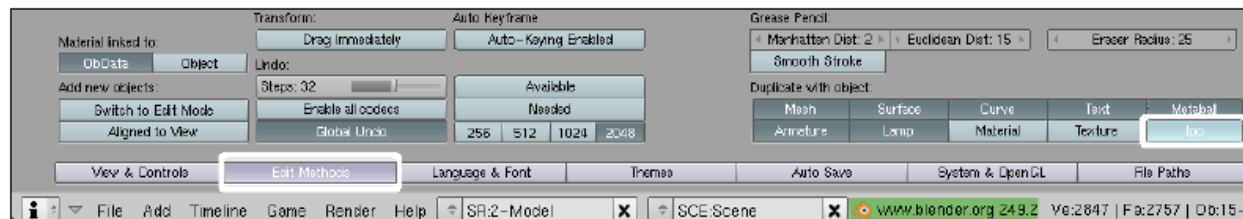
## Больше мощности — комбинирование нескольких цилиндров в двигателе

Как только мы смоделировали один цилиндр и позаботились о управлении движением отдельных частей, нашим следующим шагом будет дублирование цилиндров, для создания мотора как на вводной иллюстрации этой главы. В принципе мы можем просто выделить все

и продублировать, нажав **Shift + D**, отрегулировав время срабатывания каждого IPO-канала.

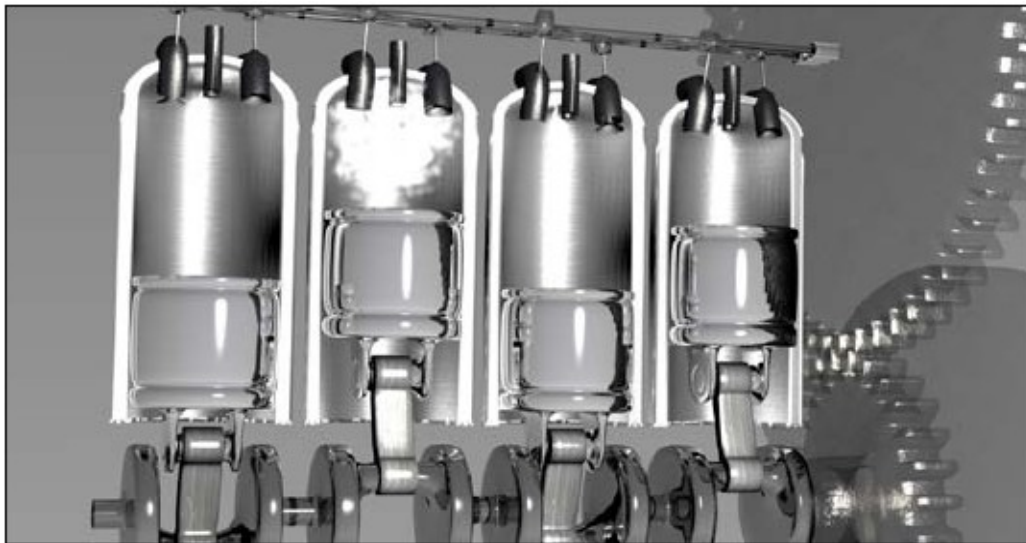
Но есть препятствие. Если мы используем **Shift + D**, вместо **Alt + D** мы получим одинаковые копии мешей объектов, вместо того чтобы просто воспользоваться ссылкой на первый объект. К тому же, мы ожидаем, что скопировали и остальные атрибуты объекта, такие как материалы, текстуры и IPO. Блендер, по-умолчанию, не дублирует вышеперечисленные категории, копируя только сам объект. Это получится неуклюже, так как изменение IPO первого поршня, к примеру, затронуло бы все остальные.

Мы могли бы сделать остальные копии уникальными впоследствии (нажав на поле количества пользователей этих кривых IPO, например, и подтвердив своё согласие со всплывающим вопросом **make single user?**), но было бы слишком утомительным повторять это для каждой копии отдельно.



Лучшим способом будет изменить настройки копирования объектов (**Duplicate with object**) в панели **Edit Methods**, как показано на скриншоте выше. Таким образом, кривые IPO, связанные с объектом, будут превращены в уникальные копии при дублировании объекта.

Результат нашей работы, четырехцилиндровый двигатель, передающий движение от ведущего вала к поршням доступен как `engine001.blend`. Изображение анимации доступной по адресу <http://vimeo.com/7170769>, показано на следующем скриншоте.



## Добавление простых ограничений

**Ограничения** (Constraints) могут быть применены к объектам и костям. В обоих случаях **ограничение** добавляется вызовом метода `append()` атрибута `constraints`. Наш следующий пример покажет, как мы можем ограничить движение стрелок часов из *rigged clock* (Глава 3, Группы вершин и материалы) для вращения вокруг оси *Z*. Код, определяющий функции для достижения поставленной задачи начинается с двух определений `import`, которые уменьшат длину кода:

```
from Blender.Constraint import Type
from Blender.Constraint import Settings
```

Функция принимает два аргумента: `obbones`, ссылка на объект Блендера, данные которого являются арматурой (то есть, не объект арматуры непосредственно) и `bone`, название кости, которую мы будем ограничивать. Важно понимать, что **ограничение**, которое мы связываем с костью, является не свойством арматуры, а позой объекта, содержащего арматуру. Множество объектов могут обращаться к одной и той же арматуре, и все позы будут связаны с

объектами, таким образом различные объекты, обращающиеся к той же самой арматуре, смогут принимать различные позы.

Итак, стартуя, функция сначала получает позу, а затем ссылку на кость, которую мы хотим ограничить. Выделенная строка показывает, как привязать **ограничение** (это аналогично тому, как если бы мы связывали **ограничение** с объектом Блендера вместо кости):

```
def zrotonly(obbones,bone):
    poseob = obbones.getPose()
    bigarmpose = poseob.bones[bone]
    c=bigarmpose.constraints.append(Type.LIMITROT)
    c[Settings.LIMIT]=Settings.LIMIT_XROT|
        Settings.LIMIT_YROT

    c[Settings.XMIN]=0.0
    c[Settings.XMAX]=0.0
    c[Settings.YMIN]=0.0
    c[Settings.YMAX]=0.0
    poseob.update()
```

Вновь присоединенное **ограничение** сохраняется в переменную `c` и в последующих строках видно, что различные атрибуты **ограничения**, становятся доступны подобно словарю. Сначала мы настраиваем атрибут `LIMIT` (это битовая маска), для ограничения вращения по осям *X* и *Y*. Далее, мы устанавливаем минимальное и максимальное значение вращения вокруг этих осей равным `0.0`, таким образом мы останавливаем любое движение. Например, в риггинге реалистичного скелета животного этими значениями можно задать пределы величин вращения к значениям, сопоставимыми с реальными соединениями между костями. И в конце, чтобы сделать изменения `pose` (позы) видимыми, мы обращаемся к методу `update()`.

## Определяем сложные ограничения

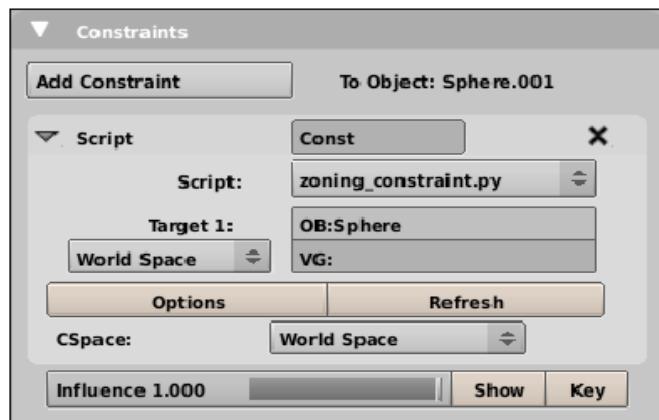
Там, где **pydrivers** предоставляют нам возможность управлять изменением одной IPO-кривой посредством изменения другой, **PyConstraints** (питон-ограничения) предоставляют нам способы задавать пределы изменения свойств объекта.

Конечно, в Блендер встроено много простых ограничений, таких как мы видели в предыдущем разделе, и часто комбинации простых ограничений достаточно для того, что нам нужно. Но если нам необходимо, чтобы наши объекты перемещались с места на место свободно в пределах не прямоугольной области, а например, для упрощения размещения светофоров и телефонных будок по сетке улиц. Как мы можем достичь этого? Введите `pyconstraints`.

`PyConstraints` - Питон-скрипты, которые присутствуют как текстовые блоки в текстовом редакторе Блендера и должны начинаться со строки комментария, идентифицирующей их как **ограничение**:

```
#BPYCONSTRAINT
```

**Ограничение** на Питоне должно содержать три функции с именами `doConstraint()`, `doTarget()`, и `getSettings()`. Первые две вызываются в любое время, когда мы двигаем или цель, или **ограничиваемый** объект, а последняя функция вызывается тогда, когда пользователь щелкает по кнопке **Options**, которая появляется, как только пользователь выбрал **pyconstraint**. Следующий скриншот показывает окно **Ограничений**, как только был выбран `pyconstraint`.



Самый легкий путь понять, что эти функции делают — посмотреть встроенный шаблон **ограничения**, который мы можем использовать в качестве основы, чтобы написать наши собственные ограничения. Он доступен в текстовом редакторе по меню **Text | Script Templates | Script Constraint**. При выборе этого меню будет

создан новый текстовый блок, который можно выбрать в выпадающем списке внизу окна текстового редактора.

## Шаблон ограничения в Блендере

Шаблон **ограничения** содержит также много полезных комментариев, но здесь мы перечислим, по большей части голые функции. Кроме того, шаблон создает окно с фиктивными свойствами. Мы столкнемся со свойствами в следующей части, так что наш пример функции `getSettings()` здесь будет почти пуст. Как показано, функции будут осуществлять функциональное ограничение, однако, ничего фактически не будет ограничено. Расположение, вращение, и масштаб **ограничиваемого** объекта останутся без изменений.

```
def doConstraint(obmatrix, targetmatrices, idprop):
    # Выделить компоненты преобразования для быстрого
    # доступа.
    obloc = obmatrix.translationPart() # перемещение
    obrot = obmatrix.toEuler()         # вращение
    obsca = obmatrix.scalePart()       # масштабирование

    # код, который реально меняет положение, вращение и
    # масштабирование, расположен здесь

    # Конвертация обратно в матрицы положения, вращения,
    # масштаба,
    mtxloc = Mathutils.TranslationMatrix(obloc)
    mtxrot = obrot.toMatrix().resize4x4()
    mtxsca = Mathutils.Matrix([obsca[0],0,0,0],
    [0,obsca[1],0,0],
                                [0,0,obsca[2],0], [0,0,0,1])

    # Рекомбинация отдельных элементов в матрицу
    # преобразования.
    outputmatrix = mtxsca * mtxrot * mtxloc

    # Возвращаем новую матрицу.
    return outputmatrix
```

В функцию `doConstraint()` передаётся матрица преобразований **ограничиваемого** объекта и список матриц

преобразования для каждого целевого объекта. Она также получает словарь свойств **ограничения**, к которым можно получить доступ по имени.

Первая вещь, которую мы делаем, - выделяем отдельные компоненты матрицы преобразования — перемещение, вращение, и масштаб **ограничиваемого** объекта. Частью перемещения будет вектор положения  $x$ ,  $y$ ,  $z$ , частью масштаба будет вектор масштабирующих коэффициентов вдоль осей  $x$ ,  $y$ ,  $z$ . Часть вращения будет представлена вектором Эйлера с вращением вокруг трех основных осей. (углы Эйлера очень упрощают работу с вращениями в трехмерном пространстве, но по началу являются довольно трудными для понимания. В википедии есть материал на эту тему [http://ru.wikipedia.org/wiki/Углы\\_Эйлера](http://ru.wikipedia.org/wiki/Углы_Эйлера), но пока что легче думать о углах Эйлера как о вращении вокруг осей  $x$ ,  $y$ ,  $z$ . *Углы Эйлера трудны? Вот от кватернионов реально мозг взрывается! - возмущение пер.*) Мы можем разделить любую матрицу преобразования целевого объекта так, как нам нужно, и затем изменить компоненты матрицы преобразования **ограничиваемого** объекта по своему усмотрению.

Функция, показанная здесь, не делает ничего, но преобразует различные компоненты преобразования обратно в матрицы, используя методы API (где это доступно), и затем рекомбинирует их, используя матричное умножение в единственную матрицу, которая впоследствии возвращается.

Функция `doTarget()` вызывается до вызова `doConstraint()` и даёт нам возможность манипулировать целевой матрицей прежде, чем она будет передана в `doConstraint()`. Аргументы - целевой объект, под-цель (или Кость или группа вершин для целевой арматуры или меша соответственно), целевая матрица, и свойства ограничения. В следующем разделе мы используем эту возможность для сохранения ссылки на целевой объект в свойствах, чтобы `doConstraint()` могла иметь доступ к этой информации. Если мы не хотим ничего изменять, то достаточно вернуть целевую матрицу, как показано в следующем коде:

```
def doTarget(target_object, subtarget_bone, target_matrix,
            id_properties_of_constraint):
    return target_matrix
```

Точно также, если нет необходимости предлагать пользователю возможность определять дополнительные свойства, `getSettings()`, может иметь просто оператор `return` (возврат). Если мы *хотим* показать всплывающее меню, `getSettings()` - то место, где это нужно сделать. Мы также увидим такой пример в следующем разделе. Следующий код будет корректной реализацией "ничегонеделания":

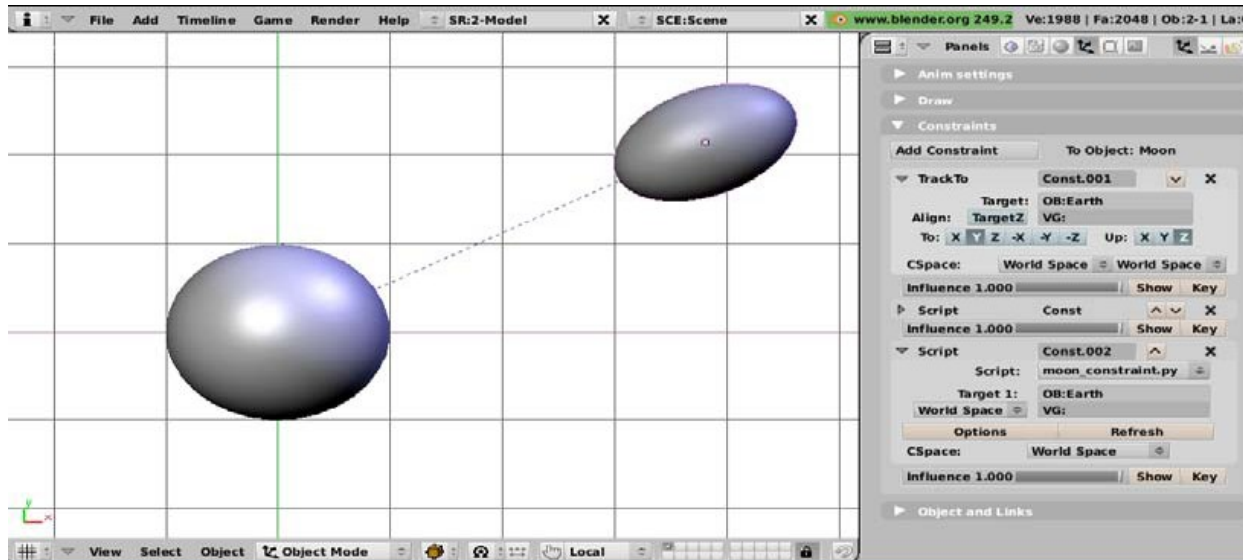
```
def getSettings(idprop):
    return
```

## Вы тоже находите меня притягательным?

Когда Луна и Земля вращаются вокруг друг друга, каждая из них чувствует гравитационное притяжение другой. На земле это приводит к приливам и отливам, но твердые тела Земли и Луны также искажаются, хотя этот эффект небольшой. Теперь известно намного больше о приливах и отливах, чем только притяжение ([http://ru.wikipedia.org/wiki/Прилив\\_и\\_отлив](http://ru.wikipedia.org/wiki/Прилив_и_отлив)), но мы можем показать гравитационные искажения в гипертрофированном виде с применением ограничений.

Один из способов сделать это - использовать **ограничение** `TrackTo`, чтобы ориентировать ось нашего ограничиваемого объекта к притягивающему объекту и добавить второе **ограничение**, которое масштабирует **ограничиваемый** объект вдоль этой оси. Величина масштаба будет обратно зависима от расстояния между **ограничиваемым** объектом и целевым объектом. Эффект проиллюстрирован на следующем скриншоте, где эффект **ограничения** `TrackTo` объединен со скриптовым **ограничением** `moon_constraint.py`.





Мы должны написать это зависимое от расстояния масштабирование самостоятельно. Если мы возьмём шаблон ограничения, предоставляемый Блендером, мы можем оставить функции `doTarget()` и `getSettings()` как есть, но мы должны написать подходящую `doConstraint()` (полный код доступен как `moon_constraint.py`):

```
def doConstraint(obmatrix, targetmatrices, idprop):
    obloc = obmatrix.translationPart() # Положение
    obrot = obmatrix.toEuler()         # Вращение
    obsca = obmatrix.scalePart()       # Масштаб

    tloc = targetmatrices[0].translationPart()
    d = abs((obloc-tloc).length)
    d = max(0.01,d)
    f = 1.0+1.0/d
    obsca[1]*=f

    mtxloc = Mathutils.TranslationMatrix(obloc)
    mtxrot = obrot.toMatrix().resize4x4()
    mtxsca = Mathutils.Matrix([obsca[0],0,0,0],
                               [0,obsca[1],0,0],[0,0,obsca[2],0], [0,0,0,1])

    outputmatrix = mtxsca * mtxrot * mtxloc

    return outputmatrix
```

Мы пропустили все строки, имеющие отношение к свойствам, так как мы не используем никаких настраиваемых пользователем свойств для этого **ограничения**. Выделенные строки показывают, что мы должны делать для вычисления зависимого от расстояния масштабирования.

В первой строке получаем позицию нашей цели. Затем мы вычисляем расстояние между ограничиваемым объектом и целью и определяем предел его минимума (чуть-чуть больше нуля), чтобы предотвратить деление на ноль в следующей выделенной строке. Используемая здесь формула отнюдь не является аппроксимацией какого-либо гравитационного влияния, но ведет себя достаточно хорошо для наших целей; коэффициент масштабирования будет близок к  $1.0$ , если  $d$  очень большое, и гладко возрастает при уменьшении расстояния  $d$ . Последняя выделенная строка показывает, что мы изменяем масштаб только по оси  $y$ , то есть по оси, которую мы ориентируем на целевой объект с помощью ограничения `TrackTo`.

#### Циклическая зависимость:



Если оба объекта имеют сравнимую массу, гравитационное искажение должно быть сравнимого размера на обоих объектах. У нас может появиться искушение добавить **ограничения** `TrackTo` и `moon_constraint.py` к обоим объектам, чтобы видеть эффект воздействия их друг на друга, но, к несчастью, это не будет работать, поскольку это создаст циклическую зависимость, и Блендер запустится с ошибкой.

## Привязка к вершинам меша

Это похоже на режим "snap to vertex" (привязка к вершине), который доступен в Блендере из меню **Object | Transform | Snap** (информацию о привязках смотрите тут: [http://wiki.blender.org/index.php/Doc:Manual/Modelling/Meshes/Snap\\_to\\_Mesh](http://wiki.blender.org/index.php/Doc:Manual/Modelling/Meshes/Snap_to_Mesh)), за исключением того, что эффект не постоянный (объект

вернётся в свою изначальную позицию, как только **ограничение** будет удалено) и силу **ограничения** можно регулировать (даже анимировать), изменяя движок Influence (Влияние).

В **ограничениях**, которые мы до сих пор разрабатывали, нам нужна была только позиция целевого объекта для вычисления эффектов на **ограничиваемом** объекте. Эту позицию было легко применять в функции `doConstraint()`, так как матрицы целей принимались в качестве аргументов. Теперь мы все же встречаем другой вызов: если мы хотим привязать к вершине, мы должны иметь доступ к данным меша целевого объекта, но целевой объект не передаётся в функцию `doConstraint()`.

Путь в обход этого препятствия - аргумент `idprop`, который передаётся в `doConstraint()`. Перед тем, как вызвать `doConstraint()`, Блендер сначала вызывает `doTarget()` для каждого целевого объекта. Эта функция передаётся в виде ссылки на целевой объект и в свойства **ограничения**. Это позволяет нам включать ссылку на целевой объект в эти свойства, и поскольку эти свойства передаются в `doConstraint()`, это обеспечивает нас средствами для передачи необходимой информации в `doConstraint()` для получения *Меш*-данных. Есть мелочь, которую мы всё-же рассмотрим здесь: свойствами в Блендере могут быть только числа или строки, так что мы не можем на самом деле хранить ссылку на объект, но должны удовольствоваться его именем. Поскольку имя является уникальным, и функция Блендера `Object.Get()` предоставляет способ извлекать объект по имени, это - не проблема.

Код для функций `doConstraint()` и `doTarget()` будет выглядеть так (полный код находится в `zoning_constraint.py`):

```
def doConstraint(obmatrix, targetmatrices, idprop):
    obloc = obmatrix.translationPart().resize3D()
    obrot = obmatrix.toEuler()
    obsca = obmatrix.scalePart()

    # Получаем целевой меш
    to = Blender.Object.Get(idprop['target_object'])
    me = to.getData(mesh=1)

    # получаем местоположение целевого объекта
```

```
tloc = targetmatrices[0].translationPart().resize3D()

# ищем ближайшую вершину на целевом объекте
smallest = 1000000.0
delta_ob=tloc-obloc
for v in me.verts:
    d = (v.co+delta_ob).length
    if d < smallest:
        smallest=d
        sv=v
obloc = sv.co + tloc

# восстанавливаем матрицу объекта
mtxrot = obrot.toMatrix().resize4x4()
mtxloc = Mathutils.TranslationMatrix(obloc)
mtxsca = Mathutils.Matrix([obsca[0],0,0,0],
                           [0,obsca[1],0,0],
                           [0,0,obsca[2],0],
                           [0,0,0,1])
outputmatrix = mtxsca * mtxrot * mtxloc
return outputmatrix
```

```
def doTarget(target_object, subtarget_bone, target_matrix,
            id_prop_of_constr):
    id_props_of_constr['target_object']=target_object.name
    return target_matrix
```

Выделенные строки показывают, как мы передаем имя целевого объекта в `doConstraint()`. В `doConstraint()` мы сначала извлекаем целевой меш. Это может вызвать исключение, например, если целевой объект не является мешем, но оно будет поймано Блендером самостоятельно. Тогда **ограничение** не станет воздействовать, ошибка будет показана в консоли, но Блендер продолжит нормальную работу.

Как только у нас будут меш-данные целевого объекта, мы извлекаем позицию целевого объекта. Нам нужно это, поскольку все координаты вершин считаются относительно неё. Затем мы сравниваем позицию **ограничиваемого** объекта с позициями всех вершин целевого меша и запоминаем ближайшую, чтобы вычислить позицию **ограничиваемого** объекта. Наконец, мы восстанавливаем матрицу преобразований **ограничиваемого** объекта, объединяя различные компоненты преобразований, как и раньше.

## Выравнивание вдоль вершинной нормали

Теперь, когда мы смогли привязать объект к ближайшей вершине в целевом меше, мы можем видеть, что что-то пропустили: объект не сориентирован в правильном направлении. Это не всегда является проблемой, например, деревья обычно направлены вверх, но во многих ситуациях было бы неплохо, если бы мы смогли сориентировать ограничиваемый объект перпендикулярно поверхности. Это делается также для всех практических целей, как ориентация ограничиваемого объекта вдоль вершинной нормали той вершины, к которой мы сделали привязку.

Следовательно, после обнаружения ближайшей вершины, мы определяем угол между вершинной нормалью и осью z (то есть, мы произвольно определяем направление Z как 'вверх'), затем вращаем **ограничиваемый** объект на тот же самый угол вокруг оси, перпендикулярной как вершинной нормали, так и оси z. Это ориентирует **ограничиваемый** объект вдоль этой вершинной нормали. Если **ограничиваемый** объект был вручную повернут до добавления **ограничения**, эти предыдущие вращения будут потеряны. Если это - не то, что нам нужно, мы можем применить все вращения перед добавлением **ограничения**.

Для того, чтобы осуществить эту возможность выравнивания, наш код изменится (zoning\_constraint.py уже содержит эти изменения): doConstraint() должно вычислять поворотную часть матрицы преобразования. Мы должны вычислить угол вращения, ось вращения, и затем новую матрицу вращения. Выделенная часть следующего кода показывает, что основные инструменты для этих вычислений уже предусмотрены модулем Mathutils:

```
vnormal = sv.no
if idprop['NormalAlign'] :
    zunit=Mathutils.Vector(0,0,1)
    a=Mathutils.AngleBetweenVecs(vnormal,zunit)
    rotaxis=zunit.cross(vnormal)
    rotmatrix=Mathutils.RotationMatrix(a,4,"r",rotaxis)
    mtxrot = rotmatrix
else:
    mtxrot = obrot.toMatrix().resize4x4()
```

В предыдущем коде мы можем видеть, что мы сделали выравнивание зависимым от свойства NormalAlign. Только если оно задано, мы вычисляем необходимое преобразование. Следовательно, нам нужно адаптировать также функцию getSettings(), поскольку пользователю нужен способ выбирать, нужно ему выравнивание или нет:

```
def getSettings(idprop):
    if not idprop.has_key('NormalAlign'):
        idprop['NormalAlign'] = True

    align = Draw.Create(idprop['NormalAlign'])

    block = []
    block.append("Additional restrictions: ")
    block.append(("Alignment: ",align,
                  "Align along vertex normal"))

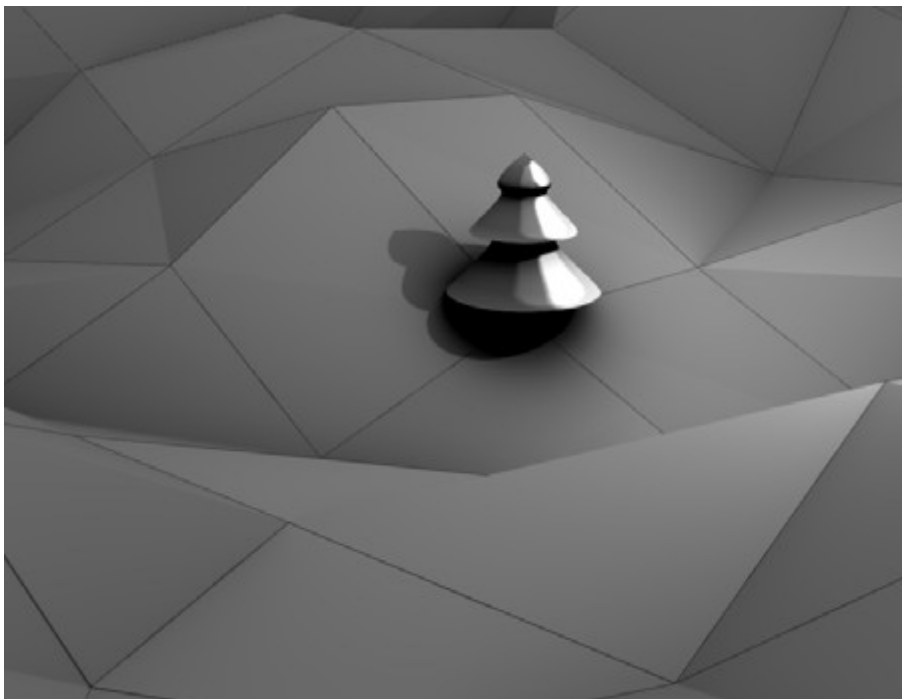
    retval = Draw.PupBlock("Zoning Constraint", block)

    if (retval):
        idprop['NormalAlign']= align.val
```

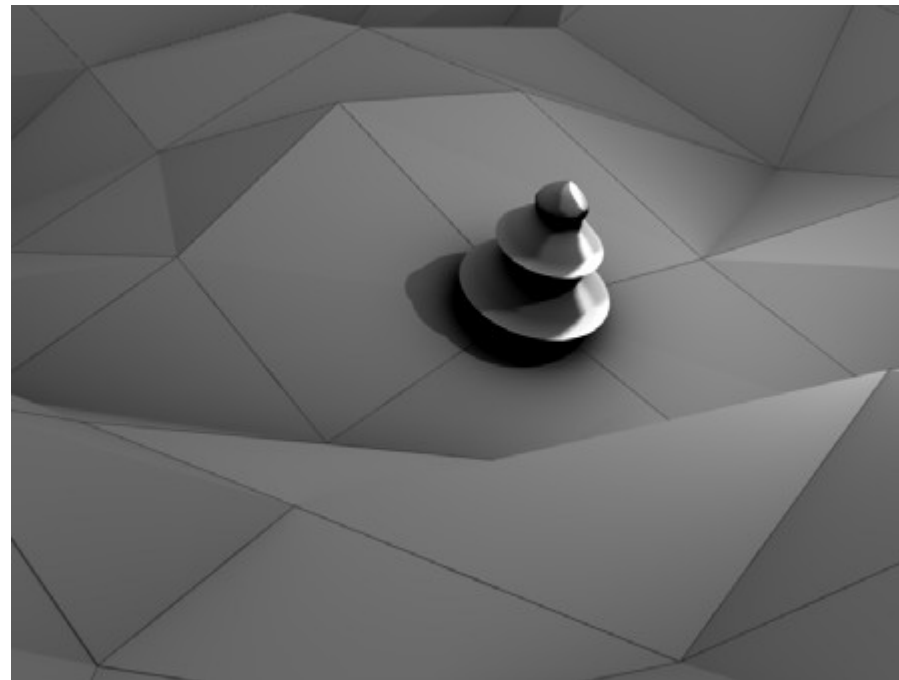
Как показано, свойство NormalAlign по умолчанию будет установлено в True (Истина). Опция затем будет представлена как простое выпадающее меню с кнопкой-переключателем. Если пользователь щелкает за пределами меню или нажимает клавишу *Esc*, PupBlock() вернёт значение None, мы не будем изменять свойство NormalAlign. В противном случае, оно будет установлено в соответствии со значением кнопки-переключателя.

Эффекты показаны на иллюстрациях. Первая показывает небольшую ёлку с **ограничением** привязки к вершине простой подразделенной плоскости земли. Она привязана в точную позицию вершины, но ось z указывает ровно вверх вдоль глобальной оси z. Скриншот показывает ёлку с **ограничением** к вершине в скалистом пейзаже.

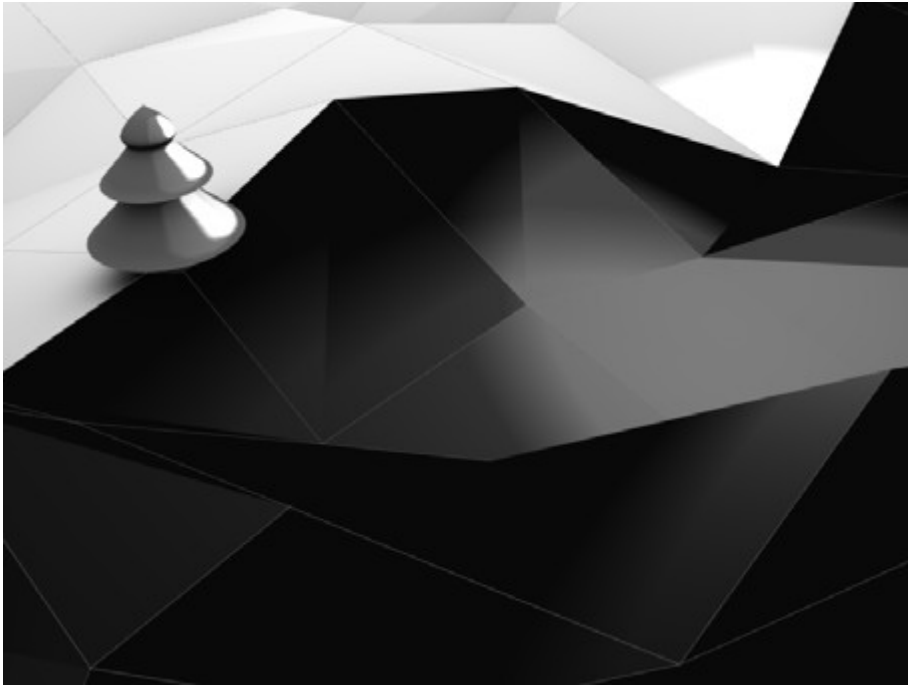




Если мы включим свойство `NormalAlign`, мы увидим, что модель дерева больше не указывает ровно вверх, но что ось  $z$  выровнена вдоль направления вершинной нормали той вершины, к которой она привязана. Следующий скриншот показывает елку с **ограничением** к вершине и выравниванием вдоль вершинной нормали.



Также возможно **ограничить** вершины, к которым модель может быть привязана, ещё дальше, например, именно к вершинам, принадлежащим к группе вершин. В следующей иллюстрации наша модель не сможет переместиться за пределы группы вершин, которая показана белым. Как это может быть выполнено, показано в следующем разделе.



## Привязка к вершинам в вершинной группе

Что, если мы хотим определить конкретно те вершины, к которым мы можем привязать объект? Это можно достигнуть, определив группу вершин, и, затем, рассматривая только вершины из этой группы в качестве кандидатов, к которым можно привязывать. Код необходимый для этого, увеличится всего на несколько строк, и важная часть `doConstraint()` будет выглядеть примерно так (выделенный код показывает дополнительные строки, имеющие дело с проверкой на принадлежность к группе вершин):

```
# получаем целевой меш
to = Blender.Object.Get(idprop['target_object'])
me = to.getData(mesh=1)

# получаем положение целевого меша
tloc = targetmatrices[0].translationPart().resize3D()

# ищем ближайшую вершину в целевом объекте
smallest = 1000000.0
delta_ob=tloc-obloc
```

```
try:
    verts = me.getVertsFromGroup(idprop['VertexGroup'])
    for vi in verts:
        d = (me.verts[vi].co+delta_ob).length
        if d < smallest :
            smallest = d
            si = vi
        obloc = me.verts[si].co+tloc
        vnormal = me.verts[si].no
except AttributeError:
    for v in me.verts:
        d = (v.co+delta_ob).length
        if d < smallest:
            smallest=d
            sv=v
    obloc = sv.co + tloc
    vnormal = sv.no
```

Автор здесь нарушил одно из важнейших правил качественного программирования, которое гласит «**Нем дублированию кода!**» Текст после **try** желательно переписать, например, так:

```
try:
    verts = me.getVertsFromGroup(idprop['VertexGroup'])
except AttributeError:
    verts = range(len(me.verts))
for vi in verts:
    d = (me.verts[vi].co+delta_ob).length
    if d < smallest :
        smallest = d
        si = vi
    obloc = me.verts[si].co+tloc
    vnormal = me.verts[si].no
```

– *Примечание занудного и наглого переводчика Striver'a*

Конструкция `try/except` гарантирует, что если свойство `VertexGroup` ссылается на несуществующую группу вершин, мы получим шанс проверить все вершины. Конечно, нам теперь нужен способ для пользователя, позволяющий выбирать группу вершин, так что функцию `getSettings()` нужно тоже адаптировать. Мы довольствуемся простым полем ввода строки, где можно набрать имя

группы вершин. Нет проверки на существование группы, и если мы не хотим ограничиваться привязкой к группе вершин, тогда мы можем или оставить это поле ввода пустым, или занести имя несуществующей группы. Не слишком изящно, но это работает (дополнительные строки выделены):

```
def getSettings(idprop):
    if not idprop.has_key('VertexGroup'):
        idprop['VertexGroup'] = 'Zone'
    if not idprop.has_key('NormalAlign'):
        idprop['NormalAlign'] = True

    vgroup = Draw.Create(idprop['VertexGroup'])
    align = Draw.Create(idprop['NormalAlign'])

    block = []
    block.append("Additional restrictions: ")
    block.append(("Vertex Group: ",vgroup,0,30,"Vertex
        Group to restrict location to"))
    block.append(("Alignment: ",align,
        "Align along vertex normal"))

    retval = Draw.PupBlock("Zoning Constraint", block)

    if (retval):
        idprop['VertexGroup']= vgroup.val
        idprop['NormalAlign']= align.val
```

Следующий скриншот показывает, как может выглядеть поле ввода для группы вершин:



Заметьте, что скриптовое **ограничение** также обеспечивает пользователя полем ввода строки VG, которая может ссылаться на группу вершин, но это отличается от поля ввода группы вершин, которую мы показываем пользователю во всплывающих **Опциях**. Это поле VG будет изменять способ *рассматривания ограничением* цели. Если здесь задаётся корректная группа вершин, целевая матрица передаваемая в `doConstraint()`, будет усреднённой позицией вершин в вершинной группе.

## Итог

В этой главе мы увидели как различные свойства анимации могут связываться вместе, и как мы можем ограничивать пространственные свойства объектов сложными **ограничениями**. Мы узнали как:

- Управлять одним **IPO** из другого посредством выражения на Питоне
- Обходить некоторые ограничения, присущие **управляющим объектам**
- Ограничивать движение объектов и костей, добавляя **ограничения (constraints)**
- Писать **ограничение** на Питоне, которое привязывает объект к ближайшей вершине на другом объекте

Затем мы взглянем на то, как выполнять некоторое действие всякий раз, когда мы передвигаемся по кадрам в нашей анимации.

## Действия при изменениях кадров

Кроме тех многих мест, с которыми мы столкнулись, где Питон может быть использован в Блендере, мы теперь посмотрим на скрипты, которые можно использовать для действия при определенных событиях. В этот тип скриптов входят две изюминки - **скриптсвязи** и **обработчики пространства**.

**Скриптсвязи (Script links)** являются скриптами, которые могут быть ассоциированы с объектами Блендера (*Меши*, *Камеры*, и так далее, а также *Сцены* и объекты *Мира*), и их можно настроить так, чтобы они автоматически срабатывали в следующих случаях:

- Сразу перед рендером кадра
- Сразу после рендера кадра
- Когда кадр сменяется
- Когда объект скорректирован
- Когда данные объекта скорректированы

Объекты Сцены можно связать с ассоциированными с ними скриптами, которые могут вызываться в двух дополнительных случаях:

- При загрузке *.blend* файла
- При сохранении *.blend* файла

**Обработчики пространства (Space handlers)** являются скриптами на Питоне, которые вызываются всякий раз, когда окно 3D-вида перерисовывается или обнаружено действие клавиатуры или мыши. Их основным применением является расширение возможностей интерфейса пользователя Блендера.

В этой главе вы узнаете:

- Что такое скриптсвязи и обработчики пространства

- Как осуществлять деятельность при каждом изменении кадров в анимации
- Как ассоциировать дополнительную информацию с объектом
- Как сделать, чтобы объект появлялся или исчезал, изменяя его слой или прозрачность
- Как осуществить схему, при которой с объектом в каждом кадре связывается различные меши
- Как прибавить функциональности 3D-виду

## Анимация видимости объектов

Часто текущей задачей в создании анимации является желание заставить объект исчезать или тускнеть в определенном кадре, либо ради самого эффекта, или чтобы заменить объект другим для достижения некоего драматического воздействия (как например, взрыв или превращение кролика в мяч).

Существует много способов создавать эти эффекты, и большинство из них конкретно не связаны со скриптсвязями, реагирующими на изменения кадра (многие могут также просто быть ключами анимации). Тем не менее, мы посмотрим на два метода, которые можно легко приспособить ко всем ситуациям такого рода, даже к тем, которые не легко осуществить ключами. Например, нам требуется несколько специфическое поведение параметра, которое легко можно сформулировать в выражении, но неудобно ловить в IPO.

## Потускнение материала

Нашим первым примером будет изменение **диффузного цвета** материала. Это будет подобно простому изменению прозрачности

(transparency), но на иллюстрациях такие изменения легче увидеть с диффузным цветом.

Нашей целью является выцветание диффузного цвета от черного к белому и обратно, в течении двухсекундного периода. Мы, следовательно, определяем функцию *setcolor()*, которая принимает материал и изменяет его диффузный цвет (атрибут *rgbColor*). Предполагается, что частота 25 кадров в секунду и, следовательно, первая строка получает номер текущего кадра и выполняет *деление по модулю*, чтобы определить, какая часть текущей целой секунды пройдена.

Выделенная строка в следующем куске кода определяет, в четной или нечетной секунде мы находимся. Если мы в четной секунде, мы наращиваем диффузный цвет до белого, так что мы просто сохраняем нашу вычисленную долю. Если мы в нечетной секунде, мы затемняем диффузный цвет до черного, так что мы вычитаем долю из максимально возможного значения (25). Наконец, мы масштабируем нашу величину в пределы между 0 и 1 и назначаем её во все три цветовых компонента для получения оттенка серого:

```
import Blender
```

```
def setcolor(mat):
    s = Blender.Get('curframe')%25
    if int(Blender.Get('curframe')/25.0)%2 == 0:
        c = s
    else:
        c = 25-s
    c /= 25.0
    mat.rgbCol = [c,c,c]

if Blender.bylink and Blender.event == 'FrameChanged':
    setcolor(Blender.link)
```

Скрипт заканчивается важной проверкой: *Blender.bylink* будет *Истиной* (True), только в том случае, если этот скрипт был вызван как script handler, и в этом случае *Blender.event* содержит тип события. Мы хотим действовать только при изменении кадра, так что мы проверяем здесь наличие этого события. Если эти условия удовлетворены, мы передаем *Blender.link* в нашу функцию *setcolor()*, так как там содержится объект, с которым наш *связанный скрипт*

ассоциирован - в нашем случае это будет объект *Material* (Материал). (Этот скрипт доступен как *MaterialScriptLink.py* в файле *scriptlinks.blend*.)

Следующим пунктом в нашем списке нужно соединить скрипт с объектом, чей материал мы хотим изменить. Следовательно, мы выбираем объект, и в **Окне Кнопок** выбираем **панель Script**. В панели **Scriptlinks** (Скриптсвязи), мы включаем скриптсвязи (кнопка *Enable Script Links*) и выбираем кнопку **MaterialScriptLinks**. (Если нет кнопки **MaterialScriptLinks**, тогда выбранному объекту не был назначен материал. Убедитесь в том, что он есть.) Там должна теперь появиться надпись **Select Script link** с кнопкой **New** (Новый). Щелкните на **New**, появится выпадающий список с доступными скриптами (файлы в текстовом редакторе). В нашем случае мы выбираем *MaterialScriptLink.py*, и на этом всё. Мы можем теперь протестировать нашу скриптсвязь, изменяя кадр в 3D-виде (с помощью клавиш стрелок). Цвет нашего объекта должен изменяться при изменении номера кадра. (Если у цвета не видно изменений, проверьте тип отображения в 3D-виде, должно быть solid или shaded.)



## Изменение слоев

Если мы хотим изменить **видимость** объекта, изменение назначенного слоя (слоёв) - более распространённая и мощная техника, чем изменения свойств материала. Изменение назначенного слоя имеет, например, преимущество в том, что мы можем сделать объект полностью невидимым для ламп, которые сконфигурированы на освещение только определенных слоёв, и множество аспектов анимации (например, отклонение частиц воздействием полей) также могут быть ограничены определенными слоями. Также, изменение слоев не ограничено объектами со связанными с ними материалами. Вы можете точно так же легко изменить слой для *Лампы* или *Камеры*.

В нашем следующем примере мы хотим назначить объекту слой 1, если количество пройденных секунд - четное, и слой 2, если время в секундах нечетное. Скрипт, осуществляющий это, очень подобен нашему скрипту, изменяющему материал. Реальная работа производится посредством функции *setlayer()*. Первая строка вычисляет слой, в котором объект должен находиться в текущем кадре, а следующая строка (выделенная) назначает список индексов слоя (состоящий из единственного слоя в данном случае) атрибуту *layers* объекта. Последние две строки функции *setlayer()* гарантируют, что изменение слоя действительно станет видимым в Блендере.

```
import Blender

def setlayer(ob):
    layer = 1+int(Blender.Get('curframe')/25.0)%2
    ob.layers = [ layer ]
    ob.makeDisplayList()
    Blender.Window.RedrawAll()

if Blender.bylink and Blender.event == 'FrameChanged':
    setlayer(Blender.link)
```

Как и в нашем предыдущем скрипте, последние строки нашего скрипта проверяют, что он был вызван как скриптсвязь и по событию изменения кадров, и если это так, передают связанный объект в функцию *setlayer()*. (Скрипт доступен как *OddEvenScriptlink.py* в файле *scriptlinks.blend*.)

Все, что осталось сделать, это назначить скрипт как *скриптсвязь (scriptlink)* выбранному объекту. Снова, это выполняется в **Окне Кнопок | панель Script**, щелкая по кнопке **Enabling Script Links** в панели **Scriptlinks** (если это необходимо, она могла все ещё быть выбранной после нашего предыдущего примера. Это глобальный выбор, то есть, включено или выключено для всех объектов). На этот раз мы выбираем *скриптсвязи объекта* вместо *скриптсвязей материала* и щелкаем на **New**, чтобы выбрать *OddEvenScriptlink.py* из выпадающего списка.

## Обратный отсчет - анимация таймера с помощью скриптсвязи

Одна из возможностей использовать скриптсвязи, которые действуют при изменении кадров - возможность модифицировать текущий меш, или с помощью изменения вершин Меш-объекта, или посредством ассоциации с объектом Блендера полностью другого меша. Это невозможно при использовании IPO, так как они ограничены ключами формы, которые интерполируются между предопределёнными формами с той же топологией меша (то же число вершин, соединенных таким же образом). Это истинно также для объектов кривых и текста.

Одним из применений этой техники будет создание объекта счетчика, который отобразит время в секундах с тех пор, как началась анимация. Это выполнено изменением текста объекта *Text3d* с помощью его метода *setText()*. Функция *setcounter()* в следующем коде делает как раз это вместе с необходимыми действиями, чтобы скорректировать отображение в Блендере. (Скрипт доступен как *CounterScriptLink.py* в файле *scriptlinks.blend*.)

```
import Blender

objectname='Counter'
scriptname='CounterScriptLink.py'

def setcounter(counterob):
    seconds = int(Blender.Get('curframe')/25.0)+1
    counterob.getData().setText(str(seconds))
    counterob.makeDisplayList()
```

```
Blender.Window.RedrawAll()
```

```
if Blender.bylink:
```

```
    setcounter(Blender.link)
```

```
else:
```

```
    countertext = Blender.Text3d.New(objectname)
```

```
    scn = Blender.Scene.GetCurrent()
```

```
    counterob = scn.objects.new(countertext)
```

```
    setcounter(counterob)
```

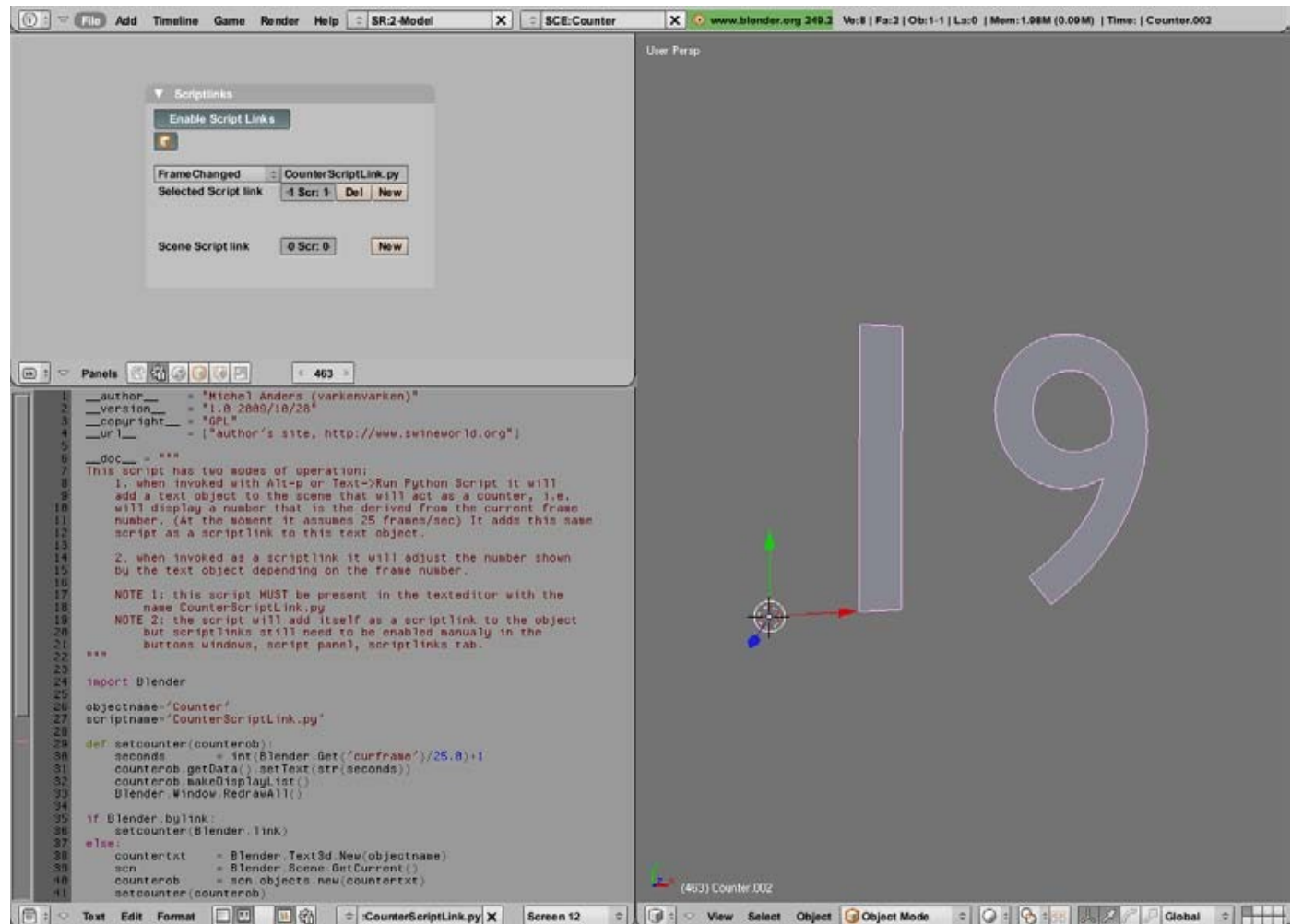
```
counterob.clearScriptLinks([scriptname])
```

```
counterob.addScriptLink(scriptname, 'FrameChanged')
```

Этот скрипт может быть ассоциирован в виде скриптсвязи с любым объектом Text3d, как показано прежде. Тем не менее, если запустить его с помощью Alt + P из текстового редактора, он создаст новый объект Text3d и присоединит себя к этому объекту как связанный скрипт. Выделенная строка показывает такую же простую проверку на его наличие, как в предыдущих скриптах, но в данном случае мы выполняем также некоторое действие в случае, если скрипт не был вызван как скриптсвязь (после else). Последние две выделенных строки показывают, как мы соединяем скрипт с вновь созданным объектом. Сначала, мы удаляем (clear) любую скриптсвязь с тем же именем, которая, возможно, была связана раньше. Это делается, чтобы предотвратить связывание этой же скриптсвязи более одного раза, что допустимо, но едва ли полезно. Затем, мы добавляем скрипт как скриптсвязь, которая будет вызываться, когда происходит изменение кадра (FrameChanged). Скриншот показывает 3D-вид с кадром из анимации вместе с окном Кнопок (слева вверху), который содержит список ассоциаций скриптсвязей с объектом.



Заметьте, что хотя возможно соединить скриптсвязь с объектом Блендера из скрипта на Питоне, скриптсвязи для него должны быть включены вручную, чтобы действительно работать! (Во вкладке ScriptLinks). В API Питона Блендера нет функциональности, позволяющей сделать это из скрипта.





## Я буду следить за вами

Иногда, при работе с сложным объектом, трудно следить за важной деталью, так как она может быть загорожена другими частями геометрии. В такой ситуации, было бы неплохо подсвечивать определенные вершины, чтобы они каким-то образом оставались видимыми, независимо от ориентации и режима *редактирования*.

**Обработчики пространства** (Space handlers) предоставляют нам способ выполнять действия всякий раз, когда окно 3D-вида перерисовывается, или когда обнаружено действие клавиатуры или мыши. Эти действия также могут включать рисование в области 3D-вида, так что мы сможем добавить **подсвечивание** (*Highlight*) в любом месте, где нам нравится.

Как нам определить, какие вершины мы хотели бы подсветить? Блендер уже предоставляет нам унифицированный способ группировать наборы вершин в виде вершинных групп, так что всё, что мы должны сделать, это позволить пользователю указать, какую группу вершин он хотел бы подсветить. Затем мы сохраним имя этой выбранной группы вершин как свойство объекта (object property). Свойства объектов предназначены для использования в игровом движке, но нет причин, почему мы не можем использовать их в качестве средства постоянно хранить нашу выбранную группу вершин.

И так, снова у нас есть скрипт, который будет вызываться двумя способами: как обработчик пространства (то есть, всякий раз, когда окно 3D-вида перерисовывается, чтобы выделить наши вершины), или при запуске его из текстового редактора с помощью *Alt + P*, чтобы подсказать пользователю выбрать группу вершин для подсвечивания.

## Схема программы: AuraSpaceHandler.py

Следующая схема показывает, какие шаги мы предпримем в каждой ситуации:

1. Получить активный объект и меш.
2. Если запущено автономно:

- Получить список групп вершин
- Предложить на выбор
- Сохранить выбор как свойство объекта

### 3. Иначе:

- Получить свойство, которое содержит группу вершин
- Получить список координат вершин
- Для каждой вершины:
  - нарисовать маленький диск

Результирующий код доступен как *AuraSpaceHandler.py* в файле *scriptlinks.blend*:

```
# SPACEHANDLER.VIEW3D.DRAW
```

Он начинается со строки комментария, которая является существенной, так как она сигнализирует Блендеру, что это - скрипт обработчика пространства, который может быть связан с 3D-видом (в настоящее время никакую другую область нельзя связать с обработчиком пространства) и должен быть вызван по событию обновления изображения *redraw*.

```
import Blender
from Blender import *

scn = Scene.GetCurrent()
ob = scn.objects.active
if ob.type == 'Mesh':
    me = ob.getData(mesh = True)
    if Blender.bylink:
        p=ob.getProperty('Highlight')
        vlist = me.getVertsFromGroup(p.getData())
        matrix = ob.matrix
        drawAuras([me.verts[vi].co*matrix for vi in vlist],
                  p.getData())
    else:
        groups = ['Select vertexgroup to highlight%t']
        groups.extend(me.getVertGroupNames())
        result = Draw.PupMenu( '|'.join(groups) )
        if result>0:
```

```
try:
    p=ob.getProperty('Highlight')
    p.setData(groups[result])
except:
    ob.addProperty('Highlight',groups[result])
```

Далее скрипт приступает к извлечению активного объекта из текущей сцены и получает меш объекта, если его тип - *Mesh*. На выделенной строке мы проверяем, запущен ли скрипт как обработчик пространства, и если это так, мы выбираем свойство с именем *Highlight* (подсветка). Данные этого свойства является именем группы вершин, которую мы хотим подсветить. Мы продолжаем, получая список всех вершин в этой вершинной группе и получая матрицу объекта. Нам она нужна, поскольку позиции вершин загружены относительно матрицы объекта. Затем, мы создаем список позиций вершин и передаем его вместе с именем группы вершин в функцию *drawAuras()*, которая позаботится о фактическом рисовании.

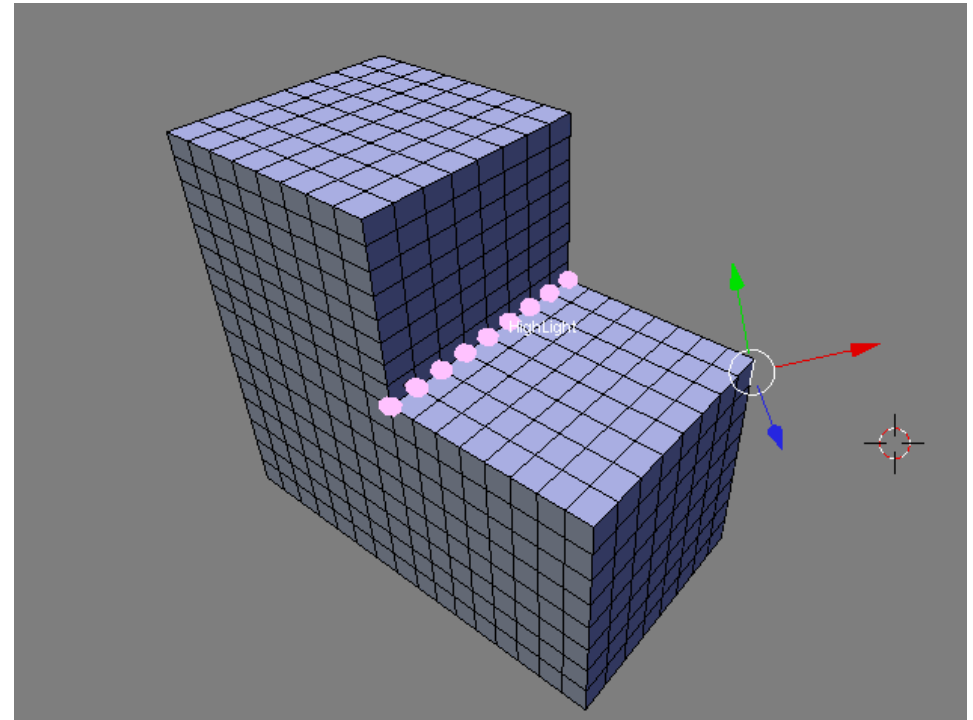
Вторая выделенная строка показывает начало кода, который будет выполняться, если мы запускаем скрипт из текстового редактора. Он создаёт строку, состоящую из имен всех групп вершин, связанных с активным объектом, разделенных символами трубы (|) и с добавленным подходящим названием. Эта строка передаётся в функцию *PupMenu()*, которая отобразит меню, и возвратит выбор пользователя, либо -1, если ничего не было выбрано.

Если была выбрана группа вершин, мы пытаемся извлечь свойство *Highlight*. Если это получается, мы записываем в данные этого свойства имя выбранной группы вершин. Если свойство еще не существовало, мы добавляем новое с именем *Highlight* и в качестве данных также присваиваем имя выбранной группы вершин.

Затем мы должны убедиться, что *скриптсвязи* включены (**окно Кнопка | панель Scripts | Scriptlinks**. Щелкните на **Enable Script Links**, если это еще не было сделано). Обратите внимание, что Блендеру все равно, имеем ли мы дело с обработчиками пространства или скриптсвязями, поскольку они включаются одинаково.

Последним шагом в использовании нашего обработчика пространства будет ассоциация его с 3D-видом. Чтобы сделать это,

включите галочку *Draw: AuraSpaceHandler.py* в меню **View - Space Handler Scripts** окна 3D-вида.



## Использование тем

Код, который мы еще не видели, имеет дело с фактическим рисованием подсветки и именем группы вершин, чтобы идентифицировать то, что мы выделяем. Он начинается с определения цвета, который мы используем для подсветки, и текста, извлекая их из текущей темы. Таким образом пользователь может настраивать эти цвета удобным способом из окна **Пользовательских настроек**:

```
theme      = Window.Theme.Get()[0]
textcolor  = [float(v)/255 for v in theme.get(
    Window.Types.VIEW3D ).text_hi[:3]]

color      = [float(v)/255 for v in
    theme.get(Window.Types.VIEW3D ).active[:3]]
```

В первой строке извлекается список **тем**, которые присутствуют. Первая из них является активной темой. Из этой темы мы извлекаем пространство *VIEW3D*, и его атрибут *text\_hi* является списком из четырех целых, представляющим цвет RGBA. Мы удаляем из списка альфа-компоненту и преобразуем его в список трех чисел с плавающей точкой (floats) в диапазоне [0, 1], которые мы используем как цвет нашего текста. Таким же образом мы создаем цвет подсветки из атрибута *active*.

Нашей следующей проблемой будет нарисовать подсветку в форме диска в специфическом месте. Так как размер диска совсем небольшой (его можно скорректировать изменением переменной *size*), мы можем аппроксимировать его достаточно хорошо формой восьмиугольника. Мы загружаем список координат *x* и *y* такого восьмиугольника в список *diskvertices*:

```
size=0.2
diskvertices=[( 0.0, 1.0), ( 0.7, 0.7),
               ( 1.0, 0.0), ( 0.7,-0.7),
               ( 0.0,-1.0), (-0.7,-0.7),
               (-1.0, 0.0), (-0.7, 0.7)]

def drawDisk(loc):
    BGL.glBegin(BGL.GL_POLYGON)
    for x,y in diskvertices:
        BGL.glVertex3f(loc[0]+x*size,loc[1]+y*size,loc[2])
    BGL.glEnd()
```

Само рисование восьмиугольника сильно зависит от функций, предоставляемых модулем Блендера *BGL* (выделено в предыдущем коде). Мы начинаем с установки режима рисования многоугольника, затем добавляем вершину для каждого кортежа в списке *diskvertices*. Позиция, переданная в функцию *drawDisk()*, будет центром, а вершины будут целиком лежать в круге с радиусом, равным размеру *size*. Когда мы вызываем функцию *glEnd()*, будет нарисован многоугольник, заполненный внутри текущим цветом.

Вы можете спросить, каким образом эти функции рисования знают, как перевести местоположение в 3D в координаты на экране, и тут есть действительно больше, чем кажется на первый взгляд, как мы увидим в следующей части кода. Необходимая функция, вызываемая для сообщения графической системе, как

преобразовать 3D-координаты в координаты экрана, не включена в функцию *drawDisk()* (в предшествующем куске кода). Дело в том, что вычисление этой информации отдельно для каждого диска должно привести к лишней потере в производительности, так как эта информация одинаковая для каждого диска, который мы рисуем.

Следовательно, мы определяем функцию *drawAuras()*, которая принимает список *locations* (позиции) и аргумент *groupname* (имя группы, строкового типа). Она вычислит параметры преобразования, вызовет *drawDisk()* для каждой позиции в списке, и, затем, добавит имя группы как на-экранную этикетку приблизительно справа от центра подсветки. Модуль Блендера *Window* предоставляет нам функцию *GetPerspMatrix()*, которая извлекает матрицу для правильного преобразования точки в пространстве 3D в точку на экране. Эта матрица размером 4x4 является объектом Питона, который должен быть преобразован в единственный список чисел с плавающей точкой, чтобы его могла использовать графическая система. Выделенные строки в следующем коде заботятся об этом. Следующие три строки сбрасывают режим проецирования и сообщают графической системе использовать нашу должным образом преобразованную перспективную матрицу для вычисления экранных координат. Заметьте, что изменение этих режимов проецирования и других настроек графики не влияет на то, как сам Блендер рисует объекты на экране, так как эти настройки сохраняются перед вызовом нашего скрипта обработчика и восстанавливаются впоследствии:

```
def drawAuras(locations,groupname):
    viewMatrix = Window.GetPerspMatrix()
    viewBuff = [viewMatrix[i][j] for i in xrange(4)
                for j in xrange(4)]
    viewBuff = BGL.Buffer(BGL.GL_FLOAT, 16, viewBuff)

    BGL.glLoadIdentity()
    BGL.glMatrixMode(BGL.GL_PROJECTION)
    BGL.glLoadMatrixf(viewBuff)

    BGL.glColor3f(*color)
    for loc in locations:
        drawDisk(loc)
    n=len(locations)
```

```

if n>0:
    BGL.glColor3f(*textcolor)
    x=sum([l[0] for l in locations])/n
    y=sum([l[1] for l in locations])/n
    z=sum([l[2] for l in locations])/n
    BGL.glRasterPos3f(x+2*size,y,z)
    Draw.Text(groupname,'small')

```

По окончании предварительных вычислений мы можем установить цвет, которым мы рисуем наши диски с помощью функции *glColor3f()*. Так как мы сохранили цвет в виде списка трех чисел с плавающей точкой, а функция *glColor3f()* принимает три отдельных аргумента, мы распаковываем этот список с помощью оператора звездочки. Затем, мы вызываем *drawDisk()* для каждого элемента в списке *locations*.

#### OpenGL функции Блендера:

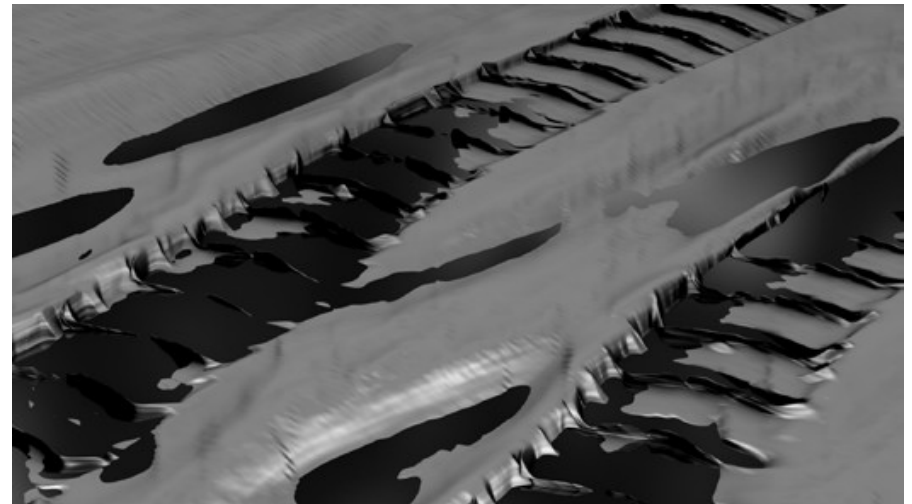
Документация по модулю Блендера BGL включает множество функций из библиотеки **OpenGL**. Многие из этих функций включены в большом количестве вариантов, которые выполняют одно и то же действие, но принимают свои аргументы различными способами. Например, *BGL.glRasterPos3f()* тесно связана с *BGL.glRasterPos3fv()*, которая принимает список трех чисел с плавающей точкой единичной точности вместо трех отдельных аргументов. За подробностями обратитесь к документации по API модулей *Blender.BGL* и *Blender.Draw* и к справочнику по OpenGL на <http://www.opengl.org/sdk/docs/man/>.

Если число подсветок, которые мы нарисовали, не нулевое, мы задаём цвет рисования в *textcolor* и затем вычисляем средние координаты всех подсвечиваемых вершин. Затем мы используем функцию *glRasterPos3f()*, чтобы установить стартовую позицию текста, который мы хотим отобразить в этих усреднённых координатах с небольшим пространством, добавленным к x-координате, чтобы немного сместить текст вправо. Затем функция Блендера *Draw.Text()* отобразит имя группы небольшим шрифтом в выбранной позиции.

## Снова о мешах — создание отпечатков

Хотя **мягкие тела (softbody)** и имитаторы **ткани (cloth)**, которые доступны в Блендере, во многих ситуациях делают свою работу отлично, иногда Вам может понадобиться иметь больше управления над процессом деформации меша, или Вы захотите симитировать какое-либо специфическое поведение, которое совсем не охвачено встроенными системами симуляции Блендера. Это упражнение показывает, как вычислять деформацию меша, которого коснулся, но не порвал другой меш. Оно не сможет быть физически точным. Мы стремимся к тому, чтобы получить вероятные результаты для твердых вещей, касающихся легко деформируемой или клейкой поверхности, например, палец, продавливающий масло, или колесо, едущее по мягкой обочине.

На рисунке ниже приведены несколько возможных отпечатков. Дорожки созданы анимированием катящейся автомобильной шины по подразделенной плоскости:



В следующей части мы обратимся к объекту, меш которого будет деформироваться в качестве исходного, и к объекту или объектам, делающим деформацию в качестве цели. В некотором смысле, это очень похоже на ограничение (constraint) и мы могли бы осуществить эти деформации как *ryconstraints*. Тем не менее, это не будет

исполнимым, поскольку ограничения оцениваются всякий раз, когда исходный меш или цели двигаются; этим самым вызывается интерфейс пользователя, что приведёт к мучительным остановкам, так как расчет пересечений и результирующей деформации мешей требует интенсивных вычислений. Следовательно, мы выбираем метод, где мы вычисляем и кешируем результаты всякий раз, когда сменяется кадр.

Наш скрипт предоставит несколько функций, он должен:

- Вычислить и кешировать деформации при каждом изменении кадра
- Изменить координаты вершин, когда присутствует кешированная информация

А при автономном запуске, скрипт должен:

- Сохранять и восстанавливать первоначальный меш
- Подсказывать пользователю возможные цели
- Ассоциировать себя как скриптсвязь с исходным объектом
- Возможно, удалять себя как скриптсвязь

Важное соображение в проектировании скрипта - как мы будем сохранять или кешировать оригинальный меш и промежуточные, деформированные меши. Поскольку мы не изменяем топологию меша (то есть, то, как вершины соединены друг с другом), а только координаты вершин, будет достаточно хранить только эти координаты. Это оставляет нас с вопросом: где сохранять эту информацию.

Если мы не хотим писать наше собственное устойчивое решение для сохранений, у нас есть два выбора:

- Использовать реестр Блендера
- Ассоциировать данные с исходным объектом в виде свойства

**Реестр (registry)** Блендера легко использовать, но мы должны иметь какой-то метод ассоциации данных с объектом, поскольку, возможно, пользователь захочет соединить больше, чем один объект с вычислителем отпечатков. Мы могли бы использовать имя объекта как ключ, но если пользователь захочет изменить это имя, мы

потеряем ссылку на сохранённую информацию, в то время как функциональность скриптсвязи должна там все еще оставаться. Тогда пользователь сам должен стать ответственным за удаление сохранённых данных, если имя объекта было изменено.

Ассоциация всех данных в виде **свойства (property)** избавит от страданий, вызванных переименованиями, и данные будут очищаться при удалении объекта, но типы данных, которые можно сохранять в свойствах, ограничены целым, действительным с плавающей точкой, или строкой. Существуют способы преобразования произвольных данных в строки, используя стандартный модуль Питона *pickle*, но, к несчастью, такому сценарию препятствуют две проблемы:

- Координаты вершин в Блендере - экземпляры объекта *Vector*, а они не поддерживают протокол *pickle*
- Размер строкового **свойства** ограничен 127 символами, и этого слишком мало, чтобы сохранить даже один кадр с координатами вершин для меша средних размеров

Несмотря на недостатки использования реестра, мы будем использовать его для разработки двух функций - одну для сохранения координат вершин для данного номера кадра, и одну для извлечения этих данных и применения их к вершинам меша. Сначала мы определяем вспомогательную функцию *skey()*, которая возвращает ключ для использования с функциями реестра, исходя из имени объекта, чьи данные меша мы хотим кешировать:

```
def skey(ob):  
    return meshcache+ob.name
```

### Не все реестры - одно и то же



Не перепутайте реестр Блендера с реестром Windows. Оба предназначены для аналогичных целей - обеспечить устойчивую память для всех типов данных, но это разные объекты. Фактические данные в реестре Блендера, которые записаны на диск, по умолчанию находятся в каталоге *.blender/scripts/bpydata/config/*, и это местоположение может быть изменено заданием параметра *datadir* с помощью *Blender.Set()*.

Наша функция *storemesh()* принимает в качестве аргументов объект и номер кадра. Первым действием нужно извлечь координаты вершин из данных меша, связанных с объектом. Затем она извлекает все данные, сохранённые в реестре Блендера для объекта, с которым мы имеем дело, и мы передаем дополнительный параметр *True* (Истина), указывающий, что если нет данных в памяти, *GetKey()* должна проверить их наличие на диске. Если совсем нет никаких данных, сохранённых для нашего объекта, *GetKey()* возвращает *None*, и в этом случае мы инициализируем наш кеш пустым словарём.

Впоследствии, мы сохраняем координаты нашего меша в этом словаре, проиндексированном номером кадра (выделено в следующем куске кода). Мы преобразуем этот номер кадра из целого в строку, которую нужно использовать в качестве фактического ключа, поскольку функция Блендера *SetKey()* принимает ключи строкового типа при сохранении данных реестра на диск, и вызовет исключение, если она сталкивается с целым. Последняя строка снова вызывает *SetKey()* с дополнительным аргументом *True*, чтобы указать, что мы хотим сохранять данные также на диск.

```
def storemesh(ob, frame):
    coords = [(v.co.x,v.co.y,v.co.z) for v in
               ob.getData().verts]
    d=Blender.Registry.GetKey(ckey(ob), True)
    if d == None: d={}
    d[str(frame)]=coords
    Blender.Registry.SetKey(ckey(ob), d, True)
```

Функция *retrievemesh()* принимает в качестве аргументов объект и номер кадра. Если она находит кешированные данные для данного объекта и кадра, она назначает загруженные координаты вершинам в меше. Сначала мы определим два новых исключения, означающие некоторые специфические ошибочные состояния, с которыми *retrievemesh()* может столкнуться:

```
class NoSuchProperty(RuntimeError): pass;
class NoFrameCached(RuntimeError): pass;
```

*retrievemesh()* вызовет исключение *NoSuchProperty*, если объект не имеет связанных кешированных данных меша, и исключение *NoFrameCached* если данные присутствуют, но не для указанного кадра. Выделенная строка в следующем коде заслуживает

некоторого внимания. Мы выбираем связанные данные меша у объекта с *mesh=True*. Это даст завернутый (wrapped) меш, а не копию, так что любые данные вершин, к которым мы получаем доступ, или изменяем, ссылаются на фактические данные. Также, мы сталкиваемся со встроенной функцией Питона *zip()*, которая принимает два списка и возвращает список, состоящий из кортежей двух элементов, по одному из каждого списка. Это эффективно позволяет просматривать два списка параллельно. В нашем случае, эти списки - список вершин и список координат и мы просто преобразуем эти координаты в векторы и назначаем их в атрибут *co* каждой вершины:

```
def retrievemesh(ob, frame):
    d=Blender.Registry.GetKey(ckey(ob), True)
    if d == None:
        raise NoSuchProperty("no property %s for object %s"
                               %(meshcache, ob.name))

    try:
        coords = d[str(frame)]
    except KeyError:
        raise NoFrameCached(("frame %d not cached on" +
                              "object %s") %(frame, ob.name))

    for v,c in zip(ob.getData(mesh=True).verts, coords):
        v.co = Blender.Mathutils.Vector(c)
```

Чтобы завершить наш набор функций кеша, мы определяем функцию *clearcache()*, которая пытается удалять данные в реестре, связанные с нашим объектом. Конструкция *try ... except ...* обеспечивает, чтобы при отсутствии сохранённых данных, действие было молча проигнорировано:

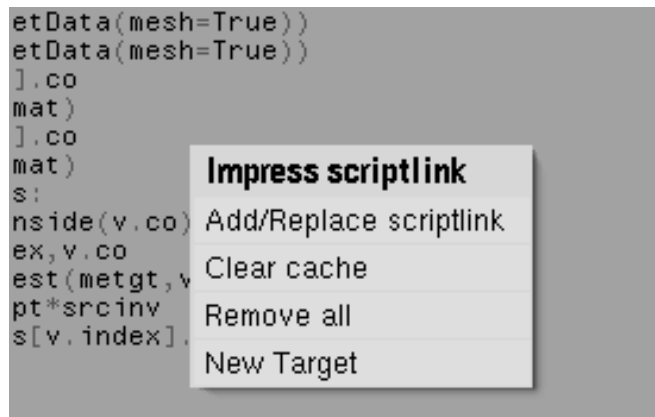
```
def clearcache(ob):
    try:
        Blender.Registry.RemoveKey(ckey(ob))
    except:
        pass
```

## Пользовательский интерфейс

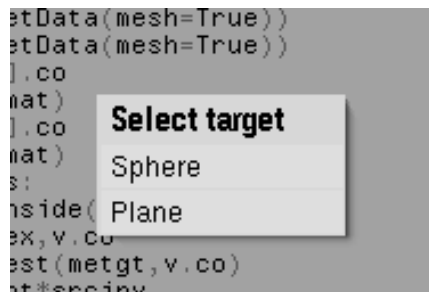
Наш скрипт будет использоваться не только как скриптсвязь, связанная с объектом, но он также будет использоваться автономно (по нажатию *Alt + P* в текстовом редакторе, например), чтобы обеспечить пользователя средствами для идентификации цели,



которая создаст отпечаток, чтобы очищать кеш, и, чтобы ассоциировать скриптсвязь с активным объектом. При использовании таким образом, он обеспечивает конечного пользователя несколькими управляющими меню, показанными на скриншотах. Первый показывает возможные действия:



Второй скриншот показывает всплывающее меню с предложением выбрать объект из списка *Меш*-объектов, из которого пользователь может выбрать, чем создавать отпечаток:



Мы сначала определяем вспомогательную функцию, которая будет использована выпадающим меню, обеспечивающим пользователя выбором *Меш*-объектов, для использования в качестве цели при создании отпечатка. *getmeshobjects()* принимает аргумент *scene* и возвращает список имен всех *Меш*-объектов. Как показано на скриншоте, список объектов-целей включает в том числе исходный объект. Хотя это законно, но вряд ли очень полезно:

```
def getmeshobjects(scene):  
    return [ob.name for ob in scene.objects if  
            ob.type=='Mesh']
```

Само меню осуществляется функцией *targetmenu()*, определенной следующим образом:

```
def targetmenu(ob):  
    meshobjects=getmeshobjects(Blender.Scene.GetCurrent())  
    menu='Select target%t|'+ "|".join(meshobjects)  
    ret = Blender.Draw.PupMenu(menu)  
    if ret>0:  
        try:  
            p = ob.getProperty(impresstarget)  
            p.setData(meshobjects[ret-1])  
        except:  
            ob.addProperty(impresstarget,meshobjects[ret-1])
```

Она выбирает список из всех *Меш*-объектов в сцене и представляет этот список на выбор пользователю, используя функцию Блендера *Draw.PupMenu()*. Если пользователь выбирает один из пунктов меню (возвращенная величина будет больше нуля, смотри выделенную строку предыдущего кода), будет загружено имя этого *Меш*-объекта как свойство, связанное с нашим объектом. Константа *impresstarget* определяется в другом месте в качестве имени для свойства. Сначала, программа независимо проверяет, есть ли там уже такое свойство, связанное с объектом, вызывая метод *getProperty()*, и присваивает данные свойству, если есть. Если метод *getProperty()* вызывает исключение, поскольку свойство еще не существует, тогда мы добавляем новое свойство к объекту и назначаем ему данные с помощью единственного вызова *addProperty()*.

Основной интерфейс пользователя определяется на верхнем уровне скрипта. Он проверяет, что не был запущен как скриптсвязь, затем предоставляет пользователю несколько действий на выбор:

```
if not Blender.bylink:  
    ret = Blender.Draw.PupMenu('Impress scriptlink%t|'+  
                              'Add/Replace scriptlink|'+  
                              'Clear cache|Remove ' +  
                              'all|New Target')  
    active = Blender.Scene.GetCurrent().objects.active  
    if ret > 0:  
        clearcache(active)  
    if ret== 1:  
        active.clearScriptLinks([scriptname])  
        active.addScriptLink(scriptname,'FrameChanged')
```



```

    targetmenu(active)
elif ret== 2:
    pass
elif ret== 3:
    active.removeProperty(meshcache)
    active.clearScriptLinks([scriptname])
elif ret== 4:
    targetmenu(active)

```

Любой правильный выбор очистит кеш (выделено), и в последующих проверках выполняются необходимые действия, связанные каждым индивидуальным выбором: **Add/Replace scriptlink** удалит скриптсвязь, если она уже присутствует, чтобы предотвратить появление дубликатов и, затем, добавит её к активному объекту. Затем будет представлено меню целей, чтобы выбрать Меш-объект для использования при создании отпечатка. Так как мы уже очистили кеш, то второй выбор, **Clear cache**, не делает ничего специфического, так что здесь у нас просто оператор *pass* (пропустить). **Remove All** попытается удалить кеш и отсоединить себя как скриптсвязь, и, наконец, меню **New target** представит меню выбора целей, чтобы дать возможность пользователю выбрать новый целевой объект, не удаляя никаких кешированных результатов.

Если скрипт выполняется как скриптсвязь, мы сначала проверяем, что мы действуем при событии *FrameChanged*, затем пытаемся извлечь любые сохраненные координаты вершин для текущего кадра (выделено в следующем коде). Если нет ранее загруженных данных, мы должны вычислить эффекты от целевого объекта для этого кадра. Следовательно, мы получаем список целевых объектов для рассматриваемого объекта, вызывая вспомогательную функцию *gettargetobjects()* (на данный момент будет возвращен список только из одного объекта), и для каждого объекта мы вычисляем эффект для нашего меша с помощью вызова *impress()*. Затем, мы сохраняем эти, возможно изменённые, координаты вершин и корректируем дисплейный список, чтобы графический интерфейс пользователя Блендера знал, как отображать наш измененный меш:

```

elif Blender.event == 'FrameChanged':
    try:
        retrievemesh(Blender.link,Blender.Get('curframe'))
    except Exception as e: # мы ловим что-нибудь
        objects = gettargetobjects(Blender.link)
        for ob in objects:
            impress(Blender.link,ob)
        storemesh(Blender.link,Blender.Get('curframe'))
        Blender.link.makeDisplayList()

```

Теперь нам осталось фактическое вычисление отпечатка целевого объекта в нашем меше.

## Вычисление отпечатка

Определение эффекта создания отпечатка от целевого объекта будем достигать следующим образом:

Для каждой вершины в меше, получающем отпечаток:

1. Определить, расположена ли она внутри целевого объекта, и если это так:
2. Установить позицию вершины в позицию ближайшей вершины на объекте, создающем отпечаток

Здесь есть несколько важных вопросов. Позиция вершины в меше сохранена относительно матрицы преобразования объекта. Другими словами, если мы хотим сравнить координаты вершин в двух разных мешах, мы должны преобразовать каждую вершину матрицами преобразования их соответствующих объектов перед выполнением любого сравнения.

Также, объект *Blender.Mesh* имеет метод *pointInside()*, который возвращает Истину, если данная точка находится внутри меша. Тем не менее, он будет работать только в надежно закрытых мешах, так что пользователь должен проверить, что объекты, которые создают отпечатки на самом деле закрыты. (Они могут иметь внутренние пузыри, но их поверхности не должны содержать рёбер, которые не примыкают в точности к 2 граням. Эти так называемые non-manifold рёбра можно выбрать в режиме *выбора рёбер* с помощью **Select | Non Manifold** в 3D-виде или нажав *Ctrl + Shift + Alt + M.*)

Наконец, перемещение вершин к ближайшей вершине на целевом объекте может быть совсем неточным, если целевой меш довольно грубый. Производительность разумная, тем не менее, было бы хорошо иметь для сравнения несколько точек, так как наш алгоритм - довольно неумелый, поскольку сначала определяет точку внутри меша, и затем отдельно вычисляет ближайшую вершину, дублируя множество вычислений. Тем не менее, так как производительность приемлема даже для мешей, состоящих из сотен точек, мы будем придерживаться нашего подхода, поскольку он сохраняет нашу программу простой и спасает нас от необходимости писать и тестировать очень сложный код.

Реализация начинается с функции, возвращающей расстояние до ближайшей вершины к данной точке `pt` и её координаты:

```
def closest(me,pt):
    min = None
    vm = None
    for v in me.verts:
        d=(v.co-pt).length
        if min == None or d<min:
            min = d
            vm = v.co
    return min,vm
```

Функция *impress()* принимает исходный и целевой объект как аргументы и модифицирует меш-данные исходного объекта, если целевой меш делает отпечаток. Первая вещь, которую она делает - извлечение матриц преобразования объектов. Как указано ранее, они будут нужны для преобразования координат вершин, чтобы их можно было сравнивать. Мы также извлекаем обратную матрицу исходного объекта. Она будет нужна, чтобы преобразовать координаты в пространство исходного объекта.

Выделенная строка извлекает завернутые (*wrapped*) меш-данные исходного объекта. Нам нужны завернутые данные, поскольку нам может понадобиться изменить координаты некоторых вершин. Следующие две строки извлекают копии меш-данных. Нам нужны эти копии, чтобы преобразование, которое мы выполним, не повлияло на фактические меш-данные. Вместо копирования мы могли бы пропустить аргумент *mesh=True*, что должно было бы дать нам ссылку на объект *Nmesh* вместо объекта *Mesh*. Тем не менее,

объекты *Nmesh* не завернуты и обозначены как устаревшие. Также, у них отсутствует метод *pointInside()*, который нам нужен, так что мы выбираем самостоятельное копирование мешей.

Затем, мы преобразуем эти копии мешей соответствующими их объектам матрицами преобразования. Использование метода этих мешей *transform()* спасает нас от цикла по каждой вершине и самостоятельного умножения координат вершин на матрицу преобразования, и этот метод, наверное, несколько быстрее, так как *transform()* полностью выполнен на языке C:

```
from copy import copy

def impress(source,target):
    srcmat=source.getMatrix()
    srcinv=source.getInverseMatrix()
    tgtmat=target.getMatrix()

    orgsrc=source.getData(mesh=True)
    mesrc=copy(source.getData(mesh=True))
    metgt=copy(target.getData(mesh=True))

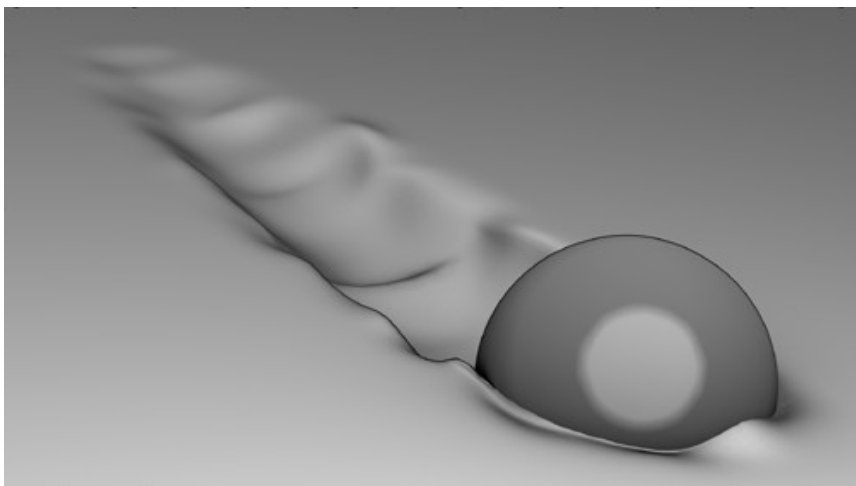
    mesrc.transform(srcmat)
    metgt.transform(tgtmat)

    for v in mesrc.verts:
        if metgt.pointInside(v.co):
            d,pt = closest(metgt,v.co)
            orgsrc.verts[v.index].co=pt*srcinv
```

Последняя часть функции *impress()* - это цикл по всем вершинам в преобразованном исходном меше и проверка на нахождение вершины внутри (преобразованного) целевого меша. Если это так, функция определяет, какая вершина в целевом меше ближайшая, и устанавливает задетую вершину в оригинальном меше в эти координаты.

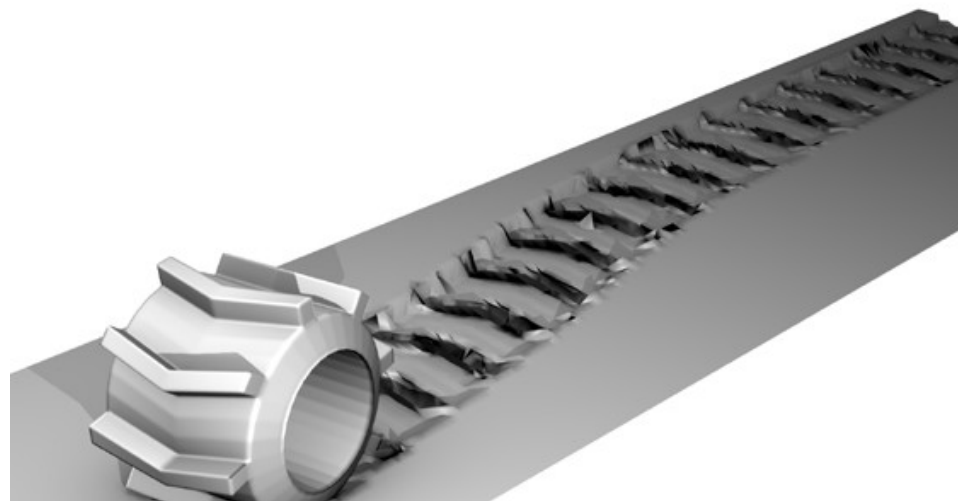
Этот оригинальный меш не преобразован, так что мы должны преобразовать эту ближайшую точку в пространство исходного объекта, умножая координаты на обратную матрицу преобразования. Поскольку вычисления преобразования являются дорогостоящими, модификация преобразованного меша и преобразование всего меша обратно в конечном счете может взять деликатное время.

Содержание ссылки на не преобразованный меш и просто преобразование обратно отдельных точек может, следовательно, оказаться предпочтительным, если только сравнительно немного вершин попали в отпечаток. Полный скрипт доступен как *ImpressScriptLink.py* в файле *scriptlinks.blend*. Следующая иллюстрация показывает возможный результат. Здесь мы создали небольшую анимацию шара (icosphere), прокатив его по грязи (подразделенная плоскость) и погружая в неё.



При работе со скриптом важно иметь в виду, что когда отпечаток вычисляется, ни одна из вершин меша, который получает отпечаток, не должна быть расположена внутри цели раньше, чем она сдвинется. Если это случилось, то, возможно, вершина будет сметена вслед за перемещением цели, исказив исходный меш вдоль пути. Например, чтобы сделать иллюстрацию колесного следа в грязи, мы анимируем катящееся колесо вдоль пути, рассчитывая отпечатки, которые оно делает в каждом кадре. В первом кадре, который мы анимируем, мы должны убедиться, что колесо не касается плоскости земли, иначе она может быть искажена, поскольку, если вершина плоскости земли оказалась внутри колеса и близко ко внутреннему ободу, она будет перемещена на ближайшую вершину на этом ободу. Если колесо катится медленно, эта вершина останется близко к этому внутреннему ободу, и тем самым фактически будет клеиться к

этому движущемуся внутреннему ободу, разрывая в процессе плоскость земли. Тот же разрушительный процесс может произойти, если целевой объект очень маленький по сравнению с исходным мешем или перемещается очень быстро. В этих обстоятельствах, вершина может проникнуть в целевой объект так быстро, что ближайшая вершина не будет находиться на ведущей поверхности, создающей отпечаток, но окажется где-нибудь в другом месте цели, и в результате вершины будут притянуты ко внешней стороне вместо проталкивания внутрь. В иллюстрации катящейся шины трактора, мы тщательно спозиционировали шину в первом кадре, чтобы она находилась строго справа от подразделенной плоскости перед ключевым кадром, начинающим катящееся движение влево. Показанное изображение взято из кадра 171 без какого-либо сглаживания или применённых материалов к плоскости.



## **Итог**

В этой главе мы узнали как привязывать изменения к развитию кадров анимации и как обращаться к информации о состоянии объекта. Мы также видели как сменять слои, например, чтобы объект на рендере стал невидимым. В подробностях, мы увидели:

- Что такое скрипты-связи и обработчики пространства
- Как выполнять действия при каждом изменении кадров в анимации
- Как ассоциировать дополнительную информацию с объектом
- Как заставить объект появляться или исчезать изменением слоя или изменением прозрачности
- Как осуществить схему, соединяющую разные меши с объектом в каждом кадре
- Как расширить функциональность 3D-вида

Далее: добавление ключей формы и кривых IPO.

## Ключи формы, кривые IPO, и Позы

Мы уже сталкивались с кривыми IPO в *Главе 4, Pydrivers and Constraints*, когда мы обсуждали Pydrivers, но с **IPO** можно делать гораздо больше, чем просто управлять одним IPO посредством другого. Например, API Блендера обеспечивает нас средствами для создания IPO из ничего, что позволяет определить движение, которое не легко воссоздать, устанавливая ключевые кадры вручную. Кроме того, некоторые типы кривых IPO имеют отчасти отличающееся поведение по сравнению с теми, с которыми мы до сих пор сталкивались. **Ключи Формы** и **Позы** - примеры (наборов) кривых IPO, которые отличаются от, например, IPO позиции. Мы столкнемся как с ключами формы, так и Позами позже в этой главе, но мы начнём с рассмотрения того, как мы можем определять IPO из ничего.

В этой главе мы изучим, как:

- определять кривые IPO
- определять ключи формы у меша
- определять IPO для этих ключей формы
- позировать арматуры
- группировать изменения поз в действия

### **Обидчивый субъект - определение IPO из ничего**

Множество путей движения объектов трудно моделировать вручную, например, когда мы хотим, чтобы объект следовал в точности по математической кривой, или если мы хотим скоординировать перемещение многочисленных объектов по некоторому пути, что не легко выполнить копированием кривых IPO или определением Управляющих объектов IPO (drivers).

Представьте себе следующий сценарий: мы хотим произвести взаимнообмен позиций некоторых объектов в течении определённого времени плавным путём без того, чтобы объекты проходили сквозь друг друга в середине, и даже не касались друг друга. Наверное, это можно выполнить настройкой ключей вручную, но в тоже время довольно обременительно, особенно если нам может понадобиться повторить это для нескольких наборов объектов. Скрипт, который мы разработаем, заботится о всех этих деталях и может быть применим к любым двум объектам.

### **Схема программы: orbit.py**

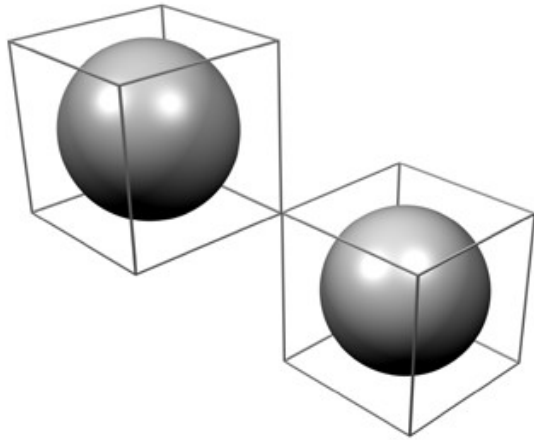
Скрипт *orbit.py*, который мы разработаем, будет предпринимать следующие шаги:

1. Определение точки середины пути между выбранными объектами.
2. Определение протяженности выбранных объектов.
3. Определение IPO для объекта один.
4. Определение IPO для объекта два.

Определение точки середины пути между выбранными объектами будет достаточно легко: мы просто возьмем среднее позиций обоих объектов. Определение протяженности выбранных объектов будет всё-таки некоторым вызовом. Объект может иметь неправильную форму, и определение кратчайшего расстояния с учётом любого возможного поворота объектов на пути трудно для вычисления. К счастью, мы можем сделать разумное приближение, так как каждый объект имеет связанный с ним **габаритный ящик (bounding box)**.

Этот габаритный ящик является прямоугольным параллелепипедом, который просто включает в себя все точки объекта. Если взять половину диагонали в качестве протяженности

(размера) объекта, то легко видеть, что это расстояние может быть гораздо больше того, насколько близко мы можем быть к другому объекту, не касаясь его, в зависимости от точной формы объекта. Но это гарантирует, что мы никогда не окажемся слишком близко. Этот габаритный ящик легко получить из объектного метода *getBoundingBox()* в виде списка из восьми векторов, каждый из которых представляет один из углов габаритного ящика. Понятие проиллюстрировано на следующем рисунке, где показаны габаритные ящики двух сфер:



Длина диагонали габаритного ящика может быть вычислена определением как максимального так и минимального значений для каждой из координат x, y, и z. Компоненты вектора, представляющего эту диагональ, являются разностями между этими максимумом и минимумом. Длина диагонали впоследствии получается взятием квадратного корня суммы квадратов компонент x, y, и z. Функция *diagonal()* довольно кратко реализована, так как она использует много встроенных функций Питона. Она принимает список векторов в виде аргумента, а затем проходит циклом по каждой из компонент (выделено. Компоненты x, y, и z объекта *Vector* в Блендере могут быть доступны по индексам 0, 1, и 2 соответственно):

```
def diagonal(bb):
    maxco=[]
    minco=[]
    for i in range(3):
        maxco.append(max(b[i] for b in bb))
```

```
        minco.append(min(b[i] for b in bb))
    return sqrt(sum((a-b)**2 for a,b in zip(maxco,minco)))
```

Она определяет пределы для каждой компоненты, используя встроенные функции *max()* и *min()*. В конце она возвращает длину, спаривая каждый минимум и максимум с помощью функции *zip()*.

Следующим шагом нужно проверить, что мы имеем в точности два выбранных объекта, и если это не так, сообщить пользователю, отображая всплывающее меню (выделено в следующем куске кода). Если у нас есть два выбранных объекта, мы извлекаем их позиции и габаритные ящики. Затем мы вычисляем максимальное расстояние *w*, на которое каждый объект должен отклониться от своего пути, оно должно быть половиной минимального расстояния между ними, что эквивалентно четверти суммы длин диагоналей этих объектов:

```
obs=Blender.Scene.GetCurrent().objects.selected

if len(obs)!=2:
    Draw.PupMenu('Please select 2 objects%t|Ok')
else:
    loc0 = obs[0].getLocation()
    loc1 = obs[1].getLocation()

    bb0 = obs[0].getBoundingBox()
    bb1 = obs[1].getBoundingBox()

    w = (diagonal(bb0)+diagonal(bb1))/4.0
```

Прежде, чем мы сможем вычислить траектории обоих объектов, мы сначала создадим два новых и пустых объекта кривых IPO:

```
ipo0 = Ipo.New('Object','ObjectIpo0')
ipo1 = Ipo.New('Object','ObjectIpo1')
```

Мы произвольно выбираем начальный и конечный кадры нашей операции обмена в 1 и 30 соответственно, но скрипт легко может быть адаптирован для того, чтобы пользователь вводил эти величины. Мы итерируем по каждой отдельной кривой IPO для IPO местоположения и создаем первую точку (или ключевой кадр) и этим самым фактически назначается кортеж (номер кадра, значение) на кривую (выделенные строки следующего кода). Последующие точки могут быть добавлены к этим кривым по индексу - их номеру кадра присвоением значения, как это сделано для кадра 30 в следующем коде:

```

for i,icu in enumerate((Ipo.OB_LOCX,
                        Ipo.OB_LOCY,Ipo.OB_LOCZ)):
    ipo0[icu]=(1,loc0[i])
    ipo0[icu][30]=loc1[i]

    ipo1[icu]=(1,loc1[i])
    ipo1[icu][30]=loc0[i]

    ipo0[icu].interpolation =
IpoCurve.InterpTypes.BEZIER
    ipo1[icu].interpolation =
IpoCurve.InterpTypes.BEZIER

```

Заметьте, что ключ позиции первого объекта в кадре 1 является текущей позицией, а ключ позиции в кадре 30 - позиция второго объекта. Для другого объекта всё с точностью до наоборот. Мы установили режимы интерполяции этих кривых на "Bezier", чтобы получить плавное движение. У нас теперь есть две кривые IPO, которые делают взаимообмен позиций двух объектов, но расчёт пока таков, что они будут двигаться сквозь друг друга.

Следовательно, нашим следующим шагом нужно добавить ключ в кадре 15 со скорректированной компонентой z. Ранее мы вычислили w, чтобы запомнить половину расстояния, на которое нужно сдвинуться каждому с пути другого. Здесь мы добавляем это расстояние к компоненте z в середине пути для первого объекта и вычитаем его для другого:

```

mid_z = (loc0[2]+loc1[2])/2.0
ipo0[Ipo.OB_LOCZ][15] = mid_z + w
ipo1[Ipo.OB_LOCZ][15] = mid_z - w

```

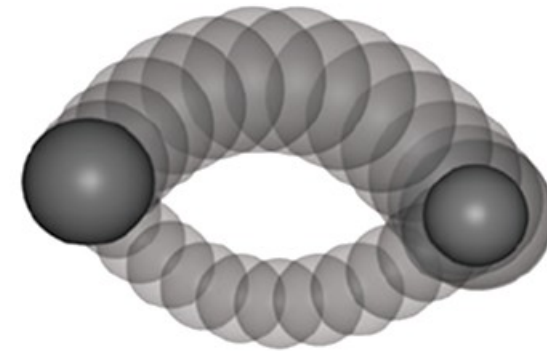
Наконец, мы добавляем новые кривые IPO к нашим объектам:

```

obs[0].setIpo(ipo0)
obs[1].setIpo(ipo1)

```

Полный код доступен как *swap2.py* в файле *orbit.blend*. Результирующие пути двух объектов схематически отображены на следующем скриншоте:



## Много проглотил - определение поз

Часто мультяшные персонажи, как кажется, имеют трудности, пытаясь поглощать свою пищу, и даже когда они наслаждаются своим обедом, скорее всего будут вынуждены пропускать его через слишком узкую глотку, чтобы он проходил удобно без всяких видимых изменений формы.

Трудно анимировать проглатывание или любое другое **перистальтическое движение**, используя ключи формы, так как это не общая форма меша, которая изменяется однородным способом: мы хотим двигать вдоль него локализованную деформацию. Один из способов сделать это - соединить арматуру, состоящую из линейной цепи костей с мешом, который мы хотим деформировать (показано на иллюстрации) и анимировать масштаб каждой индивидуальной кости во времени. Этим путём мы можем значительно расширить возможность управлять перемещением 'куска' внутри. Например, возможно сделать перемещение работающим слегка с перебоями, как будто он перемещается от кости к кости, чтобы имитировать поглощение чего-то что жесткого.





(ПЕРИСТАЛЬТИКА (peristalsis) - волнообразные сокращения, распространяющиеся вдоль некоторых полых органов в теле человека. Эти сокращения возникают самопроизвольно и характерны для таких полых органов, которые снабжены круговыми и продольными мышцами (например, обычно они наблюдаются в кишечнике). Перистальтика усиливается благодаря растяжению стенок полого органа. Как только стенки органа растягиваются, происходит сокращение круговых мышц. Перед их растяжением происходит расслабление круговых мышц и сокращение продольных, благодаря чему содержимое органа (чаще всего, кишечника) продвигается в дистальном направлении. - с сайта <http://vocabulary.ru> — пожелание приятного аппетита от переводчика ☺)

Для того, чтобы синхронизировать масштабирование индивидуальных костей таким образом, чтобы оно следовало по цепочке от родителя к ребенку, мы должны отсортировать наши кости, поскольку атрибут `bones` объекта `Pose`, который мы получаем, вызвав `getPose()` в арматуре, является словарём. Цикл по ключам или значениям этого словаря будет возвращать эти величины в произвольном порядке.

Следовательно, мы определяем функцию `sort_by_parent()`, которая принимает список костей Позы `pbones` и возвращает список строк, каждая из которых является именем кости Позы. Список будет отсортирован так, чтобы родитель был первым пунктом, со следующими за ним его детьми. Очевидно, что такая функция не будет возвращать значимый список для арматур, которые имеют кости с более чем одним ребенком, но для нашей линейной цепи костей это работает прекрасно.

В следующем коде, мы используем список имен `bones`, чтобы хранить имена костей Позы в правильном порядке. **Мы выталкиваем (pop) кость из списка костей Позы `pbones` и добавляем её имя достаточно долго, пока она ещё не добавлена (выделено).** (Я не смог адекватно перевести это предложение, потому что здесь логика и программы и текста, как мне кажется, дают сбой, подробнее смотрите ниже — прим. пер.) Мы сравниваем имена вместо объектов костей Позы, поскольку текущая реализация костей Позы надежно не поддерживает оператора `in`:

```
def sort_by_parent(pbones):
    bones=[]
    if len(pbones)<1 : return bones
    bone = pbones.pop(0)
    while(not bone.name in bones):
        bones.append(bone.name)
```

Затем, мы получаем родителя кости, которую мы только что добавили к нашему списку, и настолько долго, насколько мы можем просматривать цепь родителей, мы включаем такого родителя (или, точнее, его имя) в наш список перед текущим элементом (выделено ниже). Если цепь не может следовать дальше, мы выталкиваем новую кость Позы. Когда больше нет костей, метод `pop()` вызовет исключение `IndexError`, и мы выходим из нашего цикла `while`:

```
parent = bone.parent
while(parent):
    if not parent.name in bones:
        bones.insert(bones.index(bone.name) ,
                    parent.name)

    bone = parent
    parent = parent.parent
try:
    bone = pbones.pop(0)
except IndexError:
    break
return bones
```

---

Чем дольше я пытался разобраться с логикой этой функции, чтобы адекватно перевести два предыдущих абзаца, тем сильнее мне это не нравилось, ибо логики я не наблюдал. Тогда я немного потестировал эту функцию в файле `peristaltic.blend`, и убедился, что она правильно работает не во всех случаях. Цепочка костей в файле по направлению от родительских к дочерним выглядит так: `['Bone', 'Bone.001', 'Bone.002', 'Bone.003', 'Bone.004', 'Bone.005']`. Если на вход функции список `pbones` приходит в таком порядке: `['Bone.001', 'Bone.002', 'Bone.003', 'Bone.004', 'Bone.005', 'Bone']`, то результат получается таким, каким надо, но если на вход придёт, например, список `['Bone.002', 'Bone.001', 'Bone.003', 'Bone.004', 'Bone.005', 'Bone']` (первые два элемента поменяны

местами), то на выходе будет всего 3 кости: ['Bone', 'Bone.001', 'Bone.002']. Вот мой исправленный вариант функции:

```
def sort_by_parent(pbones):
    bones=[]
    while True: # Бесконечный цикл гарантирует перебор
                  # всех костей из входного списка
        try:
            bone = pbones.pop(0)
        except IndexError:
            break # Единственное условие выхода из цикла
        if not bone.name in bones:
            bones.append(bone.name)
            parent = bone.parent
            while(parent):
                if not parent.name in bones:
                    bones.insert(bones.index(bone.name),
                                parent.name)

                bone = parent
                parent = parent.parent
    return bones
```

- Добавление переводчика.

---

Следующий шаг - это определение самого скрипта. Сначала, мы получаем активный объект в текущей сцене и проверяем, что это - на самом деле арматура. Если нет, мы предупреждаем об этом пользователя с помощью всплывающего сообщения (выделенная часть следующего кода), в противном случае мы продолжаем и получаем связанные с арматурой данные методом *getData()*:

```
scn = Blender.Scene.GetCurrent()
arm = scn.objects.active
if arm.getType() != 'Armature':
    Blender.Draw.PupMenu("Selected object is not an " +
                          "Armature%t|Ok")
else:
    adata = arm.getData()
```

Затем, мы делаем арматуру редактируемой и убеждаемся, что у каждой кости задана опция *HINGE* (выделено). Преобразование списка опций в множество (set) и обратно в список после добавления

опций *HINGE* является способом удостовериться, что эта опция появится в списке только один раз.

```
adata.makeEditable()
for ebone in adata.bones.values():
    ebone.options =
        list(set(ebone.options) |
              set([Blender.Armature.HINGE]))
adata.update()
```

Поза связана с объектом арматуры, а не со своими данными, так что мы получаем её из объекта *arm*, используя метод *getPose()*. Позы кости очень похожи на обычные IPO, но они должны быть связаны с **действием (action)**, которое группирует эти позы. При работе с Блендером интерактивно действие создаётся автоматически, как только мы вставим ключевой кадр в позу, но в скрипте мы должны явно создать действие, если оно ещё не присутствует (выделено):

```
pose = arm.getPose()
action = arm.getAction()
if not action:
    action = Blender.Armature.NLA.NewAction()
    action.setActive(arm)
```

Следующим шагом нужно отсортировать кости Позы в порядке цепи от родительских к дочерним, используя нашу ранее определенную функцию. Всё, что осталось сделать, это двигаться по временной шкале через десять кадров за 1 шаг и задавать ключи для масштаба каждой кости на каждом шаге, увеличивая масштаб, если номер кости в последовательности соответствует нашему шагу и восстанавливая его, если нет. Одна из результирующих кривых IPO показана на скриншоте. Заметьте, что нашей предварительной установкой атрибута *HINGE* в каждой кости, мы предотвратили распространение масштабирования на детей кости:

```
bones = sort_by_parent(pose.bones.values())

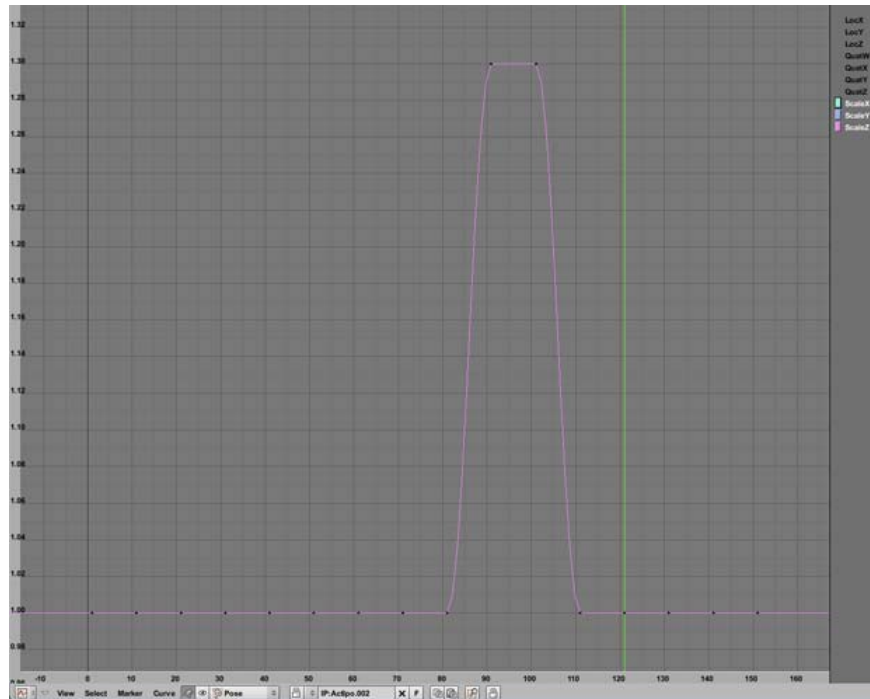
for frame in range(1,161,10):
    index = int(frame/21)-1
    n = len(bones)
    for i,bone in enumerate(bones):
        if i == index :
            size = 1.3
```

```

else :
    size = 1.0
pose.bones[bone].size=Vector(size,size,size)
pose.bones[bone].insertKey(arm,frame,
                           Blender.Object.Pose.SIZE)

```

Полный код доступен как *peristaltic.py* в файле *peristaltic.blend*.



## Применение *peristaltic.py* к арматуре

Чтобы использовать этот скрипт, Вы должны запустить его с выбранным объектом арматуры. Рецепт, чтобы продемонстрировать его применение, будет заключаться в следующем:

1. Добавьте арматуру к сцене
2. Перейдите в режим *редактирования*, и выдавите любое число костей из конца первой кости.
3. Перейдите в *объектный* режим и добавьте меш, отцентрированный в позиции арматуры. Любой меш будет работать,

но в нашей иллюстрации, мы используем цилиндр со множеством подразбиений.

4. Выберите меш, затем с Shift'ом выберите арматуру. Теперь как арматура, так и Меш-объект выбраны, но в то же время арматура является активным объектом.

5. Нажмите *Ctrl + P* и выберите **armature**. В появившемся после этого меню, выберите **Create from bone heat**. Это создаст группу вершин в меше для каждой кости в арматуре. Эти группы вершин будут использованы для деформации меша, когда мы ассоциируем арматуру с мешем в качестве модификатора.

6. Выберите меш и добавьте модификатор *armature*. Наберите имя арматуры в поле **Ob:** и убедитесь, что выбран переключатель **Vert. Group**, а **Envelopes** - нет.

7. Выберите арматуру и запустите *peristaltic.py*.

В результате будет анимированный Меш-объект, имеющий сходство с прохождением куска через узкую гибкую трубу. Несколько кадров показаны на иллюстрации:



Водосточные трубы являются, конечно, не единственным полым по форме объектом для анимации этим путём, как показано на следующей иллюстрации:



## Get down с ритмом - синхронизация ключей формы со звуком

(Словосочетание *Get Down* имеет такое количество самых разнообразных значений, что я не рискнул выбирать из них и оставляю без перевода — прим. пер.)

Многочисленные рок-видео сегодня часто показывают анимацию диффузора динамика, вибрирующего в такт со звуком музыки. И хотя возможности для манипуляций со звуком в API Блендера довольно малы, но мы увидим, что этого эффекта несложно достигнуть.

Анимация, которую мы создадим, зависит главным образом от манипуляции **ключами формы (shape keys)**. Ключи Формы можно представлять как искажения базового меша. Меш может иметь много таких искажений и каждому из них даётся определённое имя. Интересно то, что Блендер предоставляет нам возможность интерполяции между базовой формой и любой из искаженных форм непрерывным способом, позволено даже смешивать вклады от разных форм.

Вот, например, один из способов анимировать наш диффузор динамика, нужно смоделировать основную, неискаженную форму

диффузора; добавить ключ формы к этому базовому мешу; и исказить его, чтобы появилось сходство с диффузором, который вытолкнут наружу. После этого мы сможем смешивать между собой эти "вытолкнутую" и базовую формы в зависимости от громкости звука.

Анимирование установкой ключевых кадров в Блендере означает создание кривых IPO и манипуляция ими, как мы уже видели раньше. На самом деле, кривые IPO Shape или Key очень похожи на другие типы IPO и управляются практически так же. Основное различие между, например, IPO Объекта и IPO Формы - в том, что индивидуальные кривые IPO Формы проиндексированы не некоторой встроенной числовой константой (как например, *ipo.OB\_LOCX* для Объектов), а строкой, поскольку пользователь может определить любое количество именованных форм.

Также, IPO Формы доступны не через Объект, а через лежащий в его основе *Меш-объект* (или *Решетку*, или *Кривую*, так как они тоже могут иметь ключи формы).

## Манипуляция звуковыми файлами

Так что теперь, когда мы знаем, как анимировать формы, нашей следующей целью будет выяснить, как добавить какой-либо звук к нашему мешу, или, вернее, определить для каждого кадра, насколько искажённую форму должно быть видно.

Как упомянуто в предыдущем разделе, API Блендера не обеспечивает большого количества инструментов для работы со звуковыми файлами, в основном модуль *Sound* обеспечивает нас способом загрузки и воспроизведения звуковых файлов, но на этом и всё. Нет способа получить доступ к индивидуальным точкам волны, закодированным в файле.

К счастью, в стандартный дистрибутив Питона включен модуль *wave*, который обеспечивает нас средствами для чтения файлов в обыкновенном формате *.wav*. Хотя он поддерживает только несжатый формат, этого будет достаточно, так как этот формат является очень распространённым, и большинство инструментов работы со звуком, как например, **Audacity**, могут преобразовывать в этот формат. С этим модулем мы можем открыть *.wav*-файл, определить частоту

сэмпл и длительность звукового клипа, и получить индивидуальные сэмплы. Как мы увидим в объяснении следующего кода, мы все еще должны преобразовывать эти сэмплы в величины, которые мы можем использовать как значения ключей для наших ключей формы, но тяжелую работу уже сделали для нас.

### Схема кода: *Sound.py*

Вооружившись знаниями о том, как создавать кривые IPO и получать доступ к .wav-файлам, мы можем наметить следующую схему программы:

1. Определить, имеет ли активный объект пригодные заданные формы, и предложить выбрать их.
2. Позволить пользователю выбрать .wav-файл.
3. Определить количество звуковых сэмплов в секунду в файле (частота дискретизации).
4. Вычислить количество необходимых кадров анимации, основываясь на длительности звукового файла и показателе количества видеок кадров в секунду.
5. Затем, для каждого кадра анимации:
  - Усреднить звуковые сэмплы, проходящие в этом кадре
  - Установить величину смешивания выбранной кривой IPO этому среднему (нормализованному) числу

Полный код доступен как *Sound.py* в файле *sound000.blend* и объясняется следующим образом:

```
import Blender
from Blender import Scene, Window, Draw
from Blender.Scene import Render
```

```
import struct
import wave
```

Мы начинаем, импортируя необходимые модули, включая модуль Питона *wave*, чтобы иметь доступ к нашему .wav-файлу и модуль *struct*, который предоставляет функции для манипулирования двоичными данными, которые мы получим из .wav-файла.

Затем, мы определяем вспомогательную функцию, показывающую всплывающее меню в середине нашего экрана. Она ведёт себя просто подобно стандартной функции *PupMenu()* из модуля *Draw*, но устанавливает курсор в позицию середины экрана с помощью функций *GetScreenSize()* и *SetMouseCoords()* из модуля Блендера *Window*:

```
def popup(msg):
    (w,h)=Window.GetScreenSize()
    Window.SetMouseCoords(w/2,h/2)
    return Draw.PupMenu(msg)
```

Основная часть работы будет осуществляться функцией *sound2active()*. Она принимает два аргумента - имя .wav-файла, для использования и имя ключа формы для анимации, основанной на информации из .wav-файла. Сначала, мы пытаемся создавать объект *WaveReader*, вызывая функцию *open()* модуля *wave* (выделено). Если это не удаётся, мы показываем ошибку во всплывающем окне и выходим:

```
def sound2active(filename, shapekey='Pop out'):
    try:
        wr = wave.open(filename, 'rb')
    except wave.Error, e:
        return popup(str(e)+'%t|Ok')
```

Затем мы делаем некоторые разумные проверки: мы сначала проверяем, является ли .wav-файл *МОНО* файлом. Если Вы хотите использовать стерео файл, преобразуйте его сначала в моно, например с помощью свободного пакета Audacity (<http://audacity.sourceforge.net/>). Затем мы проверяем, имеем ли мы дело с несжатым .wav-файлом, поскольку модуль *wave* не может работать с другими типами. (большинство .wav-файлов являются несжатыми, но если нужно, Audacity также может их преобразовать), и мы проверяем, что сэмплы 16-битовые. Если любая из этих проверок терпит неудачу, мы выводим соответствующее сообщение об ошибке:

```
c = wr.getnchannels()
if c!=1 : return popup('Only mono files are '+
                        'supported%t|Ok')

t = wr.getcomptype()
w = wr.getsampwidth()
if t!='NONE' or w!=2 :
```



```
return popup('Only 16-bit, uncompresses files '+
            'are supported\t|Ok')
```

Теперь, когда мы можем работать с файлом, мы получаем его **частоту дискретизации** (frame rate, количество аудио сэмплов в секунду) и общее число байт (как ни странно, используя неуклюже названную функцию *getnframes()* из модуля *wave*). Затем, мы считываем все эти байты и сохраняем их в переменной *b*.

```
fr= wr.getframerate()
n = wr.getnframes()
```

```
b = wr.readframes(n)
```

Наша следующая задача в том, чтобы получить контекст рендера у текущей сцены, чтобы извлечь количество видеок кадров в секунду. Время в секундах нашей проигрываемой анимации будет определено длиной нашего аудио сэмпла, которое мы можем вычислить, разделив общее число аудио кадров в .wav-файле на количество аудио кадров в секунду (выделено в следующей части кода). Затем мы определяем константу *sampleratio* - количество аудио кадров в течение видео кадра:

```
scn          = Scene.GetCurrent()
context      = scn.getRenderingContext()
seconds      = float(n)/fr
sampleratio  = fr/float(context.framesPerSec())
```

Как упомянуто раньше, модуль *wave* дает нам доступ ко множеству свойств .wav-файла и к сырым (raw) аудио сэмплам, но не предоставляет никаких функций для преобразования этих сырых сэмплов в удобные для использования целые величины. Следовательно, нам нужно сделать это самостоятельно. К счастью, это не так уж трудно, как это может показаться. Поскольку мы знаем, что 16-битовые аудио сэмплы представлены как 2-х байтовое целое в формате "меньший-вконце" ("little-endian"), мы можем использовать функцию *unpack()* из модуля Питона *struct*, чтобы эффективно преобразовывать список байтов в список целых, передавая подходящую спецификацию формата. (Вы можете прочитать больше о .wav-файлах здесь <https://ccrma.stanford.edu/courses/422/projects/WaveFormat/>, на русском здесь: <http://www.fpga-cpld.ru/wave.html>, работа с модулем *struct* описана здесь: <http://world-python.org/article/tutorialmodules/32-modul-struct.html> — прим. пер.)

```
samples = struct.unpack('<%dh'%n,b)
```

Теперь мы можем начать анимацию ключей формы. Мы получаем стартовый кадр из контекста рендера и вычисляем конечный кадр, умножая время в секундах в .wav-файле на частоту видеок кадров. Заметьте, что он может оказаться дальше или ближе, чем конечный кадр, который мы можем получить из контекста рендера. Последний определяет конечный кадр, который будет отрендерен, когда пользователь нажмёт на кнопку **Anim**, но мы будем анимировать движение нашего активного объекта независимо от этой величины.

Затем для каждого кадра мы вычисляем от стартового кадра до последнего кадра (исключительно) среднее значение аудио сэмплов, которые попадают на каждый видеок кадр суммированием этих аудио сэмплов (находятся в списке *samples*) и деля на количество этих аудио сэмплов за видеок кадр (выделено в следующем куске кода).

Мы задаём выбранный ключ формы в величину в диапазоне [0:1], так что мы должны нормализовать рассчитанные средние числа, определяя минимальную и максимальную величины, и вычислить масштаб:

```
staframe = context.startFrame()
endframe = int(staframe +
               seconds*context.framesPerSec())

popout=[]
for i in range(staframe,endframe):
    popout.append(sum(samples[int(
        (i-1)*sampleratio):int(i*sampleratio)])/sampleratio)
minvalue = min(popout)
maxvalue = max(popout)
scale = 1.0/(maxvalue-minvalue)
```

Наконец, мы получаем активный объект в текущей сцене и получаем его IPO Формы (выделено). Мы заканчиваем, устанавливая величину ключа формы для каждого кадра в рассматриваемом нами диапазоне в масштабированное среднее аудио сэмплов:

```
ob=Blender.Scene.GetCurrent().objects.active
```

```
ipo = ob.getData().getKey().getIpo()
```

```
for i, frame in enumerate(range(staframe, endframe)):
    ipo[shapekey][frame] = (popout[i] - minvalue) * scale
```

Остальной скрипт теперь довольно прост. Он выбирает активный объект, затем пытается извлечь список имен ключей формы из него (выделено в следующей части). Это действие может потерпеть неудачу (следовательно, применяется *try ... except*), если, например, активный объект - не меш или он не имеет связанных ключей формы, в этом случае мы предупреждаем пользователя с помощью всплывающего сообщения:

```
if __name__ == "__main__":
    ob = Blender.Scene.GetCurrent().objects.active

    try:
        shapekeys = ob.getData().getKey(
            ).getIpo().curveConsts
        key = popup(('Select a shape key%t|'+
            '|').join(shapekeys))
        if key > 0:
            Window.FileSelector(lambda f: sound2active(f,
                shapekeys[key-1]),
                "Select a .wav file",
                Blender.Get('soundsdir'))
    except:
        popup('Not a mesh or no shapekeys defined%t|Ok')
```

Если мы смогли извлечь список ключей формы, мы предоставляем пользователю всплывающее меню для выбора из этого списка. Если пользователь выбирает один из пунктов, переменная *key* будет положительной и мы предоставляем пользователю диалог выбора файлов (выделено). В этот диалог передаётся *lambda*-функция, которая будет вызвана, если пользователь выберет файл, с передачей имени этого выбранного файла в качестве аргумента. В нашем случае мы создаём эту *lambda*-функцию, чтобы вызвать функцию *sound2active()*, определённую ранее с этим именем файла и выбранным ключом формы.

Начальный каталог, который будет представлен пользователю в выборе файлов, определяется последним аргументом в функции *FileSelector()*. Мы задали его параметром Блендера *soundsdir*. Это обычно // (то есть, относительный путь, указывающий на тот же

каталог, где находится .blend-файл, с которым пользователь работает), но может быть установлен в окне Пользовательских настроек (секция **File Paths**) на нечто другое.

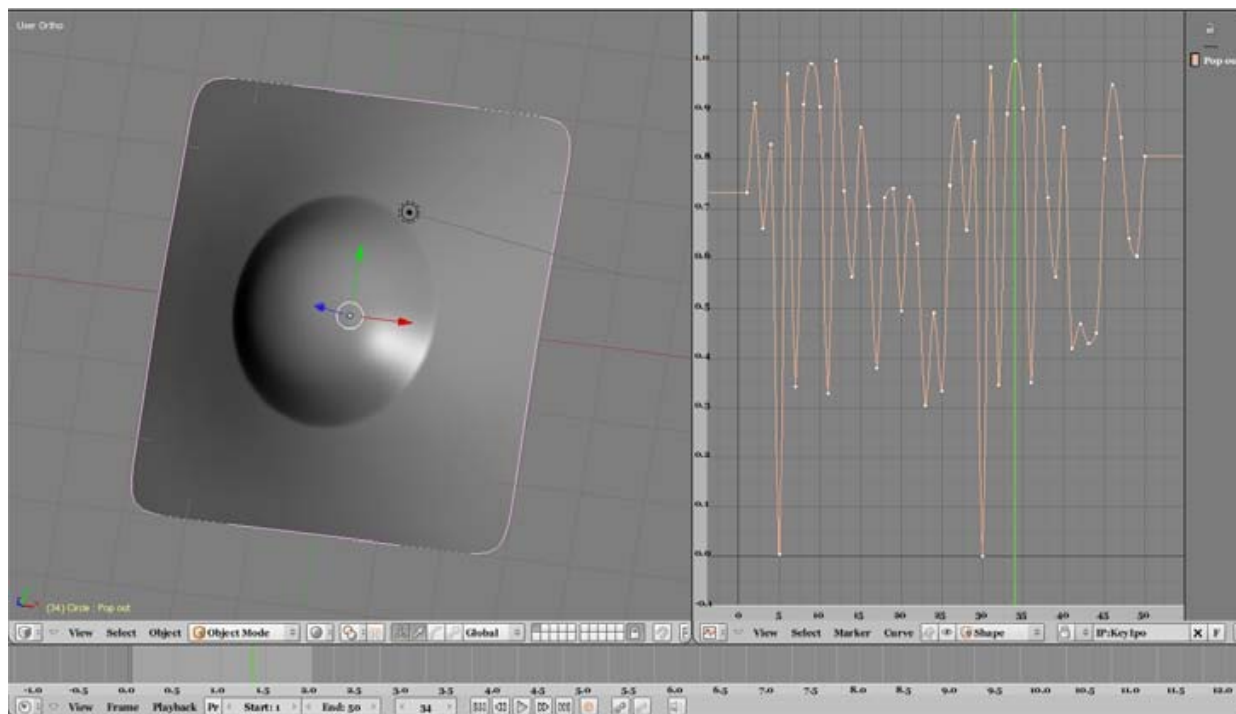
## Анимация меша .wav-файлом: последовательность действий

Теперь, когда у нас есть наш скрипт *Sounds.py*, мы можем применить его следующим образом:

1. Выбрать Меш-объект
2. Добавить ключ формы "*Basis*" к нему (**окно Кнопки, контекст редактирования, панель Shapes**). Он будет соответствовать наименее искаженной форме меша.
3. Добавить второй ключ формы и дать ему значимое имя.
4. Отредактировать этот меш, чтобы он представлял наиболее искаженную форму.
5. В режиме объектов, запустить *Sound.py* из текстового редактора, нажимая *Alt + P*.
6. Выбрать имя ключа формы, определенное раньше (не "*Basis*"), из выпадающего меню.
7. Выбрать .wav-файл для выполнения.



Результатом будет объект с кривой IPO для выбранного ключа формы, который будет колебаться согласно ритму звука, как показано на следующем скриншоте:



## Итог

В этой главе мы увидели как соединять ключи формы с мешем и как добавлять кривые IPO, чтобы анимировать переходы между этими ключами формы. Подробнее, мы узнали как:

- определять кривые IPO
- определять ключи формы на меше
- определять кривые IPO для этих ключей формы
- позировать арматуры
- группировать изменения поз в действия

В следующей главе мы должны узнать, как создавать заказные текстуры и шейдеры.

## Создание заказных шейдеров и текстур с помощью Pynodes

Иногда говорят, что, хотя Блендер имеет мощную и разностороннюю систему для определения материалов, ему недостает соответствующего шейдерного языка, чтобы определять полностью новые **шейдеры**, например, для создания материалов, которые реагируют на свет новыми способами. Тем не менее, это не совсем так.

Блендер не имеет компилируемого шейдерного языка, но он имеет мощную **нодовую** (узловую) систему для комбинирования текстур и материалов, и эти ноды могут быть скриптами на Питоне (Pynodes). Это позволяет определять полностью новые текстуры и материалы.

В этой главе мы изучим:

- Как писать Pynodes, которые создают простые цветные узоры
- Как писать Pynodes, которые производят узоры с нормальями
- Как писать анимированные Pynodes
- Как писать материалы, зависящие от высоты и наклона
- Как создавать шейдеры, которые реагируют на угол падающего света

Для того, чтобы немного проиллюстрировать эту силу, мы начнём с рассмотрения скрипта, который создает регулярные цветные узоры, созданные из треугольников, прямоугольников, или шестиугольников.



**Материалы, шейдеры, и текстуры** - термины, которые часто используются как синонимы, хотя между ними есть разница в значении. Для наших целей мы попытаемся придерживаться следующих определений: **текстура** является основным строительным блоком, например, цветной или нормальный узор или просто некоторая функция, которая возвращает значение в зависимости от позиции на поверхности. **Шейдер** принимает на вход любое количество текстур или просто базовый цвет и возвращает цвет, основанный на влиянии падающего света и, возможно, направления вида. **Материал** — это набор текстур, шейдеров, и всех типов свойств, которые могут быть приложены к объекту. Pynodes могут быть текстурами, а также шейдерами.

## Основы

Когда мы разрабатываем Pynode, мы в основном разрабатываем нечто, что предоставляет функцию, которая вызывается для каждого пикселя на экране, который должен быть затенен (shaded) этим нодом (или даже неоднократно, если включен **oversampling**). Эта функция получает, кроме прочего, координаты x, y, и z точки на затеняемом объекте, которая соответствует пикселю на экране, который мы к настоящему времени вычисляем. Затем функция должна вернуть что-то полезное, такое как цвет, значение интенсивности, или что-то чуть менее интуитивное, например, нормаль.

В окне редактора Нодов Блендера каждый нод материала, включая Pynode, представлен прямоугольником, который имеет входы слева и выходы справа. Эти входы и выходы, часто называемые **сокетами**, представлены небольшими цветными кругами (смотрите следующий скриншот). Эти сокеты можно использовать для связи нодов вместе; щелкая по выходному сокету одного нода и перетаскивая мышью ко входному сокету другого нода, эти ноды будут связаны. Так, комбинируя требуемым образом множество различных нодов, можно создать очень сложные и мощные шейдеры.

## От нодов к Pynodes

Сила системы Нодов Блендера проистекает не только из её многочисленных встроенных типов нодов, и множества способов, которыми эти ноды могут быть связаны, но также из того, что мы можем написать новые ноды на Питоне, которые можно связывать так же, как обычные ноды.

Для Pynodes нужен способ получать доступ к информации, передаваемой входными сокетами и способ посылать рассчитанные результаты в выходные сокеты. Понятие нода и сокетов структурировано в соответствии с объектно-ориентированной моделью. Давайте бросим первый взгляд на небольшой пример кода, чтобы доказать, что это не страшно (ветераны объектно-ориентированного программирования: поглядите в другую сторону

или смотрите сквозь пальцы, чтобы просто разобраться с определением класса из следующего примера):

```
from Blender import Node
```

```
class MyNode(Node.Scripted):
```

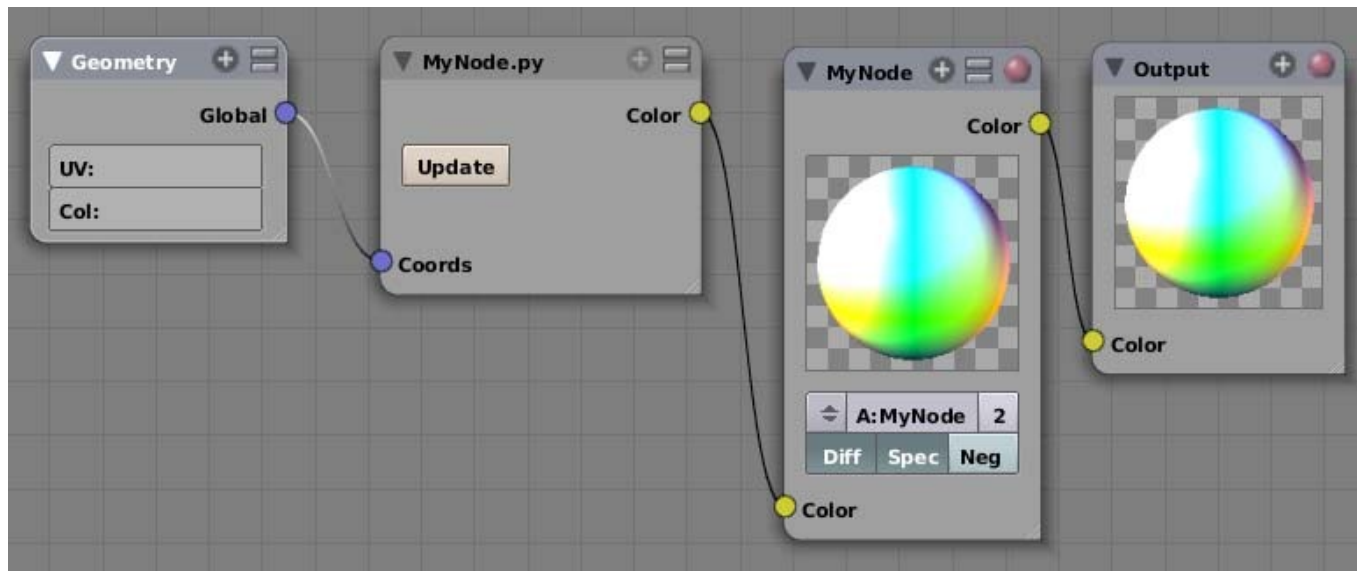
```
    def __init__(self, sockets):
        sockets.input = [Node.Socket('Coords',
                                     val= 3*[1.0])]
        sockets.output = [Node.Socket('Color',
                                     val = 4*[1.0])]
```

```
    def __call__(self):
        x,y,z = self.input.Coords
        self.output.Color = [abs(x),abs(y),abs(z),1.0]
```

Прежде чем мы посмотрим на этот код подробно, попробуем его в Блендере, чтобы посмотреть, как он работает на практике:

1. Откройте новый файл в текстовом редакторе и дайте ему значимое имя.
2. Скопируйте код примера.
3. Создайте простую сцену, например, простую UV-сферу в начале координат с парой ламп и камерой.
4. Назначьте Нодовый материал сфере как обычно.
5. Наконец, добавьте *динамический (Dinamic)* нод в Нодовом редакторе (**Add | Dynamic**) и выберите имя файла, который Вы отредактировали, щелчком на кнопке выбора *динамического* нода и выбрав файл из списка.

Результирующая сеть нодов (часто называемая **макаронами (noodle)**), может выглядеть похоже на это:



Это - векторный вход, поскольку инициализируется списком трех чисел с плавающей точкой, который определяет значение по умолчанию, если этот входной сокет не подключен к другому ноду. Векторные сокеты представлены как синие круги в нодовом редакторе.

Другие типы входного сокета также возможны и этот тип определяется величиной аргумента *val*. Выходные сокеты определяются так же. Список трех чисел с плавающей точкой определяет векторный сокет, список четырех чисел - цветовой сокет (с красным, зеленым, синим, и альфа компонентом), а сокет, представляющий простое значение, как, например, интенсивность, инициализируется

Если Вы рендерите сферу, результатом будет красочный шар, похожий на виджет выбора цвета.

Теперь вернёмся к коду.

На первой строке мы импортируем модуль *Node* из Блендера, поскольку мы создаём новый тип нода, но основное его поведение уже определено в модуле *Node*.

Затем мы определяем класс *MyNode*, подкласс *Node.Scripted*, который будет вести себя просто подобно ноду *Scripted*, за исключением тех частей, которые мы переопределим.

Затем, мы определяем функцию *\_\_init\_\_()*, которая будет вызываться первый раз при создании этого типа Pynode в редакторе нодов, или всякий раз, когда мы щелкаем на кнопку **Update**. Когда это случается, Блендер передаёт два аргумента в эту функцию: *self*, ссылку на нод, который мы используем, и *sockets*, ссылку на объект, которая будет указывать на наши списки входных и выходных сокетов. С их помощью ноды в редакторе нодов получают данные на вход или посылают их дальше.

На выделенной строке мы определяем список определений входных сокетов; в нашем случае только один с названием *Coords*.

единственным числом. Заметьте, что мы не можем отличать входы, которые должны заполняться пользователем от тех, что должны быть подключены к другому ноду. Мы используем входные сокеты для них обоих и должны подтверждать их предполагаемое использование. К настоящему времени, нет средства добавлять кнопки или другие управляющие элементы на Pynode.

Нашему примеру Pynode нужен также выход, так что мы определяем список, состоящий из единственного выходного сокета называемого *Color*. У него есть четыре величины с плавающей точкой по-умолчанию, определяющих красную, зеленую, синюю, и альфа величины соответственно.

Затем мы определяем функцию *\_\_call\_\_()*, которая вызывается всякий раз при затенении пикселя. Она не принимает никаких аргументов, но *self* - это ссылка на текущий нод, которая используется в следующих строках для получения доступа к входному и выходному сокетам.

В теле функции *\_\_call\_\_()* мы извлекаем три компонента из входного сокета с названием *Coords* и назначаем их переменным, которые легко запомнить. Наконец, мы создаем новый четырехкомпонентный список, который представляет наш

рассчитанный цвет и назначаем его выходному сокету с названием *Color*.

Это - основа для определения простых текстур, но существует больше информации, пригодной для нода (как мы увидим в следующих разделах), так что мы сможем разработать несколько красивых продвинутых эффектов. В следующем разделе мы создадим чуть более сложный нод, который формируется на тех же принципах, что мы видели раньше, но создаёт более полезные узоры.

## Регулярное заполнение

Текстура шахматной доски является, возможно, самой простой текстурой, которую Вы можете себе представить и, следовательно, часто используется в качестве примера при программировании текстур. Поскольку Блендер уже имеет встроенную клетчатую текстуру (начиная с версии 2.49, в текстурном контексте окна нодов), мы хотим пройти на один шаг дальше и создать текстурный нод, который отображает не только текстуру шахматной доски, но может **заполнять (tilings)** также треугольниками и шестиугольниками.

```
from Blender import Node, Noise, Scene
from math import sqrt, sin, cos, pi, exp, floor
from Blender.Mathutils import Vector as vec
```

```
# создаёт регулярное заполнение для использования в
# качестве цветовой карты
```

```
class Tilings(Node.Scripted):
    def __init__(self, sockets):
        sockets.input = [Node.Socket('type' ,
                                   val= 2.0, min = 1.0, max = 3.0),
                        Node.Socket('scale' ,
                                   val= 2.0, min = 0.1, max = 10.0),
                        Node.Socket('color1',
                                   val= [1.0,0.0,0.0,1.0]),
                        Node.Socket('color2',
                                   val= [0.0,1.0,0.0,1.0]),
                        Node.Socket('color3',
                                   val= [0.0,0.0,1.0,1.0]),
                        Node.Socket('Coords',
```

```
val= 3*[1.0])]
```

```
sockets.output = [Node.Socket('Color',
                               val = 4*[1.0])]
```

Первые несколько строк начинают определение наших входных и выходных сокетов. Выход в любом случае будет просто цветом, но набор входных сокетов у нас более разнообразный. Мы определяем три различных входных цвета, поскольку при заполнении шестиугольниками нужно три цвета, чтобы дать каждому шестиугольнику цвет, отличный от своего соседа.

Мы также определяем вход *Coords*. Этот входной сокет может перехватывать любой выход сокета геометрии. Таким образом у нас есть множество возможностей отобразить нашу цветную текстуру на объект, который мы текстурируем. Сокет *Scale* определяется также, чтобы управлять размером нашей текстуры.

Наконец, мы определяем сокет *Type*, чтобы выбирать узор, который мы хотим генерировать. Так как API для Rynode не обеспечивает выпадающих меню или любого другого простого управляющего элемента для выбора, мы делаем сокет с одиночным значением и произвольно выбираем величины, представляющие наш выбор: *1.0* для треугольников, *2.0* для шахматного поля, и *3.0* для шестиугольников.

Мы заканчиваем нашу функцию `__init__()` определением множества констант и словаря распределений цвета, который мы будем использовать при генерации шестиугольной текстуры.

```
self.cos45 = cos(pi/4)
self.sin45 = sin(pi/4)
self.stretch = 1/sqrt(3.0)
self.cmap = { (0,0):None, (0,1):2,   (0,2):0,
              (1,0):0,   (1,1):1,   (1,2):None,
              (2,0):2,   (2,1):None, (2,2):1 }
```

Следующим шагом будет определение функции `__call__()`:

```
def __call__(self):

    tex_coord = self.input.Coords
    # мы игнорируем любую z-координату
    x = tex_coord[0]*self.input.scale
    y = tex_coord[1]*self.input.scale
```

```
c1 = self.input.color1
c2 = self.input.color2
c3 = self.input.color3
```

```
col= c1
```

Функция `__call__()` начинается с определения нескольких сокращений для входных величин и умножения координатного входа на выбранный масштаб, чтобы растянуть или уменьшить сгенерированный узор. Следующий шаг должен установить тип желательного узора и вызвать подходящую функцию для вычисления выходного цвета для данных координат. Результирующий цвет назначается в наш единственный выходной сокет:

```
if self.input.type<= 1.0:
    col = self.triangle(x,y,c1,c2)
elif self.input.type <= 2.0:
    col = self.checker(x,y,c1,c2)
else:
    col = self.hexagon(x,y,c1,c2,c3)
```

```
self.output.Color = col
```

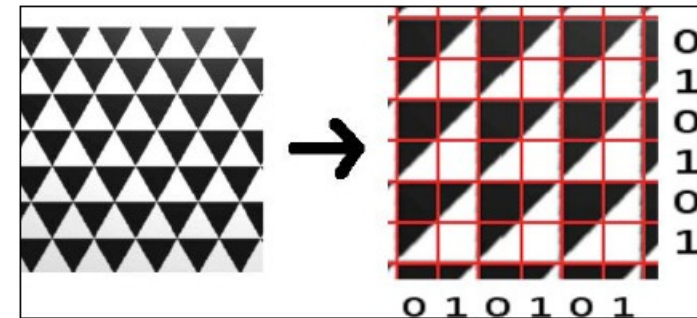
Все различные функции генерации узоров очень похожи; они берут координаты  $x$  и  $y$  и два или три цвета в качестве аргументов и возвращают единственный цвет. Так как это функции-члены класса, они также принимают дополнительный первый аргумент `self`.

```
def checker(self,x,y,c1,c2):
    if int(floor(x%2)) ^ int(floor(y%2)):
        return c1
    return c2
```

Функция `checker` проверяет, в какой строке и колонке мы находимся, и если номер строки и номер колонки - оба нечетные или четные (что устанавливает оператор *исключающее или*), она возвращает один цвет, если нет, то возвращает другой цвет.

```
def triangle(self,x,y,c1,c2):
    y *= self.stretch
    x,y = self.cos45*x - self.sin45*y,
           self.sin45*x + self.cos45*y
    if int(floor(x%2)) ^ int(floor(y%2)) ^ \
        int(y%2>x%2) : return c1
    return c2
```

Функция `triangle` сначала одновременно вращает как  $x$ , так и  $y$  координаты на угол 45 градусов (превращение квадратов в вертикальные ромбы). Затем она определяет цвет, основываясь на номерах строки и колонки в точности подобно функции `checker`, но с уловкой: третье условие (выделено) проверяет, слева ли мы от диагонали, пересекающей квадрат, и поскольку мы вращали нашу сетку, на самом деле мы проверяем действительно ли координаты выше горизонтальной линии, делящей наш ромб. Это может звучать немного сложным, но Вы можете посмотреть на следующую иллюстрацию, чтобы понять идею:



```
def hexagon(self,x,y,c1,c2,c3):
    y *= self.stretch
    x,y = self.cos45*x - self.sin45*y,
           self.sin45*x + self.cos45*y
    xf = int(floor(x%3))
    yf = int(floor(y%3))
    top = int((y%1)>(x%1))
    c = self.cmap[(xf,yf)]
    if c == None:
        if top :
            c = self.cmap[(xf,(yf+1)%3)]
        else :
            c = self.cmap[(xf,(yf+2)%3)]
    return (c1,c2,c3)[c]
```

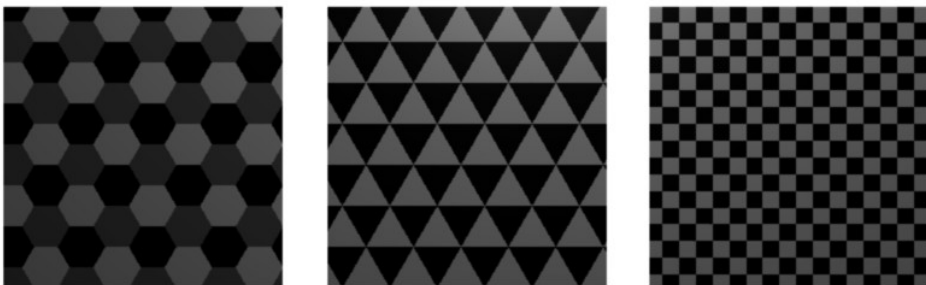


Функция *hexagon* (шестиугольник) во многих отношениях похожа на функцию *triangle* (в конце концов шестиугольник - шесть треугольников, склеенных вместе). Следовательно, в ней применяется та же хитрость с вращением, но, вместо выбора цвета с использованием простой формулы, здесь всё несколько сложнее, и, следовательно, мы используем цветовую карту (выделено в предыдущем фрагменте кода). В основном, мы делим экран на горизонтальные и вертикальные полосы, и выбираем цвет, основываясь на том, в какую полосу мы попали.

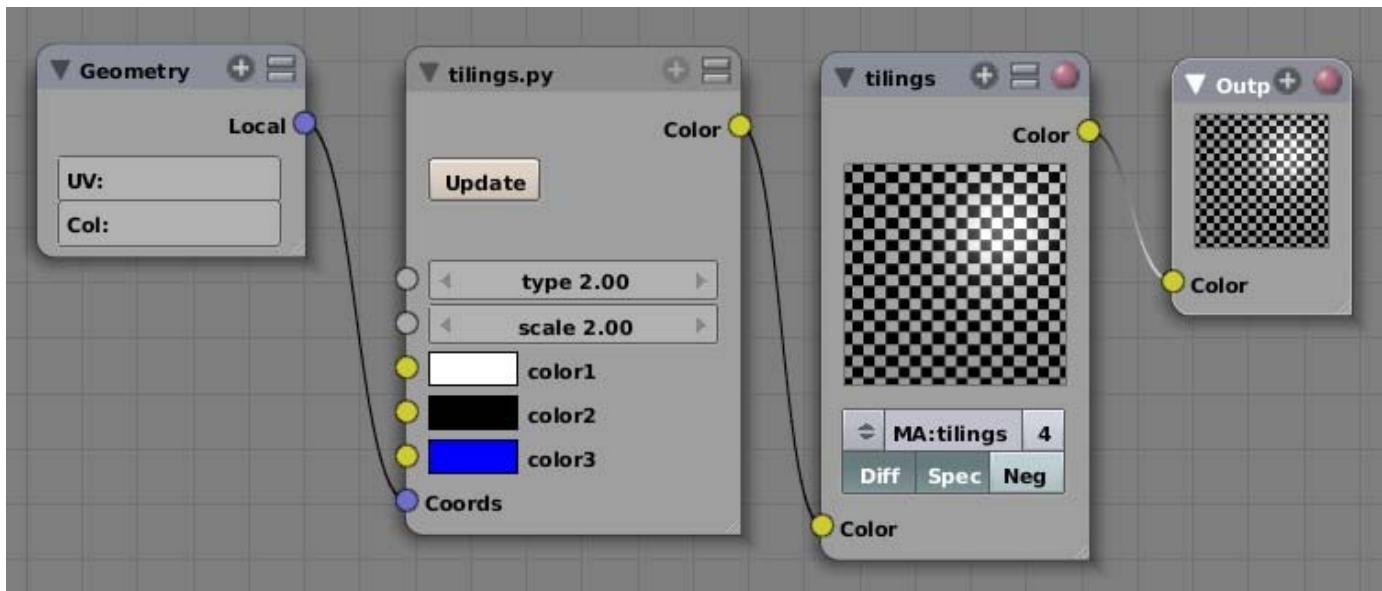
Последняя часть магии - на последней строке нашего скрипта:

```
__node__ = Tilings
```

В текущей реализации Pynodes, Блендеру нужно это присвоение, чтобы идентифицировать класс в качестве нода. Наш нод появится в выпадающем меню скриптовых нодов как **Tilings**. Полный код доступен как *tilings.py* в файле *tilings.blend* вместе с примером нодовой сети. Некоторые возможные узоры показаны на следующем скриншоте:



Соответствующая нодовая сеть показана на следующем скриншоте. Заметьте, что мы не подключили никаких нодов к цветовым входам, но можно создать даже более сложные узоры, если мы это сделаем.



## Anti-aliasing

Если вы посмотрите внимательно на диагональные границы шестиугольного или треугольного узора, вы должны обратить внимание на некоторые артефакты наподобие лестницы, даже если *oversampling* был установлен на большое значение.

Сам Блендер достаточно умен, чтобы прилагать выбранный уровень **anti-aliasing**, например, к границам объектов, но в большинстве случаев текстуры на поверхности должны самостоятельно заботиться о наложении anti-aliasing. Встроенные текстуры Блендера, конечно, разработаны именно таким образом, но наши собственные текстуры, произведенные с помощью Pynodes, должны заниматься этим явно.

Существуют многочисленные математические методы, позволяющие уменьшить aliasing в сгенерированных текстурах, но большинство из них не так просто осуществить, или они требуют специфических знаний о способе генерации узора. К счастью, Блендер предоставляет нам опцию **Full OSA** (окно **Кнопки | контекст Затенения | кнопки Материала | панель Links and pipeline**). Если мы включим эту опцию, Блендер будет вынужден



производить oversample с каждым пикселем в нашей текстуре в количестве, выбранном в кнопках рендера. Это дорогой вариант, но он позволит отделаться от эффектов aliasing без необходимости осуществлять специфические параметры фильтрации в нашем текстурном Pynode.

## Индексирование текстуры вектором

На нашем плиточном узоре мы ограничили цвета в минимальное число, которое нужно для различения каждой из соседних плиток. Но возможно ли нам назначать произвольные цвета, основанные на некоторой шумовой текстуре? Таким образом мы могли бы раскрасить рыбу чешую общим произвольным узором, раскрашивая каждую отдельную чешуйку однотонно.

Мы не можем просто подключить цветную текстуру к цветовым входам, так как это приведёт, может быть, к интересной модели, но каждая плитка не будет иметь однородной окраски. Решением будет модифицировать наш Pynode, чтобы производить уникальный вектор, который станет однородным в пределах любой данной плитки. Этот вектор затем может быть подключен к любой шумовой текстуре, которая принимает вектор на входе, так же, как делают все текстуры Блендера. Этот вектор используется нодом текстуры шума, чтобы указывать на единственную точку в произвольной текстуре, и таким способом мы можем произвести произвольно окрашенные, но однородные элементы.

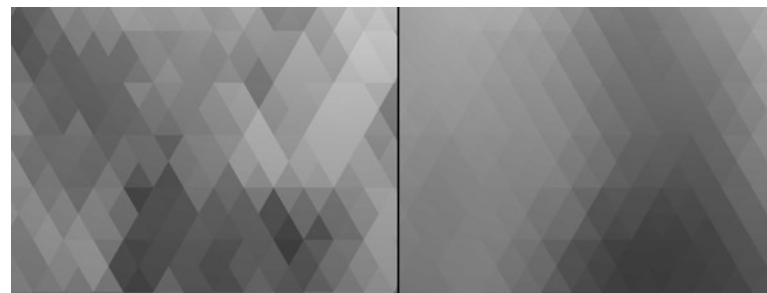
Чтобы обеспечить такую функциональность, мы модифицируем наш код, удалив цветовые входы, и заменяя цветовой выход векторным выходом (не показано). Код в функции `__call__()` теперь должен будет производить вектор вместо цвета. Здесь мы покажем модифицированную функцию `triangle` (полный код доступен как `tilingsv.py` в файле `tilingsv.blend`):

```
def triangle(self,x,y):
    y *= self.stretch
    x,y = self.cos45*x - self.sin45*y,
           self.sin45*x + self.cos45*y
```

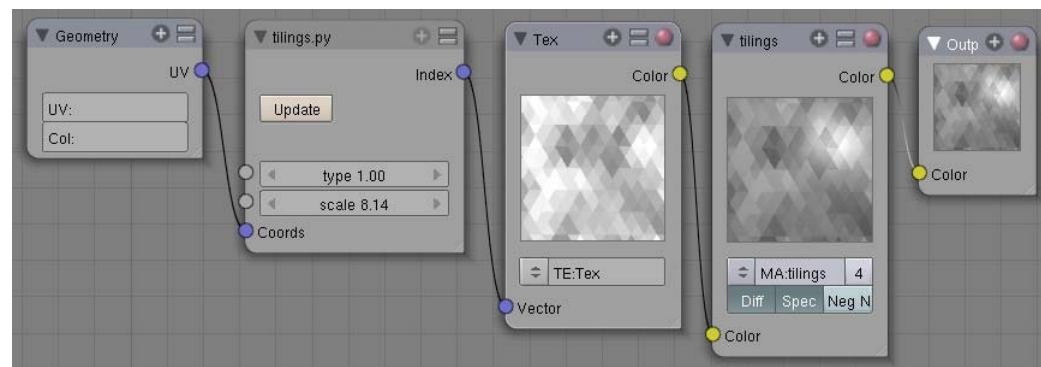
```
if int(floor(x%2)) ^ int(floor(y%2)) ^ \
   int(y%2>x%2) :
    return [floor(x),floor(y),0.0]
return [floor(x)+0.5,floor(y),0.0]
```

Логика в основном та же, но, как показано на выделенной строке, мы возвращаем вектор, который зависит от позиции. Тем не менее, из-за операции `floor()`, он постоянен в пределах треугольника. Заметьте, что для альтернативного треугольника мы добавляем незначительное смещение; не имеет значения какое именно смещение мы выберем до тех пор, пока оно постоянно и производит вектор, отличающийся от других треугольников.

Результаты показывают произвольный узор из треугольников, которые следуют за большими корреляциями шума оставляя каждый индивидуальный треугольник с однородным цветом. Образец справа имеет больший размер шума в использованной текстуре cloud:



Возможная настройка нодов показана на следующем скриншоте:



## Свежий бриз - текстуры с нормальми

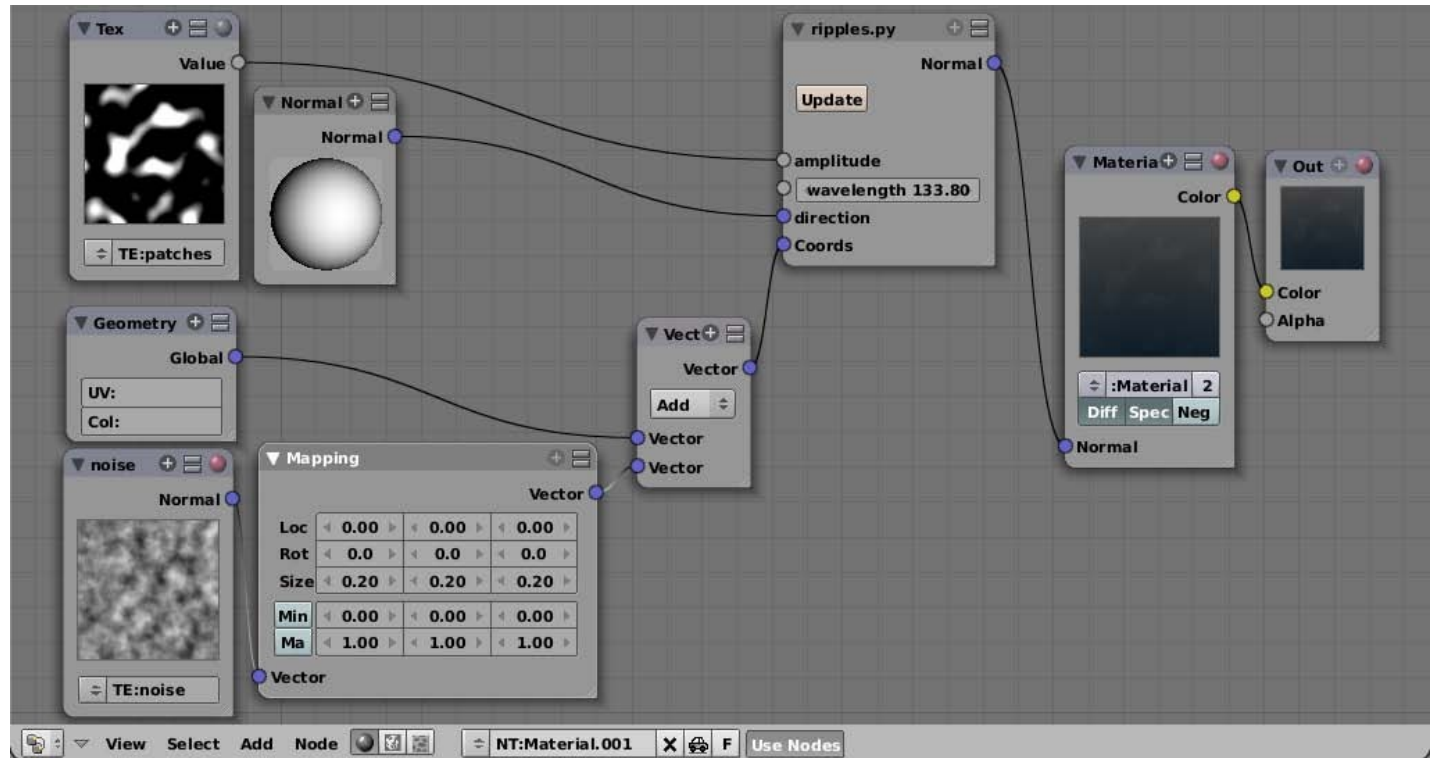
Текстура может иметь больше, чем просто геометрический вход. Если Вам нужна текстура, изменяющая свое поведение в зависимости от другой текстуры, этого не получится достигнуть простой настройкой нодов, которую Вы можете обеспечить дополнительными входными сокетами. Мы разработаем Rynode, генерирующий карту нормалей, которая имитирует небольшие пятна **всплесков (wavelets)** в пруду во время почти безветренного дня.

Места, где эти пятна появляются, определяются дополнительным входным сокетом, который может быть связан с почти любой текстурой шума. Мы дадим этому входному сокету имя *amplitude* (амплитуда), поскольку мы используем его, чтобы перемножать с нашими рассчитанными **нормальми**. Таким образом, наши всплески будут исчезать везде, где наша шумовая текстура будет нулевой.

Длина волн ряби управляется еще одним входом с названием *wavelength*, и наш нод *Ripples* (Пульсации) будет также иметь входной сокет для координат.

Четвертый и последний вход, называемый *direction* - вектор, который контролирует ориентацию наших элементарных волн. Может быть установлено вручную пользователем, но при желании, может быть соединён с нодом *normal*, который предоставляет простой способ манипулировать направлением с помощью мыши.

Окончательная настройка нодов, которая объединяет все это, показана на скриншоте редактора нодов:



Скрипт для нода прост; после нескольких необходимых операций импорта мы определим многочисленные входные сокеты и наш единственный выходной сокет.

```
from Blender import Node
from math import cos
from Blender.Mathutils import Vector as vec
```

```
class Ripples(Node.Scripted):
    def __init__(self, sockets):
        sockets.input = [
            Node.Socket('amplitude', val= 1.0,
                        min = 0.001, max = 1.0),
            Node.Socket('wavelength', val= 1.0,
                        min = 0.01, max = 1000.0),
            Node.Socket('direction', val= [1.0,0.0,0.0]),
            Node.Socket('Coords', val= 3*[1.0])]
```

```
sockets.output = [Node.Socket('Normal',
                               val = [0.0,0.0,1.0])]
```

```
def __call__(self):

    norm = vec(0.0,0.0,1.0)

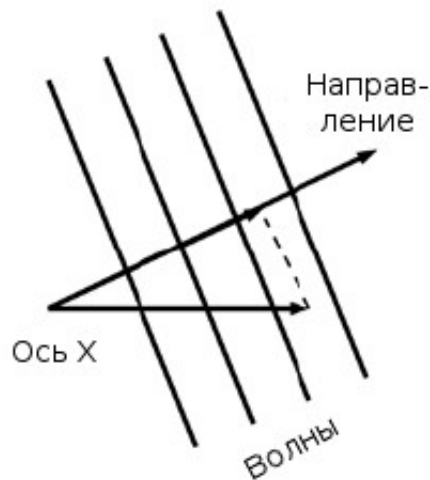
    p = vec(self.input.Coords)
    d = vec(self.input.direction)
    x = p.dot(d)*self.input.wavelength
    norm.x=-self.input.amplitude*cos(x)

    n = norm.normalize()

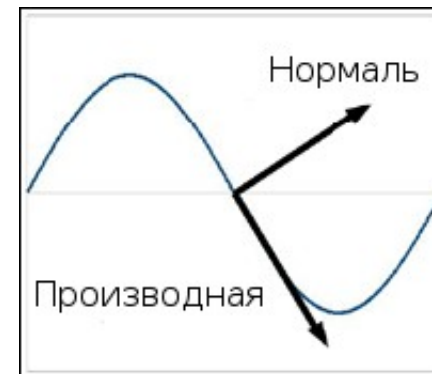
    self.output.Normal = n*.01
```

```
__node__ = Ripples
```

Снова, вся реальная работа выполняется в функции `__call__()` (выделено в предыдущем куске кода). Мы сначала определяем сокращения  $p$  и  $d$  для векторов координат и направления соответственно. Наши элементарные волны - функции синуса и позиция на этой синусоиде определяется проекцией позиции на вектора направления. Эта проекция вычисляется скалярным произведением - операция предоставлена методом `dot()` объекта *Vector*.

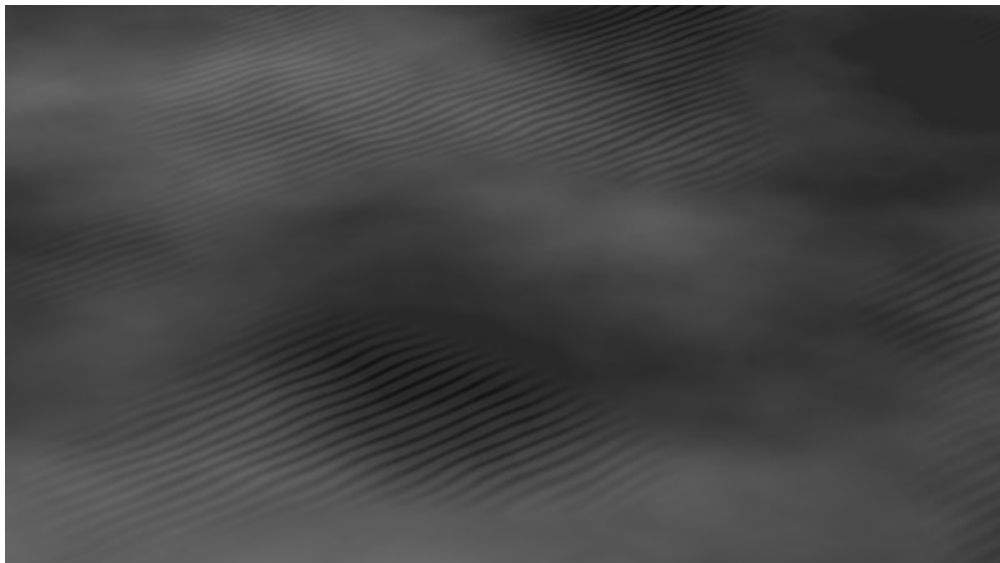


Затем, проекция умножается на длину волны. Если бы мы вычислили синус, у нас была бы высота нашей волны. Но нас, тем не менее, интересует не высота, а нормаль. Нормаль всегда направлена вверх и перемещается вместе с нашей синусоидальной волной (смотри следующую диаграмму). Можно показать, что эта нормаль - вектор с z-компонентой 1.0 и x-компонентой, равной отрицательной производной функции синуса, то есть, минус косинус. Скрипт (*ripples.py*) и пример настройки нодов доступны как файл *ripples.blend*.



В нодовой сети, которую мы показывали раньше, Вы могли обратить внимание, что вместо связи нода геометрии непосредственно с нашим нодом *ripples*, мы добавили второй нод текстуры, и скомбинировали этот нод с вводом геометрии сложив с масштабированным выходом *normal* текстурного нода. Мы могли бы смешать с некоторым шумом в ноду *ripples* непосредственно, но этим способом мы даем значительно больше управления пользователю над типом и количеством шума, который он хочет добавить (если хочет). Это - обычная модель: ноды должны разрабатываться по возможности простыми, чтобы облегчить их использование многократно с различными настройками.

Эти пульсации не были предназначены быть анимированными, но в следующем разделе мы разработаем нод, который это сможет.



## Капли - анимированные Pynodes

Множество узоров не являются статическими, а изменяются во времени. Одним из примеров являются пульсации, сформированные каплями, падающими в пруд. Блендер представляет параметры времени рендера, такие как, например, стартовый кадр, частота кадров, и текущий кадр, так что у нас есть много зацепок, чтобы сделать наши Pynodes зависимыми от времени. Мы увидим как использовать эти зацепки в скрипте, который генерирует рисунок капель. Узор, который изменяется достоверно, имеет сходство с расширяющимися волнами, вызванными каплями, падающими в пруд. На пути мы также приобретём несколько полезных хитростей, чтобы ускорить вычисления, сохраняя результаты дорогих вычислений в самом Pynode, чтобы позже многократно их использовать.

### Параметры времени рендера

Наиболее важные параметры рендера при работе с изменяющимися во времени вещами - текущий номер кадра и частота кадров (количество кадров в секунду). Эти параметры

предусмотрены сгруппированными вместе, в виде контекста рендера в модуле *Scene*, большинство через вызовы функций, некоторые как переменные:

```
scn                = Scene.GetCurrent()
context            = scn.getRenderingContext()
current_frame      = context.currentFrame() #Текущий кадр
start_frame        = context.startFrame()   #Начальный кадр
end_frame          = context.endFrame()     #Конечный кадр
frames_per_second  = context.fps            #Частота
                                                         #кадров, fps
```

Теперь, с этой информацией, мы можем вычислить время, или абсолютное, или относительно стартового кадра:

```
absolute_time = current_frame/float(frames_per_second)
relative_time = (current_frame-start_frame)/ \
                float(frames_per_second)
```

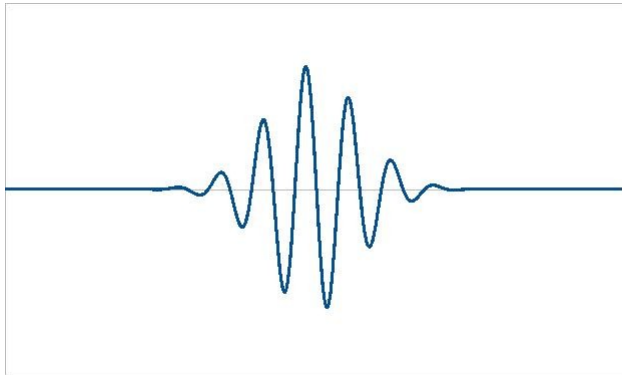
Заметьте преобразование во *float* (число с плавающей точкой) в знаменателе (выделено). Этим способом мы гарантируем, чтобы деление рассматривалось как операция с плавающей точкой. Не строго необходимо, поскольку fps возвращается с типом плавающей точки, но множество людей считают частоту кадров как некоторую целую величину, например, 25 или 30. Тем не менее, так бывает не всегда (например, кодировка NTSC использует дробную частоту кадров), так что мы лучше сделаем это явно. Также заметьте, что мы не можем покончить с этим делением, в противном случае, когда люди захотят изменить своё решение о выбранной частоте кадров, скорость анимации должна измениться.

### Всё, что выглядит хорошо — это хорошо

Точно имитировать то, как выглядят пульсации, вызванные падением капелек, может показаться трудным, но это просто, хотя и немного запутано. Читатели, интересующиеся базовой математикой, могут проверить какие-нибудь ссылки (например, <http://en.wikipedia.org/wiki/Wave>). Нашей целью, тем не менее, не является моделирование реального мира с максимально возможной точностью, а обеспечение художника текстурой, которая выглядит хорошо и управляется так, чтобы текстуру можно было применить даже в нереалистичных ситуациях.



Так, вместо определения скорости, с которой движется волна в зависимости от чего-нибудь, например, вязкости воды, мы делаем скорость в виде регулируемого входа в наш Pynode. То же самое для высоты и ширины волн, и показателя, с которым высота волн уменьшается по мере расширения. В основном, мы аппроксимируем наш небольшой пакет пульсаций, его расхождение наружу из точки падения капельки, функцией косинуса, умноженной на экспоненциальную функцию и показатель торможения. Это снова может показаться опасным погружением в математику, но может легко быть визуализировано:



Для того, чтобы вычислить высоту в любой позиции  $x$ ,  $y$  на нашей текстуре, вышеуказанное можно осуществить следующим образом:

```
position_of_maximum=speed*time
damping = 1.0/(1.0+dampf*position_of_maximum)
distance = sqrt((x-dropx)**2+(y-dropy)**2)
height = damping*a*exp(-(distance-
position_of_maximum)**2/c) * \
    cos(freq*(distance-position_of_maximum))
```

Здесь, *dropx* и *dropy* - позиция ударившей капли, *a* - наш регулируемый параметр высоты.

Эффекты от многих брошенных капель в разное время и в разных позициях можно просто вычислить суммированием результирующих высот.

## Хранение дорогостоящих результатов для многократного использования

Единственная капля - это, конечно, не дождь, так что мы хотели бы видеть сложенные эффекты от множества случайных капель. Следовательно, мы должны выбирать произвольные позиции и время ударов для стольких капелек, сколько мы хотели бы симитировать.

Мы должны были бы делать это каждый раз при вызове метода `__call__()` (то есть, для каждого видимого пикселя в нашей текстуре). Тем не менее, это было бы огромными тратами процессорных сил, поскольку вычисление множества случайных чисел и получение и возврат памяти для, возможно, многих капель дорого.

К счастью, мы можем сохранить эти результаты в качестве экземпляров переменных нашего Pynode. Конечно, мы должны быть достаточно осторожными, чтобы проверять, что никакие входные параметры не были изменены между вызовами `__call__()` и предпринять соответствующие меры, если они изменились. Общая картина будет выглядеть следующим образом:

```
class MyNode(Node.Scripted):

    def __init__(self, sockets):
        sockets.input = [Node.Socket('InputParam',
                                     val = 1.0)]
        sockets.output = [Node.Socket('OutputVal',
                                     val = 1.0)]
        self.InputParam = None
        self.Result = None

    def __call__(self):
        if self.InputParam == None or \
            self.InputParam != self.input.InputParam :
            self.InputParam = self.input.InputParam
            self.Result = интенсивные_вычисления ...
            self.output.OutputVal = другие_вычисления ...
```

Этот образец работает, только если входной параметр изменяется редко, например, только если его изменяет пользователь. Если вход изменяется с каждым пикселем, поскольку входной сокет подключен к выходу другого нода - схема с запоминанием, наоборот, будет дороже по времени вместо какой-либо экономии.

## Вычисление нормалей

Нашей целью будет сгенерировать волновой узор, который можно использовать в качестве нормали. Так что нам нужен некий способ получить нормаль из рассчитанных высот. Блендер не предоставляет нам такого преобразующего нода для материалов, так что мы должны разработать схему самостоятельно.



В противовес нодам материалов, ноды текстур Блендера обеспечивают преобразующую функцию, называемую 'Value to Normal' (величина в нормаль), которая доступна в нодовом редакторе текстур из меню **Add|Convertor|Value to Normal**.

Теперь, как и в случае ряби, мы могли бы, в принципе, вычислить также точную нормаль для нашей капли дождя, но, вместо движения по математическому пути, мы снова применяем метод, используемый многими встроенными текстурами шума для вычисления нормалей, который работает независимо от основной функции.

Пока мы можем оценивать функцию в трех точках:  $f(x,y)$ ,  $f(x+nabla,y)$ , и  $f(x,y+nabla)$ , мы можем оценить направление нормали в  $x,y$ , изучая наклон нашей функции в направлениях  $x$  и  $y$ . Нормаль поверхности будет вектором, перпендикулярным к плоскости, определенной этими двумя наклонами. Мы можем взять любую малую величину для  $nabla$ , чтобы попробовать с ней, и если это не будет выглядеть хорошо, мы можем её уменьшить.

## Собираем всё это вместе

Взяв все эти идеи из предыдущих параграфов, мы можем приготовить следующую программу для нашего Pynode Raindrops (с опущенными операторами *import*):

```
class Raindrops(Node.Scripted):
    def __init__(self, sockets):
        sockets.input = [
            Node.Socket('Drops_per_second' ,
                        val = 5.0, min = 0.01, max = 100.0),
            Node.Socket('a',val=5.0,min=0.01,max=100.0),
            Node.Socket('c',val=0.04,min=0.001,max=10.0),
```

```
Node.Socket('speed',val=1.0,min=0.001, max=10.0),
Node.Socket('freq',val=25.0,min=0.1, max=100.0),
Node.Socket('dampf',val=1.0,min=0.01, max=100.0),
Node.Socket('Coords', val = 3*[1.0])]
```

```
sockets.output = [
    Node.Socket('Height', val = 1.0),
    Node.Socket('Normal', val = 3*[0.0])]
```

```
self.drops_per_second = None
self.ndrops = None
```

Код инициализации определяет множество входных сокетов помимо координатного. *Drops\_per\_second* (капель в секунду) должен быть самочитаемым. *a* и *c* - общая высота и ширина пульсаций,двигающихся наружу из точки удара. *speed* и *freq* определяют, как быстро наши пульсации двигаются и насколько близко волны друг к другу. То, как быстро высота волн уменьшается во время пути наружу, определяет *dampf*.

Мы также определяем два выходных сокета: *Height* будет содержать рассчитанную высоту и *Normal* будет содержать соответствующую нормаль в этой же точке. *Normal* - это то, что Вы должны обычно использовать для получения поверхностного эффекта распространения, но рассчитанная высота может быть полезной, например, чтобы смягчить величину отражательной способности поверхности.

Инициализация заканчивается с определением некоторых переменных экземпляра, которые будут использованы, чтобы определить, нужно ли нам вычислять позицию падения капли заново, как мы увидим в определении функции `__call__()`.

Определение функции `__call__()` начинается с инициализации множества локальных переменных. Одно примечательное место - то, где мы установили произвольное семя, используемое функциями модуля *Noise* (выделено в следующем коде). Таким образом, мы убеждаемся, что всякий раз, когда мы пересчитываем точки удара, мы получаем повторяемые результаты, что если мы установили бы количество капель в секунду сначала на десять, а позже на двадцать, и, затем вернулись к десяти, сгенерированный узор будет тем же. Если Вы хотели бы изменить это, Вы могли бы добавить

дополнительный входной сокет, который нужно использовать как вход для функции *setRandomSeed()*:

```
def __call__(self):

    twopi = 2*pi

    col = [0,0,0,1]
    nor = [0,0,1]
    tex_coord = self.input.Coords
    x = tex_coord[0]
    y = tex_coord[1]

    a = self.input.a
    c = self.input.c
    Noise.setRandomSeed(42)

    scn          = Scene.GetCurrent()
    context       = scn.getRenderingContext()
    current_frame = context.currentFrame()
    start_frame   = context.startFrame()
    end_frame     = context.endFrame()
    frames_per_second = context.fps
    time = current_frame/float(frames_per_second)
```

Следующим шагом нужно определить, должны ли мы вычислять позиции точек удара капель заново. Это необходимо, только если величина входного сокета *drops\_per\_second* была изменена пользователем (Вы могли бы соединить этот вход с некоторым другим нодом, который будет изменять эту величину на каждом пикселе, но это плохая идея), или когда стартовый или конечный кадр анимации изменились, как эти влияния количества капель мы должны вычислять. Этот тест выполняется на выделенной строке следующего кода сравнением вновь полученных величин с сохранёнными в переменных экземпляра:

```
drops_per_second = self.input.Drops_per_second
# вычисление числа капель для генерации
# в период анимации
ndrops = 1 + int(drops_per_second * \
    (float(end_frame) - start_frame+ 1)/ \
    frames_per_second )

if self.drops_per_second != drops_per_second \
```

```
or self.ndrops != ndrops:
    self.drop = [ (Noise.random(), Noise.random(),
        Noise.random() + 0.5) for i in range(ndrops)]
    self.drops_per_second = drops_per_second
    self.ndrops = ndrops
```

Если мы должны вычислить позиции капель заново, мы назначаем список кортежей в переменную экземпляра *self.drop*, каждый из которых состоит из координат x и y позиции капли и случайного размера капли, от которой будет зависеть высота волн.

Строк оставшейся части полностью выполняются всякий раз при вызове *\_\_call\_\_()*, но выделенная строка показывает значимую оптимизацию. Поскольку капли, которые еще не упали в текущем кадре, не привносят изменений высоты, мы исключаем их из вычисления:

```
speed=self.input.speed
freq=self.input.freq
dampf=self.input.dampf

height = 0.0
height_dx = 0.0
height_dy = 0.0
nabla = 0.01
for i in range(1+int(drops_per_second*time)):
    dropx,dropy,dropsz = self.drop[i]
    position_of_maximum=speed*time- \
        i/float(drops_per_second)
    damping = 1.0/(1.0+dampf*position_of_maximum)
    distance = sqrt((x-dropx)**2+(y-dropy)**2)
    height += damping*a*dropsz* \
        exp(-(distance-position_of_maximum)**2/c)* \
        cos(freq*(distance-position_of_maximum))
    distance_dx = sqrt((x+nabla-dropx)**2+ \
        (y-dropy)**2)
    height_dx += damping*a*dropsz* \
        exp(-(distance_dx-position_of_maximum)**2/c) \
        * cos(freq*(distance_dx-position_of_maximum))
    distance_dy = sqrt((x-dropx)**2+ \
        (y+nabla-dropy)**2)
    height_dy += damping*a*dropsz* \
        exp(-(distance_dy-position_of_maximum)**2/c) \
        *cos(freq*(distance_dy-position_of_maximum))
```



В предыдущем коде мы действительно вычисляем высоту в трех различных позициях, чтобы получить возможность аппроксимировать нормаль (как объяснено раньше). Эти величины используются в следующих строках, чтобы определить x и y компоненты нормали (z компонента установлена в единицу). Сама рассчитанная высота делится на количество капель (таким образом, средняя высота не изменится при изменении количества капель) и на общий коэффициент масштабирования *a*, который может быть задан пользователем прежде, чем будет подсоединён выходной сокет (выделено):

```
nor[0]=height-height_dx
nor[1]=height-height_dy
```

```
height /= ndrops * a
self.output.Height = height
```

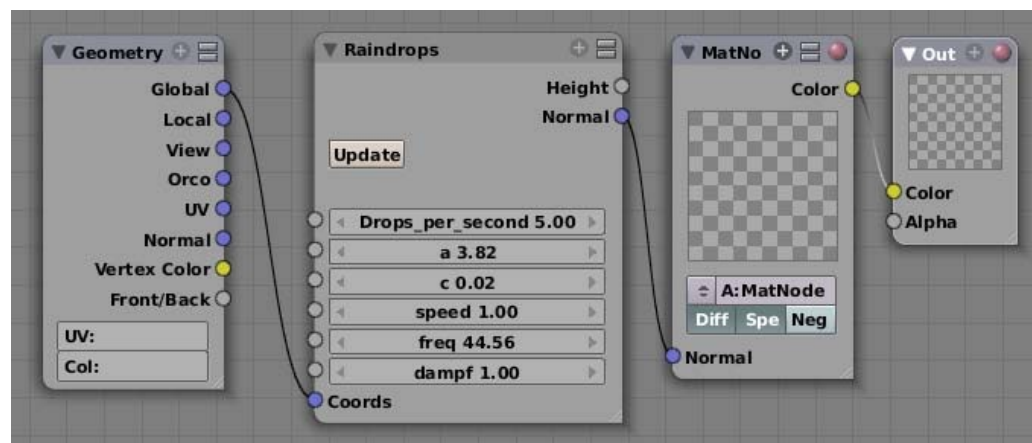
```
N = (vec(self.shi.surfaceNormal)+0.2 * \
      vec(nor)).normalize()
self.output.Normal= N
```

```
__node__ = Raindrops
```

Рассчитанная нормаль затем добавляется к поверхностной нормали того пикселя, который мы вычисляем, таким образом, волны будут все еще хорошо выглядеть на искривленной поверхности, и нормируется перед назначением её в выходной сокет. Последняя строка как обычно определяет значимое имя для этого Pynode. Полный код и пример настройки нодов доступны как *raindrops.py* в файле *raindrops.blend*. Пример кадра из анимации показан на следующем скриншоте:



Пример нодовой сети показан на следующем скриншоте:




## Грозовой перевал — материал, зависимый от наклона

В Блендере очень просто генерировать фрактальную местность (просто добавьте плоскость, перейдите в режим *редактирования*, выберите всё, затем несколько раз подразделите фрактально  $W > 3$ ).

Если Вы хотите чего-то большего, Вам в помощь существует несколько отлично разработанных скриптов (посмотрите, например, [http://sites.google.com/site/androcto/Home/python-scripts/ANTLandscape\\_104b\\_249.py](http://sites.google.com/site/androcto/Home/python-scripts/ANTLandscape_104b_249.py)). Но как Вы наложите текстуры на такую местность? В этом примере мы изучим метод, выбирающий между различными входами материала, основываясь на величине угла наклона поверхности, которую мы затеняем. Это позволит нам создать эффект, при котором очень крутые откосы обычно лишены зелени, даже если они оказались ниже линии деревьев. В комбинации с высотой-зависимым материалом мы сможем затенить гористую местность достаточно убедительно.

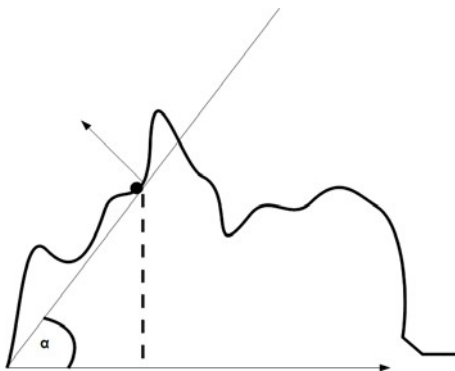
#### Уменьшение времени вычислений:



Rynodes в вычислительном отношении затратны, так как они вызываются для каждого видимого пикселя. Умное программирование может иногда уменьшить количество необходимых вычислений, но если требуется дальнейшее ускорение, может помочь *компилятор-на-ленту (just-in-time compiler)*. **psyco** является таким компилятором и, мы столкнемся с ним в последней главе, где мы будем применять его на Rynodes и посмотрим, имеет ли он какой-либо заметный эффект.

## Определение уклона

Уклон может быть определен как угол между плоскостью пола и касательной к поверхности в интересующей нас точке.



Поскольку мы принимаем нашу (воображаемую) плоскость пола вытянутой горизонтально вдоль осей  $x$  и  $y$ , этот угол полностью определяется  $z$ -компонентой нормали к поверхности в этой же точке. Теперь мы можем вычислить этот угол точно (это  $\arcsin(z/\sqrt{x^2+y^2})$ ), но, как художникам, нам, возможно, в любом случае захочется иметь некоторое дополнительное управление, таким образом мы просто берем нормализованную  $z$ -компоненту нормали к поверхности и изменяем эту выходную интенсивность с помощью любого нода `color ramp`, который нам нравится. В пределах Rynode, нормаль поверхности является легко доступным вектором: `self.input.shi.surfaceNormal`. Однако есть препятствие...

## Мировое пространство против пространства камеры

Нормаль поверхности, которую мы имеем в распоряжении, определена в пространстве камеры. Это означает, что, например, когда нормаль поверхности указывает прямо в камеру, она определена как  $(0, 0, -1)$ . В данный момент мы хотим определить нашу нормаль поверхности в мировом пространстве. Нормаль, которая указывает прямо вверх, например, должна иметь величину  $(0, 0, 1)$  независимо от позиции или наклона камеры (в конце концов, растительность на горном склоне обычно не изменяется с изменением угла камеры). К счастью, мы можем провести преобразование из **пространства камеры** в **мировое пространство**, взяв матрицу камеры мирового пространства и умножив нормаль поверхности на вращающую часть этой матрицы. Результирующий код выглядит похожим на это:

```
class Slope(Node.Scripted):
    def __init__(self, sockets):
        sockets.output = [Node.Socket('SlopeX', val = 1.0),
                          Node.Socket('SlopeY', val = 1.0),
                          Node.Socket('SlopeZ', val = 1.0),]

        self.offset = vec([1,1,1])
        self.scale = 0.5
```

Заметьте, что код инициализации не определяет входных сокетов. Мы получим нормаль поверхности в позиции пикселя, который мы затеняем, из входа `shader` (выделено в следующей части кода). Мы определяем три отдельных выходных сокета для  $x$ ,  $y$ , и  $z$  компонент наклона для удобства использования в нодовой сети. Так

как мы, по большей части, используем именно z-компоненту наклона, то если мы будем иметь её доступной в отдельном сожете, нам не придётся использовать для её извлечения из вектора дополнительный нод обработки вектора.

```
def __call__(self):  
  
    scn=Scene.GetCurrent()  
    cam=scn.objects.camera  
    rot=cam.getMatrix('worldspace').rotationPart(  
        ).resize4x4();  
    N = vec(self.shi.surfaceNormal).normalize(  
        ).resize4D() * rot  
    N = (N + self.offset ) * self.scale  
    self.output.SlopeX=N[0]  
    self.output.SlopeY=N[1]  
    self.output.SlopeZ=N[2]
```

```
__node__ = Slope
```

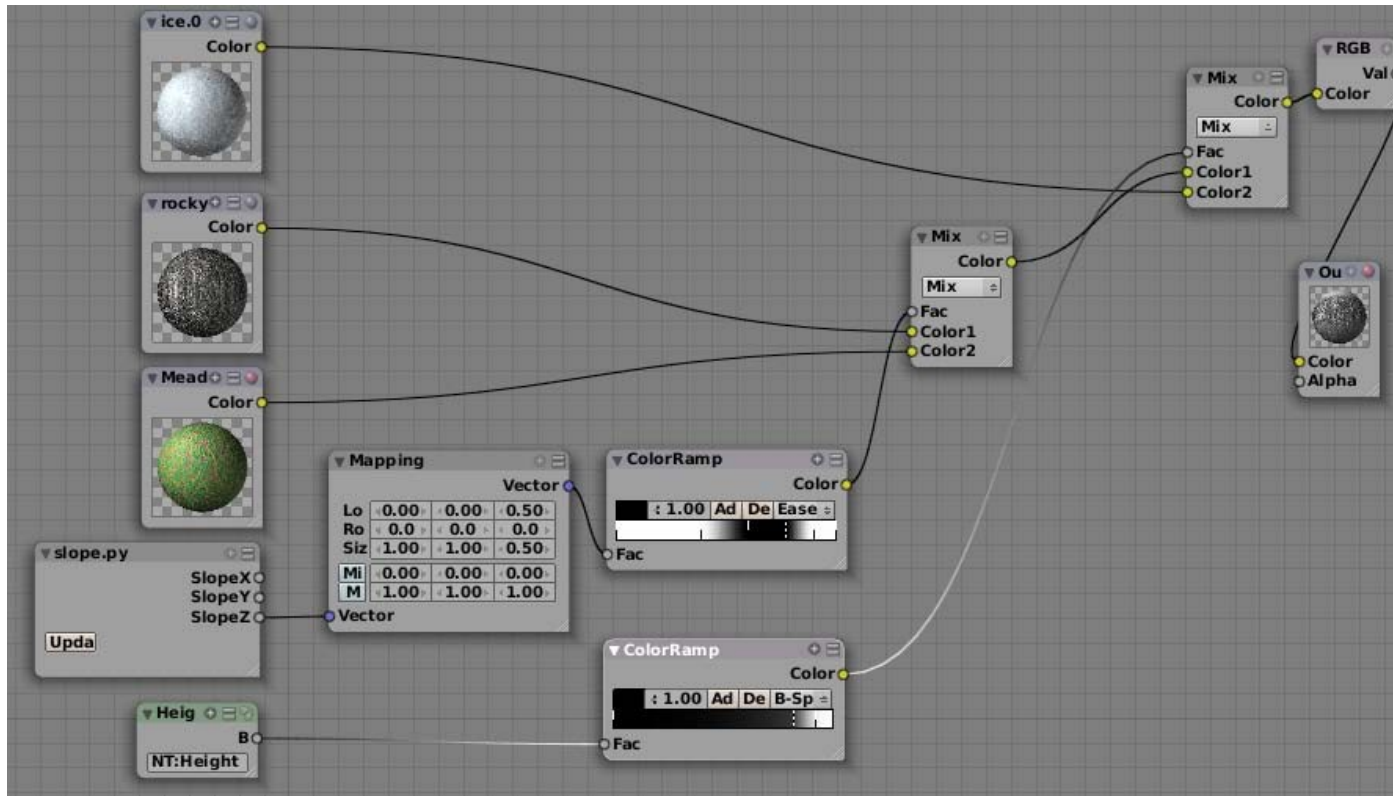
Преобразование из пространства камеры в мировое пространство делается в строке, которая ссылается на нормаль поверхности (выделено). Ориентация зависит только от вращения, следовательно, мы извлекаем вращающую часть матрицы преобразования камеры до того, как мы умножим нормаль поверхности на неё. Так как нормализованный результат может указывать вниз, мы заставляем z-компоненту находиться в диапазоне [0, 1], прибавляя 1 и умножая на 0.5. Полный код доступен как *slope.py* в файле *slope.blend*.

Есть одна важная вещь, о которой нужно отдавать себе отчет: нормаль поверхности, которую мы здесь используем, не интерполируется, и, следовательно, она одинаковая везде вдоль поверхности единственной грани, даже если был установлен атрибут грани *smooth*. Это не должно быть проблемой в тонко подразделенном ландшафте, где вход наклона не используется непосредственно, тем не менее, это отличается от того, что Вы могли ожидать. В текущей реализации Pynodes это ограничение трудно, если не совсем невозможно, преодолеть.

Следующая иллюстрация показывает возможный пример.



Эффекты, показанные выше, были реализованы объединением различных материалов в нодовой сети, показанной на следующем скриншоте. Эта настройка также доступна в *slope.blend*. Два нижних материала смешивались с использованием нашего наклоно-зависимого нода, и результирующий материал смешивается с верхним материалом, основанным на Pynode, который вычисляет высоту.



## Мыльные пузыри — шейдер, зависимый от точки зрения

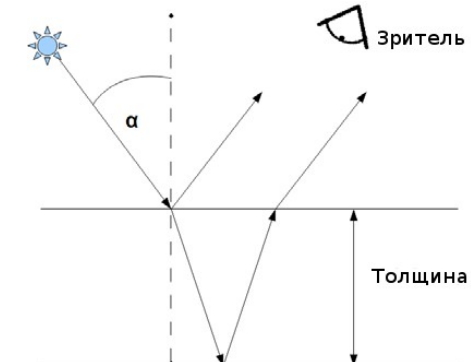
У некоторых материалов вид меняется в зависимости от угла, под которым мы на них смотрим. Перья птиц, некоторые причудливые автомобильные краски, нефтяные разливы на воде, и мыльные пузыри - вот несколько примеров. Этот феномен изменения цветов известен как радужность (iridescence). Если мы хотим осуществить нечто подобное, нам нужен доступ к вектору вида и нормали поверхности. В нашем шейдере мыльного пузыря мы увидим один из способов сделать это.

*Сначала немного математики:* Почему это мыльные пузыри показывают все эти различные цвета? Мыльные пузыри - это в основном искривлённые водяные плёнки (с небольшим количеством

мыла), и свет отражается от поверхности раздела между воздухом и водой. Следовательно, падающий луч частично отражается, когда он попадает на внешнюю поверхность пузыря, и отражается снова, когда он достигает внутренней поверхности. Следовательно, отраженный свет, который попадает в глаз — является суммой света, прошедшего различные расстояния; часть его прошла дополнительное расстояние в две толщины мыльного пузыря.

Теперь учтём, что свет ведёт себя подобно волне, а волны, которые интерферируют, могут или ослаблять или усиливать друг друга в зависимости от их фазы, и поэтому, два световых луча, прошедшие расстояния, разница которых не кратна в точности их длине волны, гасят друг друга. В результате, белый свет (континуум, совокупность цветов), отраженный мыльным пузырём с толщиной, равной половине длины волны некоторого специфического цвета,

покажет только этот единственный цвет, поскольку все остальные цвета подавлены, так как они "не соответствуют" должным образом толщине между внутренней и внешней поверхностью. (Существует гораздо больше информации о мыльных пузырях. Для большей и более точной информации вот ссылка: [http://ru.wikipedia.org/wiki/Мыльные\\_пузыри.](http://ru.wikipedia.org/wiki/Мыльные_пузыри.))



Теперь мы знаем, что расстояние пройденное между двумя отражающими поверхностями, определяет цвет, который мы воспринимаем, мы можем также понять, почему цвет будет варьироваться в мыльном пузыре. Первым фактором является кривизна пузыря. Пройденное расстояние будет зависеть от угла между падающим светом и поверхностью: чем меньше угол, тем более длинное расстояние свет должен пройти между поверхностями. Угол падения изменяется, так как поверхность кривая, и таким образом, изменяется расстояние, и, следовательно, цвет. Второй причиной изменения цвета является неравномерность поверхности: незначительные изменения из-за тяжести или вихри, вызванные воздушными течениями или перепадами температур, также вызывают различия в цвете.

Вся эта информация переводится в удивительно короткую часть кода (полный код доступен как *irridescence.py* в файле *irridescence.blend* вместе с примером нодовой сети).

Наряду с координатами, у нас есть ещё два входных сокета — один для толщины водяной плёнки и один для вариаций. Вариации будут добавляться к толщине и этот сокет может быть присоединён к текстурному ноду, чтобы генерировать вихри и тому подобное. У нас есть единственный выходной сокет для рассчитанного расстояния:

```
class Irridescence(Node.Scripted):
    def __init__(self, sockets):
        sockets.input = [
            Node.Socket('Coords', val= 3*[1.0]),
            Node.Socket('Thickness', val=275.0,
                        min=100.0, max=1000.0),
            Node.Socket('Variation', val=0.5, min=0.0,
                        max=1.0)]

        sockets.output = [Node.Socket('Distance',
                                       val=0.5, min=0.0, max=1.0)]
```

Вычисления отраженного цвета начинается с получением списка всех ламп на сцене, так как мы хотим вычислить угол падающих световых лучей. Сейчас, мы принимаем во внимание вклад только первой лампы, которую мы нашли. Тем не менее, более полная реализация должна рассматривать все лампы, и может быть, даже их цвет. Для наших вычислений мы должны убедиться, что нормаль

поверхности  $N$  и вектор падения света  $L$  находятся в одном и том же пространстве. Так как предоставляемая нормаль поверхности будет в пространстве камеры, мы должны трансформировать этот вектор матрицей преобразования камеры, как мы это делали для нашего наклоно-зависимого шейдера (выделено в следующем куске кода):

```
def __call__(self):

    P = vec(self.input.Coords)
    scn=Scene.GetCurrent()
    lamps = [ob for ob in scn.objects if
              ob.type == 'Lamp']

    lamp = lamps[0]

    cam=scn.objects.camera
    rot=cam.getMatrix('worldspace').rotationPart(
                                                ).resize4x4();

    N = vec(self.shi.surfaceNormal).normalize(
                                                ).resize4D() * rot

    N = N.negate().resize3D()
    L = vec(lamp.getLocation('worldspace'))
    I = (P - L).normalize()
```

Затем, мы вычисляем угол между нормалью поверхности и вектором падения (VecT - псевдоним для функции *Mathutils.angleBetweenVecs()*), и используем этот угол падения, чтобы вычислить угол между нормалью поверхности *внутри* водяной плёнки, так как он определяет расстояние прохождения света. Мы используем **закон Снелла** для его вычисления, а для показателя преломления водной плёнки возьмём 1.31. Расчет расстояния после этого - вопрос простой тригонометрии (выделено ниже):

```
angle = VecT(I,N)

angle_in = pi*angle/180
sin_in = sin(angle_in)
sin_out = sin_in/1.31
angle_out = asin(sin_out)

thickness = self.input.Thickness + \
            self.input.Variation
distance = 2.0 * (thickness / cos (angle_out))
```



Рассчитанное расстояние равняется длине волны цвета, который мы воспримем. Тем не менее, Блендер работает не с длинами волн, а с цветами RGB, так что нам всё еще нужно преобразовать эту длину волны в кортеж (R, G, B), который представляет тот же цвет. Это можно было бы сделать посредством применения некоей спектральной формулы (смотрите, например, здесь: <http://www.philiplaven.com/p19.html>), но, может быть, будет даже более универсальным вариантом масштабировать это рассчитанное расстояние, и использовать его как вход для цветовой полосы (color band). Таким образом мы можем воспроизвести не-физически точную радужность (если захотим):

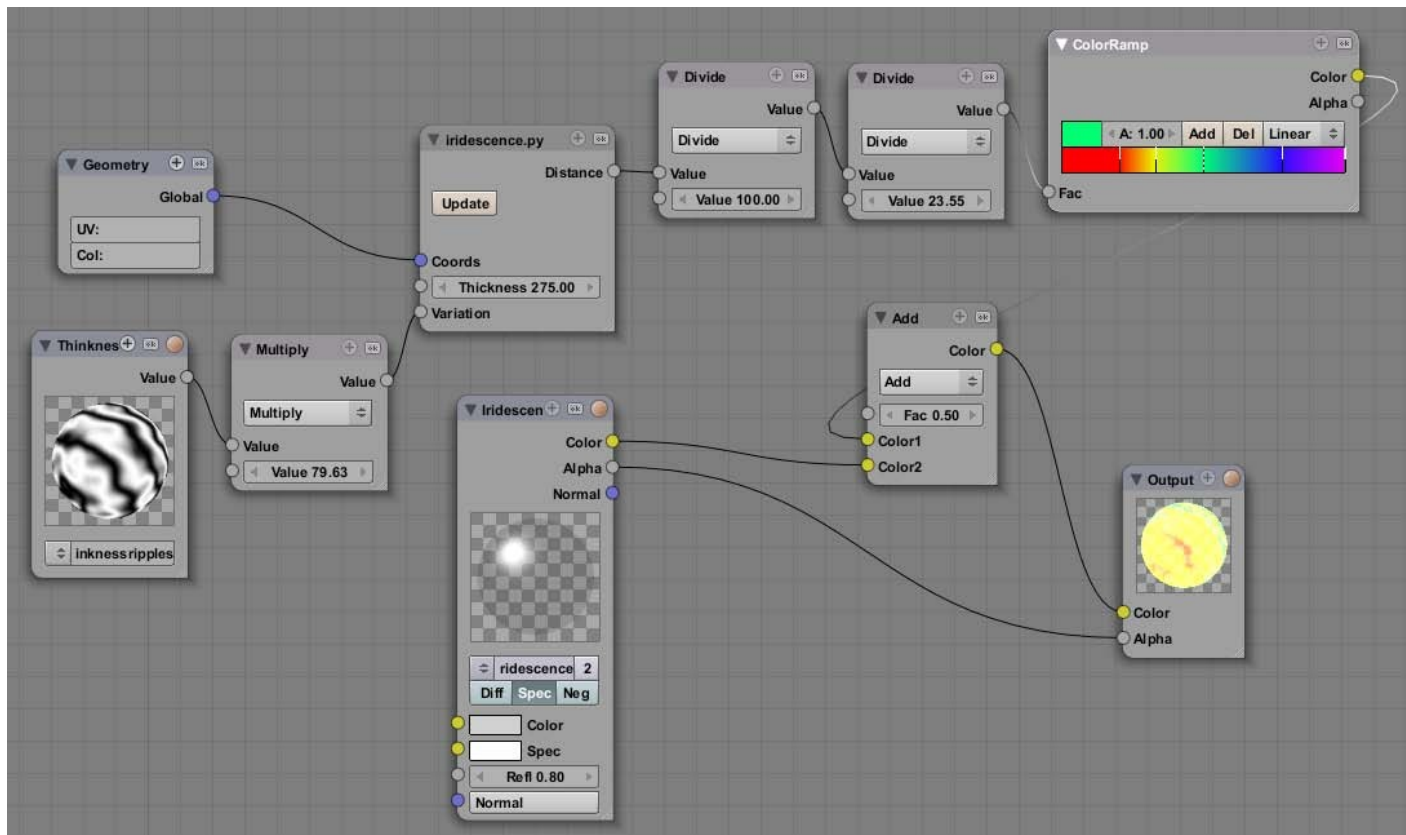
```
self.output.Distance = distance
```

Чтобы использовать этот Rynode, нужно иметь в виду некоторые моменты. Сначала, убедитесь, что рассчитанный цвет влияет только на цвет specular материала мыльного пузыря, в противном случае всё покажется вымытым.

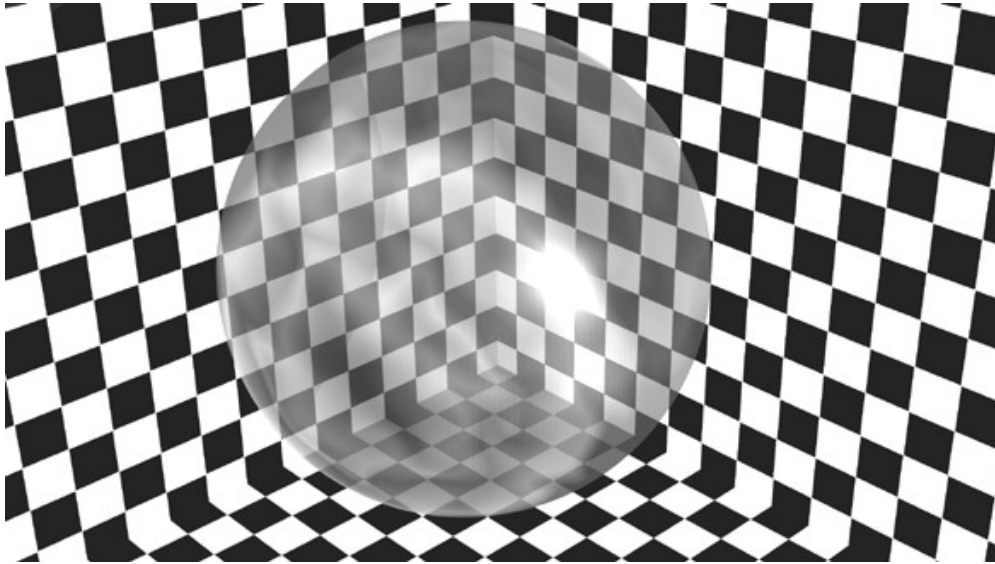
Кроме того, важно добавлять некоторое изменение к толщине слоя, так как никакой реальный мыльный пузырь не имеет точной однородной толщины. Выбор текстуры шума может привести к очень различному виду. В следующем примере нодовой сети мы добавили на вход немного шумовой текстуры wood, чтобы получать вихревые полосы, часто обнаруживаемые на мыльных плёнках.

Наконец, сделайте материал мыльной плёнки очень прозрачным, но с высокой отражательной способностью (specular). Экспериментируйте с величинами, чтобы добиваться точного эффекта, и примите во внимание настройку освещения. Пример, показанный на иллюстрации – пробный, чтобы получить некий результат в черно-белом представлении, и, следовательно, не

реалистичен, но сеть в файле примера *iridescence.blend* настроена производить красочный приятный результат при рендере.



Использование color ramp и текстуры шума показано на предыдущем скриншоте, куда мы добавили несколько нодов деления, чтобы масштабировать наше расстояние в диапазон в пределах [0,1], который можно использовать как вход для color ramp:



## **Итог**

В этой главе мы увидели, что отсутствие компилируемого шейдерного языка в Блендере не препятствует использованию в нём спроектированных заказных узоров и шейдеров. Pynodes - встроенная часть нодовой системы Блендера, и мы увидели как использовать их для создания эффектов, от простых цветных узоров до довольно сложных анимированных волн. В частности, мы узнали:

- Как писать Pynodes, которые создают простые цветные узоры
- Как писать Pynodes, которые производят узоры с нормальями
- Как писать анимированные Pynodes
- Как писать материалы, зависимые от высоты и наклона
- Как создавать шейдеры, которые реагируют на угол падающего света

В следующей главе мы посмотрим на автоматизацию процесса рендера в целом.



## Рендеринг (визуализация) и обработка изображений

В предшествующих главах мы рассматривали в основном аспекты скриптования индивидуальных компонентов, составляющих сцену Блендера, такие как, например, меши, лампы, материалы, и так далее. В этой главе мы взглянем на процесс визуализации в целом. Мы будем автоматизировать процесс рендера, объединять различными способами результирующие изображения, и даже превратим Блендер в специализированный веб-сервер.

В этой главе Вы узнаете как:

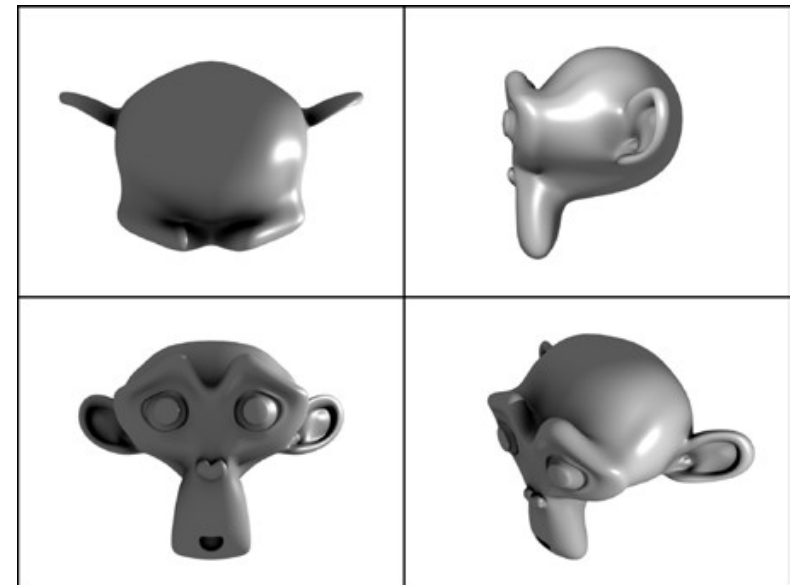
- Автоматизировать процесс рендера
- Создавать множество видов для презентации продукта
- Создавать билборды из сложных объектов
- Манипулировать изображениями, в том числе результатами рендера, используя библиотеку обработки изображений Python Imaging Library (PIL)
- Построить сервер, который создает изображения по требованию, которые могут быть использованы как вопросы в CAPTCHA

### **Различные виды - комбинирование множества направлений камеры**

Теперь Вы можете ожидать, что визуализация также может быть автоматизирована, и Вы совершенно правы. API Питона Блендера обеспечивает доступ почти ко всем параметрам процесса рендера, и позволяет Вам рендерить индивидуальные кадры так же, как анимацию. Это позволяет автоматизировать многие задачи, которые было бы скучно делать руками.

Предположим, что Вы создали объект, и хотите создать единственное изображение, которое показывает его с различных

углов. Вы могли бы отрендерить их отдельно и объединить во внешнем приложении, но мы напишем скрипт, который не только рендерит эти виды, но также объединяет их в единственном изображении, используя возможности манипуляции изображениями Блендера и внешний модуль, называемый PIL. Эффект, которого мы пытаемся достигнуть, изображен на иллюстрации Сюзанны, показывающей её со всех наилучших сторон.



Блендер является отличным средством, которое предоставляет Вам возможность не только моделировать, анимировать и настраивать рендер, но имеет также функциональность, необходимую для композиции. Одна из областей, которая не слишком выделяется, это "манипуляция изображениями". Блендер, конечно же, имеет окно редактора UV/Image, но оно разработано очень специфически для манипулирования UV-раскладками и для просмотра изображений, а не для работы с ними. Редактор нодов

также способен на изощрённую обработку изображений, но у него нет документированного API, так что его нельзя сконфигурировать из скрипта.

Конечно, Блендер не может делать всё, и, несомненно, он не пытается конкурировать с графическими пакетами, такими как **GIMP** ([www.gimp.org](http://www.gimp.org)), но некоторые встроенные функции обработки изображений были бы кстати. (Каждым изображением можно управлять на уровне пикселей, но это довольно медленный процесс для больших изображений, и нам по-прежнему придётся осуществлять высокоуровневую функциональность, такую, как например, альфа-смешивание или поворот изображений).

К счастью, мы из Питона можем иметь доступ к любому изображению, сгенерированному Блендером, а в Питоне довольно просто добавить дополнительные пакеты, которые обеспечивают нужную функциональность, и использовать их из наших скриптов. Единственным недостатком является то, что любой скрипт, который использует эти дополнительные библиотеки, не будет автоматически переносимым, так что пользователи должны будут сами удостовериться, что у них имеются нужные библиотеки.

**Python Imaging Library (PIL)**, библиотека, которую мы будем использовать, свободно доступна и просто устанавливается. Следовательно, это не должно стать проблемой для среднего пользователя. Тем не менее, возможно осуществить функциональность простой вставки (мы увидим ниже), просто используя модуль *Image* Блендера, мы предоставим её в полном коде минималистского модуля *pim*, в котором будет только необходимый минимум, чтобы иметь возможность использовать наш пример без необходимости устанавливать PIL. Эта независимость имеет цену: наша функция *paste()* - почти в 40 раз медленнее, чем такая же из PIL, и результирующее изображение может сохраняться только в формате TARGA (*.tga*). Но вы, наверное, и не заметите этого, так как Блендер может просто отлично отображать TARGA файлы. Полный код оснащен некоторой хитростью, позволяющей использовать модуль PIL (в случае, если он доступен), и наш заменяющий модуль в противном случае. (Это не показано в книге.)

### The Python Imaging Library (PIL) (Библиотека Питона формирования изображений)



PIL - это пакет с открытыми исходными текстами, свободно доступный на сайте <http://www.pythonware.com/products/pil/index.htm>. Он состоит из множества модулей Питона и основной библиотеки, который поставляется для Windows в скомпилированном виде (и его очень легко скомпилировать для Linux, или даже он может быть уже доступен в дистрибутиве). Просто следуйте за инструкциями на сайте, чтобы установить его (но не забывайте использовать правильную версию питона при установке PIL; если у Вас установлено более одной версии Питона, используйте для установки ту же, что использует Блендер).

### Схема кода - combine.py

Какие шаги мы должны предпринять, чтобы создать наше комбинированное изображение? Нам понадобится:

1. Создать камеры, если нужно.
2. Настроить и откалибровать камеры на предмет.
3. Отрендерить виды со всех камер
4. Объединить рендеренные изображения в единственную картинку.

Код начинается с импорта всех необходимых модулей. Из пакета PIL нам нужен модуль *Image*, но мы импортируем его под другим именем (*pim*), чтобы предотвратить столкновение с именем модуля *Image* Блендера, который мы также используем:

```
from PIL import Image as pim
import Blender
from Blender import Camera, Scene, Image, Object,
Mathutils, Window
import bpy
import os
```

Первая функция-утилита, с которой мы столкнёмся, это - *paste()*. Эта функция объединяет четыре изображения в одно. Изображения передаются как имена файлов, а результат сохраняется как *result.png*, если не задано другое имя файла. Мы принимаем все четыре изображения, чтобы иметь одинаковые размеры, которые мы определяем, открывая первый файл, как изображение PIL и анализируя атрибут размера *size* (выделено в следующем коде). Изображения будут разделены и разграничены небольшой линией с однотонным цветом. Её ширина и цвет жестко кодируется в переменных *edge* и *edgcolor*, хотя Вы могли бы решить передавать их как аргументы:

```
def paste(top, right, front, free, output="result.png") :
    im = pim.open(top)
    w,h= im.size
    edge=4
    edgcolor=(0.0,0.0,0.0)
```

Затем, мы создаем пустое изображение, достаточно большое, чтобы вместить все четыре входных изображения с соответствующими границами. Мы не рисуем никаких границ специально, а просто определяем новое изображение с однотонным цветом, на которое будем вставлять все четыре изображения с подходящим смещением:

```
comp = pim.new(im.mode, (w*2+3*edge, h*2+3*edge),
               edgcolor)
```

Мы уже открыли верхнее изображение, так что всё, что мы должны сделать - вставить его в верхнем левом квадранте нашего комбинированного изображения, сдвинув его как в горизонтальном, так и в вертикальном направлениях на ширину границы:

```
comp.paste(im, (edge, edge))
```

Вставка трёх остальных изображений следует за той же схемой: открыть изображение и вставить его в правильной позиции. Наконец, комбинированное изображение сохраняется (выделено). Тип сохраняемого файла определяется его расширением (например, *png*), но его можно было бы переопределить, передав аргумент формата методу *save()*. Заметьте, что не было никакой причины определять формат для входных файлов, так как тип изображения функция *open()* определяет по их содержанию.

```
im = pim.open(right)
comp.paste(im, (w+2*edge, edge))
im = pim.open(front)
comp.paste(im, (edge, h+2*edge))
im = pim.open(free)
comp.paste(im, (w+2*edge, h+2*edge))
comp.save(output)
```

Наша следующая функция рендерит вид из конкретной камеры и сохраняет результат в файл. Камера для рендера передаётся как имя Объекта Блендера (то есть, это не имя основного объекта *Camera*). Первая строка извлекает объект *Camera* и текущую сцену и делает камеру текущей в сцене - той которая будет рендерить (выделено ниже). Функция *setCurrentCamera()* принимает Объект Блендера, а не объект Камеры, и именно по этой причине мы передаём имя объекта.

```
def render(camera):
    cam = Object.Get(camera)
    scn = Scene.GetCurrent()
    scn.setCurrentCamera(cam)
    context = scn.getRenderingContext()
```

Так как нам может понадобиться использовать эту функцию в **фоновом процессе**, мы используем метод *renderAnim()* контекста рендера, а не метод *render()*. Дело в том, что метод *render()* не может быть использован в фоновом процессе. Следовательно, мы устанавливаем в значение текущего кадра как начальный, так и конечный кадры, чтобы гарантировать, что функция *renderAnim()* отрендерит единственный кадр. Мы также устанавливаем *displayMode* на 0, чтобы предотвратить появление дополнительного окна рендера (выделено в следующем куске кода):

```
frame = context.currentFrame()
context.endFrame(frame)
context.startFrame(frame)
context.displayMode=0
context.renderAnim()
```

Метод *renderAnim()* рендерит кадры в файлы, так что наша следующая задача в том, чтобы извлечь имя файла того кадра, который мы только что визуализировали. Точный формат имени файла может задаваться пользователем в окне **Пользовательских настроек**, но явный вызов функции *getFrameFilename()* даёт нам

уверенность, что мы получим правильное имя:

```
filename= context.getFrameFilename()
```

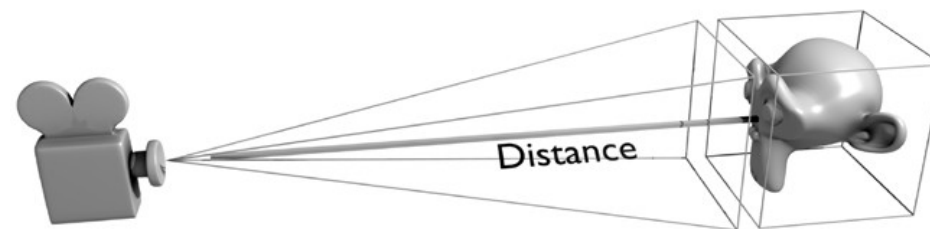
Так как номер кадра будет одинаковым для вида каждой камеры, что мы рендерим, мы должны переименовать этот файл, в противном случае он будет переписан. Следовательно, мы создаём новое подходящее имя, состоящее из пути к кадру, который мы только что рендерили, и имени камеры. Мы используем переносимые функции обработки пути из питонового модуля *os.path*, так, чтобы всё работало одинаково хорошо как под Windows, так и под Linux, например.

Так как наш скрипт, возможно, уже использовался, мы пытаемся удалить любой существующий файл с тем же именем, поскольку переименование файла на существующее имя потерпит неудачу под Windows. Конечно, файла пока могло и не быть - в этой ситуации нас защищает блок *try*. Наконец, наша функция возвращает имя вновь созданного файла:

```
camera = os.path.join(os.path.dirname(filename), camera)
try:
    os.remove(camera)
except:
    pass
os.rename(filename, camera)
return camera
```

Следующая важная задача - кадрировать вид камеры, то есть, так выбрать подходящий **угол** для всех камер, чтобы подогнать предмет под доступную область на изображении оптимальным образом. Мы хотим, чтобы угол камеры был одинаковым для всех камер, чтобы предоставить зрителю последовательную перспективу со всех углов просмотра. Конечно, это можно сделать вручную, но это скучно, так что мы определим функцию, которая будет работать за нас.

Способ, которым мы это сделаем - это взять **габаритный ящик** (bounding box) нашего предмета и определить угол зрения камеры, исходя из того, что этот габаритный ящик должен просто заполнить наш вид. Поскольку мы можем вычислить расстояние от камеры до центра габаритного ящика, угол зрения должен быть таким же, как и острый угол треугольника, формируемого габаритным ящиком и расстоянием до камеры.



Мы вычисляем этот угол для всех камер и затем настраиваем угол для каждой камеры в самый широкий из вычисленных, чтобы предотвратить нежелательное отсечение нашего предмета. Заметьте, что этот алгоритм может потерпеть неудачу, если камеры находятся слишком близко к предмету (или, что то же самое, если предмет слишком большой), в этом случае некоторое отсечение может произойти.

Код содержит много трудной математики, так что мы начнём, импортируя необходимые функции:

```
from math import asin,tan,pi,radians
```

Сама функция принимает список имен объектов Блендера (камер) и габаритный ящик (список векторов, по одному для каждого угла габаритного ящика). Она начинается с определения минимального и максимального размеров габаритного ящика для всех трех осей, затем вычисляются ширины. Мы допускаем, что наш предмет отцентрирован в начале координат. Переменная *maxx* содержит самую большую ширину из всех осей.

```
def frame(cameras,bb):
    maxx = max(v.x for v in bb)
    maxy = max(v.y for v in bb)
    maxz = max(v.z for v in bb)
    minx = min(v.x for v in bb)
    miny = min(v.y for v in bb)
    minz = min(v.z for v in bb)
    wx=maxx-minx
    wy=maxy-miny
    wz=maxz-minz
```

```
m=Mathutils.Vector((wx/2.0,wy/2.0,wz/2.0))
maxw=max((wx,wy,wz))/2.0
```

Затем, мы получаем глобальные координаты для каждого объекта *Camera*, чтобы вычислить расстояние *d* до средней точки габаритного ящика (выделено в следующем коде). Мы сохраняем частное от максимальной ширины и расстояния:

```
sins=[]
for cam in cameras:
    p=Mathutils.Vector(Object.Get(cam).getLocation(
                                                'worldspace'))

    d=(p-m).length
    sins.append(maxw/d)
```

Мы вычисляем наибольшее из этих частных (так как оно соответствует самому широкому углу), определяем угол через арксинус, и заканчиваем, устанавливая атрибут *lens* (линза) объекта Камеры. Соотношение между углом просмотра камеры и величиной атрибута *lens* в Блендере - сложное и плохо документированное (*lens* содержит аппроксимацию фокусного расстояния идеальной линзы). Показанная формула взята из исходного кода Блендера (выделено).

```
maxsin=max(sins)
angle=asin(maxsin)
for cam in cameras:
    Object.Get(cam).getData().lens = 16.0/tan(angle)
```

Другая удобная функция - та, которая создаёт четыре камеры и устанавливает их на сцену размещенными должным образом вокруг начала координат. Функция в принципе простая, но немного усложнена, поскольку она пытается заново использовать существующие камеры с тем же именем, чтобы предотвратить нежелательное размножение камер, если скрипт будет работать неоднократно. Словарь *cameras* индексируется по имени и содержит список позиций, поворотов, и величин *lens*:

```
def createcams():
    cameras = {
        'Top' : (( 0.0, 0.0,10.0),( 0.0,0.0, 0.0),35.0),
        'Right': ((10.0, 0.0, 0.0),(90.0,0.0,90.0),35.0),
        'Front': (( 0.0,-10.0, 0.0),(90.0,0.0, 0.0),35.0),
        'Free' : (( 5.8, -5.8, 5.8),(54.7,0.0,45.0),35.0)}
```

*Я это уже вроде упоминал, но скажу здесь ещё раз.*

*Категорически не рекомендуется вставлять числа (да и строки тоже) в текст программы. Расстояние 10.0 взято совершенно произвольно. А вдруг размер объекта окажется больше? Откуда взялось 5.8, человеку, незнакому с математикой, вообще будет непонятно, хотя в данном случае это просто длина ребра куба, длина диагонали которого равна 10.0 ( $\sqrt{(10.0^2)/3.0} \approx 5.8$ ). Правильным было бы объявить в начале программы константу, равную 10.0, а расстояние для камеры 'Free' вычислять из неё. Тогда для работы с объектом больших или меньших размеров потребовалось бы изменить значение всего одной константы. - дополнение переводчика.*

Для каждой камеры в словаре *cameras* мы проверяем, существует ли она уже как объект Блендера. Если это так, мы проверяем, имеет ли этот объект Блендера связанный с ним объект Камеры. Если последнее не является истиной, мы создаем перспективную камеру с тем же именем, как объект верхнего уровня (выделено), и ассоциируем его с объектом верхнего уровня посредством метода *link()*:

```
for cam in cameras:
    try:
        ob = Object.Get(cam)
        camob = ob.getData()
        if camob == None:
            camob = Camera.New('persp',cam)
            ob.link(camob)
```

Если там ещё не было объекта верхнего уровня, мы создаем его и связываем с ним новый объект перспективной Камеры:

```
except ValueError:
    ob = Object.New('Camera',cam)
    Scene.GetCurrent().link(ob)
    camob = Camera.New('persp',cam)
    ob.link(camob)
```

Мы выставляем позицию, поворот, и атрибут *lens*. Заметьте, что углы поворота выражаются в радианах, так что мы преобразуем их из более понятных градусов, которые мы использовали в нашей таблице (выделено). Мы заканчиваем, вызывая функцию *Redraw()* (обновление изображения), чтобы изменения появились в интерфейсе пользователя:

```
ob.setLocation(cameras[cam][0])
ob.setEuler([radians(a) for a in cameras[cam][1]])
camob.lens=cameras[cam][2]
Blender.Redraw()
```

Наконец, мы определяем метод *run()*, который связывает все компоненты вместе. Он определяет активный объект, затем проходит циклом по списку имен камер, чтобы отрендерить каждый вид и добавить результирующее имя файла в список (выделено):

```
def run():
    ob = Scene.GetCurrent().objects.active
    cameras = ('Top', 'Right', 'Front', 'Free')
    frame(cameras, ob.getBoundingBox())
    files = []
    for cam in cameras:
        files.append(render(cam))
```

Мы поместим скомбинированное изображение в тот же каталог, что и отдельные виды, и назовём его *result.png*:

```
outfile = os.path.join(os.path.dirname(
    files[0]), 'result.png')
```

Мы затем называем нашу функцию *paste()*, передавая список имён файлов компонентов, развёрнутый в индивидуальные аргументы оператором звездочка (\*), и, последний штрих, загружаем файл результата как изображение Блендера и показываем его в окне редактора изображений (выделено ниже). Функция *reload* (перегрузка) необходима чтобы удостовериться, что предыдущее изображение с тем же самым именем будет обновлено:

```
paste(*files, output=outfile)
im=Image.Load(outfile)
bpy.data.images.active = im
im.reload()
Window.RedrawAll()
```

Функция *run()* умышленно не создаёт никаких камер, поскольку пользователь может захотеть сделать это сам. Сам окончательный скрипт заботится о создании камер, но это можно изменить довольно легко, достаточно закомментировать строку. После проверки, если скрипт работает автономно, он просто создает камеры и вызывает метод *run*:

```
if __name__ == "__main__":
    createcams()
    run()
```

Полный код доступен как *combine.py* в файле *combine.blend*.

## Рабочий процесс - как продемонстрировать вашу модель

Скрипт можно использовать следующим образом:

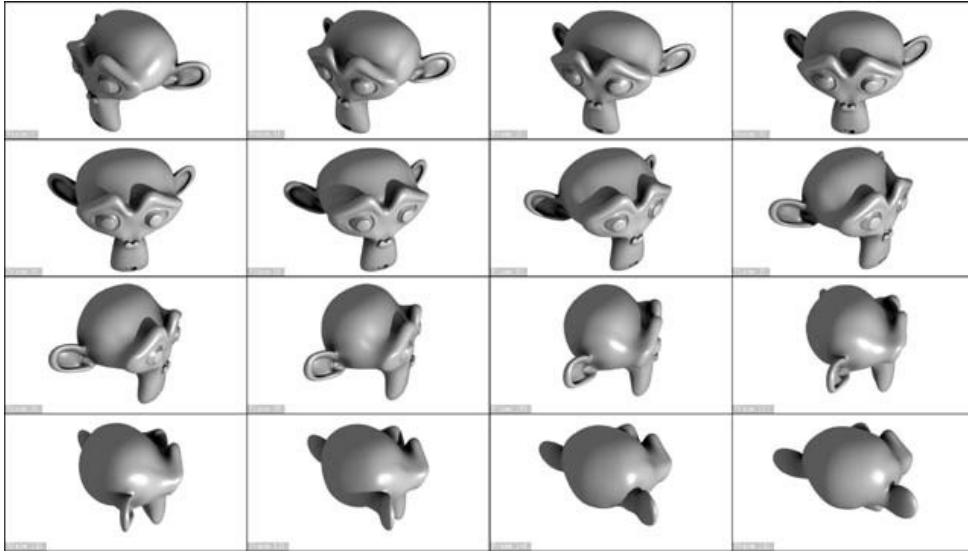
1. Поместите ваш предмет в начало координат (позиция (0, 0, 0)).
2. Создайте подходящие условия освещения.
3. Запустите *combine.py*.

Скрипт можно загрузить в текстовый редактор, чтобы запустить его с помощью Alt + P, но Вы также можете поместить его в каталог *scripts* Блендера, чтобы сделать его доступным из меню **Scripts | Render**.

## Now, strip — создание киноленты из анимации

Монтаж нескольких камер для разных точек зрения в одном изображении - просто один из примеров, где многочисленные изображения могут быть эффективно объединены в единственное. Другой пример - когда мы хотели бы показать кадры из анимации, в то время как у нас нет доступа к средствам для воспроизведения анимации. В таких ситуациях мы хотели бы показать что-то похожее на киноленту, где мы объединяем небольшие изображения, например, каждого десятого кадра, на единственном листе. Пример показан на следующей иллюстрации.

Хотя тут больше изображений для объединения, чем при нескольких видах камер, код для создания такой киноплёнки довольно похож.



Первой функцией, которую мы разрабатываем, будет *strip()* (лента), которая берет список имён файлов изображений для объединения, и необязательное имя, которое будет дано комбинированному изображению. Третий дополнительный аргумент - *cols*, количество колонок в комбинированном изображении. По умолчанию он равен четырём, но для длинных последовательностей может быть более естественным печатать их на горизонтальной бумаге, и использовать здесь большую величину. Функция возвращает объект *Image* Блендера, содержащий комбинированное изображение.

Мы снова используем модуль *pim*, который будет или псевдонимом для модуля PIL, если он доступен, или ссылкой на нашу собственную реализацию, если PIL не доступен. Важные отличия от нашего предыдущего кода комбинирования изображений выделены. Первая выделенная часть показывает, как вычислять размер комбинированного изображения, исходя из количества строк и колонок плюс количество пикселей, необходимое для цветных окантовок вокруг изображений. Вторая выделенная строка показывает, где мы вставляем картинки в целевое изображение:

```
def strip(files,name='Strip',cols=4):
    rows = int(len(files)/cols)
    if len(files)%int(cols) : rows += 1

    im = pim.open(files.pop(0))
    w,h= im.size
    edge=2
    edgecolor=(0.0,0.0,0.0)

    comp = pim.new(im.mode,
                    (w*cols+(cols+1)*edge,
                     h*rows+(rows+1)*edge),
                    edgecolor)

    for y in range(rows):
        for x in range(cols):
            comp.paste(im, (edge+x*(w+edge), edge+y*(h+edge)))
            if len(files)>0:
                im = pim.open(files.pop(0))
            else:
                comp.save(name,format='png')
                return Image.Load(name)
```

Функция *render()*, которую мы определяем, принимает количество пропускаемых кадров в виде аргумента и рендерит любое количество кадров между начальным и конечным кадрами. Эти начальный и конечный кадры могут быть заданы пользователем в панели кнопок рендера. Эти кнопки рендера также содержат величину шага, но эта величина не доступна в API Питона. Это означает, что наша функция будет несколько избыточнее, чем нам хотелось бы, так как мы должны создать цикл, который рендерит каждый отдельный кадр (выделено в следующем коде) вместо прямого вызова *renderAnim()*. Следовательно, мы должны манипулировать атрибутами *startFrame* и *endFrame* контекста рендера (как и раньше), но мы будем осторожными и восстановим эти атрибуты перед возвратом списка имён файлов отрендеренных картинок. Если бы мы не нуждались в каком-либо программном контроле значения величины пропуска, мы могли бы просто заменить вызов *render()* вызовом *renderAnim()*:



```
def render(skip=10):
    context = Scene.GetCurrent().getRenderingContext()
    filenames = []
    e = context.endFrame()
    s = context.startFrame()
    context.displayMode=0
    for frame in range(s,e+1,skip):
        context.currentFrame(frame)
        context.startFrame(frame)
        context.endFrame(frame)
        context.renderAnim()
        filenames.append(context.getFrameFilename())
    context.startFrame(s)
    context.endFrame(e)
    return filenames
```

После определения этих функций сам скрипт теперь просто вызывает *render()*, чтобы создавать изображения, и *strip()*, чтобы объединить их. Результирующее изображение Блендера перезагружается (*reload*) для его обновления на экране, если изображение с таким именем уже присутствовало, затем все окна перерисовываются (выделено):

```
def run():
    files = render()
    im=strip(files)
    bpy.data.images.active = im
    im.reload()
    Window.RedrawAll()

if __name__ == "__main__":
    run()
```

Полный код доступен как *strip.py* в файле *combine.blend*.

## Рабочий процесс — использование strip.py

Создать ленту анимационных кадров можно следующим образом:

1. Создать вашу анимацию. (Ага, это, конечно самый простой пункт ☺ -прим. пер.)
2. Запустить *strip.py* из текстового редактора.

3. Комбинированное изображение появится в окне редактора UV/image.

4. Сохранить изображение с именем по вашему выбору.

## Рендер билбордов

Слово **Billboard** дословно переводится как «доска для объявлений» или «рекламный щит», что конечно же мало подходит для нашего случая. В разработке игр *billboards* часто применяются, и к сожалению, адекватного перевода для этого нигде нет, везде используется эта уродливая транскрипция «**билборд**». Придётся и мне ей пользоваться - сожаление переводчика.

Реализм в сценах часто обеспечивается массой деталей, особенно на естественных объектах. Тем не менее, такой реализм даётся небесплатно, так как детализированные модели часто содержат много граней, и эти грани поглощают память и увеличивают время рендера. Реалистичная модель дерева может содержать больше полмиллиона граней, так что лес из них будет почти невозможно отрендерить, и, тем более, если этот лес является частью пейзажа в игре, идущей в быстром темпе.

Блендер поставляется со множеством инструментов, позволяющих уменьшить количество необходимой памяти при рендере множества копий объекта; различные Меш-объекты могут ссылаться на одни и те же данные меша, как, например, при **DupliVerts**. (Объекты-потомки, которые копируются в позицию каждой вершины родительского объекта. Смотри <http://www.is.svitonline.com/sailor/doc/man/specmod/dupliverts.htm> более подробно.) Дублирование объектов в системах частиц также позволяет нам создавать множество экземпляров того же самого объекта без действительного дублирования всех данных. Эти методы могут предотвратить потери огромного количества памяти, но детализированные объекты все еще могут требовать процессорных мощностей для рендера, поскольку их детали все еще должны быть отрендерены.

**Билборды** являются методом, используемым для наложения изображения сложного объекта на простой объект, такой, как

например, одиночная квадратная грань, и размножения этого простого объекта столько раз, сколько нужно. Изображение должно иметь подходящую прозрачность, в противном случае каждый объект будет закрывать другие не так, как требуется. За исключением этого момента, такая техника очень проста и может значительно уменьшить время рендера, и она даёт довольно реалистичные результаты для объектов, установленных на средних расстояниях или дальше. Системы частиц Блендера могут использовать билборды или как простые квадраты с наложенными изображениями, или накладывать изображение на простой объект и использовать его, как объект дублирования. Последнее также относится и к объектам duplivert.

Хитрость в том, что нужно сгенерировать изображение с подходящим освещением, чтобы использовать его как изображение, которое можно приложить к квадрату. На самом деле мы хотим создать два изображения: одно снятое с передней стороны, одно справа и построить объект, состоящий из двух квадратных граней, перпендикулярных друг другу с наложенными двумя изображениями. Такой объект даст нам несколько больше свободы в последствии при размещении камеры на нашей сцене, так как он не обязательно должен быть виден точно с одной стороны. Это хорошо работает только для объектов с приблизительно цилиндрической симметрией, как например, деревья или многоэтажки, но зато это очень эффективно.

Рабочий процесс для создания таких объектов достаточно сложен, так что его желательно автоматизировать:

1. Спозиционировать две камеры, спереди и справа от детального объекта.
2. Откадрировать обе камеры, чтобы они захватывали весь объект с одинаковым углом.
3. Отрендерить прозрачные изображения с premultiplied (заранее перемноженным) альфа-каналом и без неба.
4. Создать простой объект из двух перпендикулярных квадратов.
5. Наложить каждое отрендеренное изображение на квадрат.

6. Скрыть детальный объект от рендера.

7. Необязательно, скопировать простой объект в систему частиц (пользователю может не понадобится автоматизировать эту часть, если он захочет расставить простые объекты вручную).

"Premultiplication", упомянутое в третьем шаге, возможно, требует некоторого пояснения. Очевидно, отрендеренные изображения нашего сложного объекта не должны показывать никакого фонового неба, так как их скопированные клоны могут позиционироваться где угодно, и могут показывать различные части неба через свои прозрачные части. Как мы увидим, это достаточно просто сделать, но когда мы просто рендерим прозрачное изображение и перекрываем им позже некоторый фон, изображение может иметь некрасивые бросающиеся в глаза края.

Способ избежать этого - отрегулировать отрендеренные цвета, перемножив их с величиной альфы и контекст рендера имеет необходимые атрибуты, чтобы включить такой режим. Мы не должны забывать отмечать изображения, рендеренные как "premultiplied", при использовании их в качестве текстур, в противном случае они будут выглядеть слишком тёмными. Различие проиллюстрировано на следующем скриншоте, где мы скомпоновали и расширили правильно premultiplied левую половину и отрендеренную с небом правую половину. У ствола дерева справа проявляется светлый край. (Посмотрите отличную книгу Роджера Викаса "Foundation Blender Compositing", если нужна дополнительная информация.)



Буковое дерево (использованное на этой и последующих иллюстрациях) - это высокодетальная модель (свыше 30,000 граней), созданная Yorik van Havre с помощью свободного пакета моделирования растений **ngPlant**. (Смотри его вебсайт для большего количества отличных примеров: <http://yorik.uncreated.net/greenhouse.html>). Далее первый набор изображений показывает буковое дерево спереди и результирующий рендер передней грани билборда слева. (немного темнее из-за premultiplication).



Следующий набор скриншотов показывает то же буковое дерево, отрендеренное справа вместе с рендером правой грани билборда слева. Как может быть заметно, исполнение конечно, не идеально с этой точки зрения, но это крупный план, а разумный трехмерный аспект сохраняется.



Чтобы показать, как устроена конструкция билбордов, следующий скриншот показывает две грани с наложенными отрендеренными изображениями. Прозрачность умышленно уменьшена, чтобы было видно отдельные грани.



Нашей первой проблемой будут некоторые ранее используемые функции, которые мы писали для презентации модели с несколькими видами. Эти функции находятся в текстовом буфере с именем *combine.py*, и мы не сохраняли его во внешний файл. Мы создадим наш скрипт *cardboard.py* как новый текстовый буфер в том же *.blend* файле, где и *combine.py*, и хотим ссылаться на последний так же, как на внешний модуль. Блендер позволяет это делать, так как он ищет модуль в текущих текстовых буферах, если он не может найти внешний файл.

Поскольку внутренние текстовые буферы не имеют информации о том, когда они последний раз изменялись, мы должны убедиться, что загружена самая последняя версия. Об этом позаботится функция *reload()*. Если мы её не выполним, Блендер не сможет обнаружить возможных изменений в *combine.py*, что могло бы привести нас к использованию его более старой скомпилированной версии:

```
import combine
reload(combine)
```

Мы не будем использовать заново функцию *render()* из *combine.py*, поскольку сейчас у нас другие требования для рендеренных изображений, которые мы наложим на билборды. Как уже объяснялось, мы должны убедиться, что мы не получим никаких светлых краёв в местах с частичной прозрачностью, так что мы заранее включаем *premultiply* в альфа-канале (выделено). Мы восстанавливаем контекст рендера в 'рендер неба' (*rendering the sky*) обратно до возврата из этой функции, поскольку легко забыть установить его обратно вручную, и Вы можете потратить время на удивление, куда подевалось ваше небо:

```
def render(camera):
    cam = Object.Get(camera)
    scn = Scene.GetCurrent()
    scn.setCurrentCamera(cam)
    context = scn.getRenderingContext()
    frame = context.currentFrame()
    context.endFrame(frame)
    context.startFrame(frame)
    context.displayMode=0
    context.enablePremultiply()
    context.renderAnim()
    filename= context.getFrameFilename()
    camera = os.path.join(os.path.dirname(filename),camera)
    try:
        os.remove(camera) # удаление, в противном случае
                          # переименование
                          # потерпит неудачу в windows
    except:
        pass
    os.rename(filename,camera)

    context.enableSky()
    return camera
```

Каждое отрендеренное изображение должно быть преобразовано в подходящий материал, чтобы наложить его на квадрат с UV-отображением. Функция *imagemat()* будет делать это просто; она принимает объект Блендера *Image* в качестве аргумента и возвращает объект Материала. Этот материал будет сделан полностью прозрачным (выделено), но эта прозрачность и цвет модифицируются текстурой, которую мы назначаем в первый

текстурный канал (вторая выделенная строка). Тип текстур установлен в *Image* и, поскольку мы визуализировали эти изображения с premultiplied альфа-каналом, мы используем метод *setImageFlags()*, чтобы указать, что мы хотим использовать этот альфа-канал, и устанавливаем атрибут *premul* изображения в Истину:

```
def imagemat(image):
    mat = Material.New()
    mat.setAlpha(0.0)
    mat.setMode(mat.getMode() | Material.Modes.ZTRANSP)
    tex = Texture.New()
    tex.setType('Image')
    tex.image = image
    tex.setImageFlags('UseAlpha')
    image.premul=True
    mat.setTexture(0, tex, Texture.TexCo.UV,
                   Texture.MapTo.COL | Texture.MapTo.ALPHA)

    return mat
```

Каждая грань, к которой мы применяем материал, должна иметь UV-раскладку. В нашем случае, это будет самой простой из возможных раскладок, так как квадратная грань будет отображена так, чтобы в точности соответствовать прямоугольному изображению. Это часто называют **сбросом отображения**, и следовательно, функция, которую мы определим, называется *reset()*. Она принимает объект Блендера *MFace*, который мы считаем четырёхугольником, и присваивает его атрибуту *uv* список 2D-векторов, по одному для каждой вершины. Эти векторы размещают каждую из вершин в углах изображения:

```
def reset(face):
    face.uv=[vec(0.0,0.0),vec(1.0,0.0),
             vec(1.0,1.0),vec(0.0,1.0)]
```

Функция *cardboard()* заботится о создании фактического Меш-объекта из двух объектов *Image*, переданных как аргументы. Она начинается с создания двух квадратных граней, которые пересекают друг друга вдоль оси z. Следующий шаг должен добавить UV-слой (выделено) и сделать его активным:

```
def cardboard(left,right):
    mesh = Mesh.New('Cardboard')
    verts=[(0.0,0.0,0.0),(1.0,0.0,0.0),
            (1.0,0.0,1.0),(0.0,0.0,1.0),
```

```
(0.5,-0.5,0.0),(0.5,0.5,0.0),
(0.5,0.5,1.0),(0.5,-0.5,1.0)]
    faces=[(0,1,2,3),(4,5,6,7)]
    mesh.verts.extend(verts)
    mesh.faces.extend(faces)
```

```
mesh.addUVLayer('Reset')
    mesh.activeUVLayer='Reset'
```

Затем мы создаем подходящие материалы из обоих изображений, и назначаем эти материалы в атрибут меша *materials*. Далее, мы сбрасываем (reset) UV-координаты обеих граней, и назначаем им материалы (выделено). Мы обновляем (update) меш, чтобы сделать изменения видимыми до возврата из функции:

```
    mesh.materials=[imagemat(left),imagemat(right)]
```

```
    reset(mesh.faces[0])
    reset(mesh.faces[1])
mesh.faces[0].mat=0
mesh.faces[1].mat=1
```

```
    mesh.update()
    return mesh
```

Чтобы заменить меш дублированием объекта системой частиц, мы строим утилиту *setmesh()*. Она принимает имя объекта со связанной системой частиц и Меш-объект как аргументы. Она находит Объект по имени, и извлекает первую систему частиц (выделено в следующем куске кода). Объект дублирования находится в атрибуте *duplicateObject*. Заметьте, что этот атрибут *только для чтения*, так что к настоящему времени нет возможности поменять объект из Питона. Но мы можем заменить *данные* объекта и, мы это делаем посредством передачи Меш-объекта в метод *link()*. Оба объекта, эмиттер и объект дублирования системой частиц изменятся, так что мы удостоверимся, что изменения станут видимыми, вызывая метод *makeDisplayList()* для них обоих перед запуском обновления изображения (redraw) всех окон Блендера:

```
def setmesh(obname,mesh):
    ob = Object.Get(obname)
ps = ob.getParticleSystems()[0]
    dup = ps.duplicateObject
    dup.link(mesh)
```

```
ob.makeDisplayList()
dup.makeDisplayList()
Window.RedrawAll()
```

Функция *run()* включает всю работу, которую нужно сделать, чтобы преобразовать активный объект в набор билбордов, и назначить его в систему частиц. Сначала мы извлекаем ссылку на активный объект, и убеждаемся, что он будет видимым при рендере:

```
def run():
    act_ob = Scene.GetCurrent().objects.active
    act_ob.restrictRender = False
```

Следующим шагом нужно сделать остальные объекты на сцене невидимыми до того, как мы отрендерим билборды. Некоторые из них, возможно, уже были сделаны невидимыми пользователем, следовательно, мы должны запомнить эти состояния, чтобы мы могли восстановить их позже. Также мы не изменяем состояние ламп или камер, так как сделав их невидимыми, мы останемся с полностью черными изображениями (выделено):

```
renderstate = {}
for ob in Scene.GetCurrent().objects:
    renderstate[ob.getName()] = ob.restrictRender
    if not ob.getType() in ('Camera', 'Lamp') :
        ob.restrictRender = True
act_ob.restrictRender = False
```

Как только всё настроено, чтобы рендерить только активный объект, мы рендерим переднее и правое изображения с должным образом откадрированными камерами, просто подобно тому, как мы это делали в скрипте *combine.py*. Фактически, здесь мы заново используем функцию *frame()* (выделено):

```
cameras = ('Front', 'Right')
combine.frame(cameras, act_ob.getBoundingBox())
images={}
for cam in cameras:
    im=Image.Load(render(cam))
    im.reload()
    images[cam]=im
bpy.data.images.active = im
Window.RedrawAll()
```

Затем мы восстанавливаем предыдущую видимость всех объектов на сцене прежде, чем мы создадим новый меш из двух

изображений. Мы заканчиваем, делая активный объект невидимым для рендера и заменяя меш объекта дублирования в определенной системе частиц нашим новым мешем:

```
for ob in Scene.GetCurrent().objects:
    ob.restrictRender = renderstate[ob.getName()]
```

```
mesh = cardboard(images['Front'], images['Right'])
act_ob.restrictRender = True
setmesh('CardboardP', mesh)
```

Последние строки кода создают камеры, необходимые для рендера билбордов (если эти камеры в данный момент отсутствуют), вызывая функцию *createcams()* из модуля *combine* до вызова *run()*:

```
if __name__ == "__main__":
    combine.createcams()
    run()
```

Полный код доступен как *cardboard.py* в файле *combine.blend*.

## Рабочий процесс - использование cardboard.py

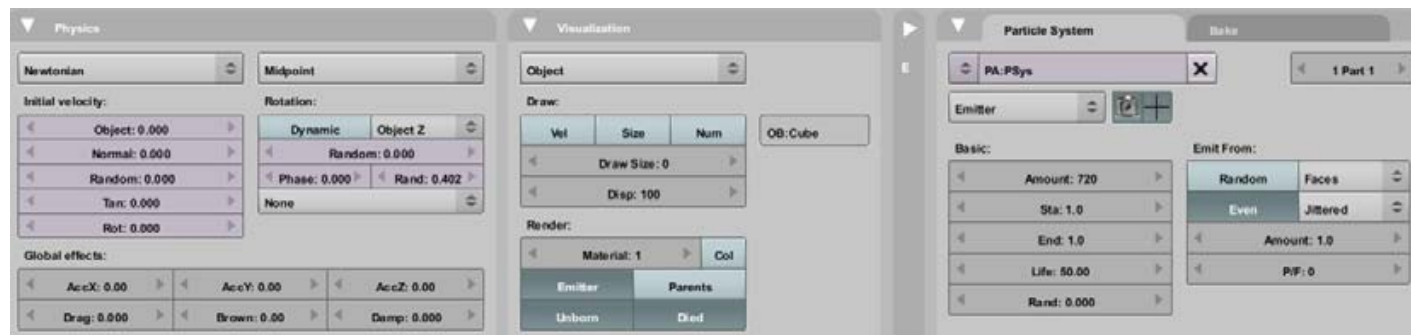
Допустим, что у вас есть высокополигональный объект, и что Вы хотели бы преобразовать его в набор билбордов, тогда работа могла бы выглядеть примерно так:

1. Создать объект с именем *CardboardP*.
2. Назначить систему частиц на этот объект.
3. Создать временный куб.
4. Назначить временный куб дублированным объектом в первой системе частиц объекта *CardboardP*.
5. Выбрать (сделать активным) объект, который будет отрендерен как набор билбордов.
6. Запустить *cardboard.py*.
7. Выбрать первоначальную камеру и отрендерить сцену.

Конечно, скрипт можно изменить, чтобы пропустить автоматизированную замену меша объектов дублирования, если это нужно. Например, если мы хотели бы использовать объекты *dupliverts* вместо частиц, мы должны просто сгенерировать *cardboard*



объект и назначить свой меш на объект дублирования. Если мы используем систему частиц, мы, вероятно, не хотим, чтобы все размноженные объекты были ориентированы точно в одном и том же направлении. Мы могли бы, следовательно, сделать их вращение случайным, пример настройки для этого показан на следующем скриншоте:



Следующий скриншот иллюстрирует применение билбордов, созданных из модели дерева, и использованных в системе частиц:



## Генерация вопросов CAPTCHA

Во многих ситуациях, как например, блогах, форумах, и онлайн-опросах (можно назвать ещё несколько), операторы вебсайтов хотят избежать автоматизированных почтовых отправок от спамботов, но не хотят напрягать посетителей-людей регистрацией с аутентификацией. В таких ситуациях, которые стали обычными,

посетителю предлагают так называемый вопрос CAPTCHA (<http://ru.wikipedia.org/wiki/CAPTCHA>). **Вопрос CAPTCHA** (или просто **Captcha**) в самой своей простой форме - изображение, которое должно быть трудным для компьютерного распознавания, но простым для расшифровки человеком, обычно это искаженное или смазанное слово или число.

Конечно, никакой из методов не является абсолютно надёжным, и несомненно, вопросы Captcha не лишены недостатков, они не будут устойчивыми при доступных больших компьютерных мощностях, но они все еще остаются весьма эффективными. Хотя в настоящее время считается, что способы с простым размытием и окраской — задачи решенные,

компьютерам все еще требуется серьезное время на разделение индивидуальных символов в слове, когда они слегка перекрывают друг друга, в то время как для людей это не проблема.

Учитывая эти аргументы, это может быть отличным применением рендеринга текста в 3D, так как, предположительно, трехмерное представление слов при подходящих условиях освещения (то есть, резкие тени) даже труднее для интерпретации, чем двумерный текст. Наша задача тогда заключается в разработке сервера, который будет отвечать на запросы, чтобы сделать трехмерное изображение какого-нибудь текста.

Мы разработаем наш сервер как веб-сервер, который будет реагировать на запросы, адресованные ему как URL'ы в форме `http:<hostname>:<port>/captcha?text=<sometext>`, и который возвращает PNG-изображение - 3D-представление этого текста. Таким образом, будет легко внедрить этот сервер в архитектуру, в которой некоторое программное обеспечение, например блог, может легко использовать эту функциональность, просто подключаясь к нашему серверу через *HTTP*. Пример сгенерированного вопроса показан на иллюстрации:





## Разработка сервера CAPTCHA

При использовании модулей, доступных в полном дистрибутиве Питона, задача создания сервера *HTTP* становится не такой уж пугающей, как может показаться. Наш сервер *Captcha* будет основан на классах, предоставленных модулем Питона *BaseHTTPServer*, так что мы начинаем с импорта этого модуля вместе с несколькими дополнительными модулями-утилитами:

```
import BaseHTTPServer
import re
import os
import shutil
```

Модуль *BaseHTTPServer* определяет два класса, которые вместе включают полную реализацию сервера *HTTP*. Класс *BaseHTTPServer* реализует основной сервер, который будет слушать поступающие *HTTP*-запросы на некотором сетевом порту, и мы используем этот класс, как есть.

При получении корректного *HTTP*-запроса *BaseHTTPServer* пошлет этот запрос обработчику запросов. Наша реализация такого обработчика запросов, основанная на *BaseHTTPRequestHandler*, довольно скучна, так как ожидается, что всё, что он будет делать - запрашивать поля *GET* и *HEAD* в форме *captcha?text=abcd*. Следовательно, всё мы должны сделать - переписать методы *do\_GET()* и *do\_HEAD()* базового класса.

От запроса *HEAD* ожидается возвращение только заголовков запрошенного объекта, а не содержимого, чтобы сохранять время, за которое содержимое не изменится со времени последнего запроса

(что-то, что может быть определено проверкой заголовка *Last-Modified*). Мы игнорируем такую аккуратность; мы возвращаем заголовки именно тогда, когда мы получаем запрос *HEAD*, но мы, тем не менее, будем генерировать полностью новое изображение. Это в некоторой степени расточительно, но зато код будет простым. Если важна производительность, можно разработать другую реализацию.

Наша реализация начинается с определения метода *do\_GET()*, который просто вызывает метод *do\_HEAD()*, который будет генерировать вопрос *Captcha* и возвращать заголовки клиенту. *do\_GET()*, впоследствии, копирует содержание файлового объекта, возвращённого методом *do\_HEAD()* в выходной файл, такой как объект обработчика запроса (выделено), который в свою очередь возвращает это содержимое клиенту (например, браузеру):

```
class CaptchaRequestHandler(
    BaseHTTPServer.BaseHTTPRequestHandler):
    def do_GET(self):
        f=self.do_HEAD()
        shutil.copyfileobj(f,self.wfile)
        f.close()
```

Метод *do\_HEAD()* сначала определяет, получили ли мы правильный запрос (то есть, *URI* в форме *captcha?text=abcd*), вызывая метод *gettext()* (выделено, определяется позже в коде). Если *URI* некорректен, метод *gettext()*, возвращает *None* и тогда *do\_HEAD()* возвращает клиенту ошибку **File not found** (Файл не найден), вызывая метод *send\_error()* базового класса:

```
def do_HEAD(self):
    text=self.gettext()
    if text==None:
        self.send_error(404, "File not found")
        return None
```

Если был запрошен корректный *URI*, фактическое изображение генерируется методом *captcha()*, который возвращает имя файла сгенерированного изображения. Если этот метод терпит неудачу по любой причине, клиенту возвращается **Internal server error** (Внутренняя ошибка сервера):

```
try:
    filename = self.captcha(text)
except:
    self.send_error(500, "Internal server error")
```

```
return None
```

Если все прошло хорошо, мы открываем файл изображения, отсылаем клиенту ответ **200** (показывающий успешную операцию), и возвращаем заголовок *Content-type*, устанавливающий, что мы возвращаем *png*-изображение. Затем мы используем функцию *fstat()* с номером *handle* открытого файла в качестве аргумента, чтобы извлечь длину сгенерированного изображения и вернуть её как заголовок *Content-Length* (выделено), сопроводив временем модификации и пустой строкой, означающей конец заголовков перед возвратом открытого файлового объекта *f*:

```
f = open(filename, 'rb')
self.send_response(200)
self.send_header("Content-type", 'image/png')
fs = os.fstat(f.fileno())
self.send_header("Content-Length", str(fs[6]))
self.send_header("Last-Modified",
                  self.date_time_string(fs.st_mtime))
self.end_headers()
return f
```

Метод *gettext()* проверяет, что запрос, передаваемый нашему обработчику запросов в переменной пути - правильный URI, сверяя его с регулярным выражением. Функция *match()* из модуля Питона *re* возвращает *MatchObject* (объект сопоставления), если регулярное выражение соответствует параметру, и *None*, если нет. Если есть соответствие, мы возвращаем содержание первой группы объекта сопоставления (символы, которые соответствуют выражению между круглыми скобками в регулярном выражении, в нашем случае значение текстового аргумента), в противном случае мы возвращаем *None*:

```
def gettext(self):
    match = re.match(r'^.*?/captcha\?text=(.*)$',
                     self.path)
    if match != None:
        return match.group(1)

    return None
```

Теперь мы добрались до задачи, специфичной для Блендера - сгенерировать рендеренный в 3D текст, который будет возвращён в виде *png* изображения. Метод *captcha()* принимает текст для рендера

как аргумент, и возвращает имя файла сгенерированного изображения. Мы допускаем, что освещение и камера в *.blend* файле, в котором мы запускаем *captcha.py*, настроены правильно, чтобы удобно отображать наш текст. Следовательно, метод *captcha()* просто настраивает правильным образом объект *Text3d* и рендерит его.

Первая задача состоит в том, чтобы определить текущую сцену и проверить, присутствует ли объект с именем *Text*, который можно использовать заново (выделено). Заметьте, что вполне допустимо иметь другие объекты на сцене, чтобы ещё более затемнить отображение:

```
def captcha(self, text):
    import Blender
    scn = Blender.Scene.GetCurrent()
    text_ob = None
    for ob in scn.objects:
        if ob.name == 'Text' :
            text_ob = ob.getData()
            break
```

Если не нашлось никакого ранее используемого объекта *Text3d*, создаём новый :

```
if text_ob == None:
    text_ob = Blender.Text3d.New('Text')
    ob=scn.objects.new(text_ob)
    ob.setName('Text')
```

Следующий шаг - установить текст объекта *Text3d* в значение аргумента, переданного в метод *captcha()*, и сделать его трёхмерным, настроив глубину выдавливания. Мы также изменяем ширину символов и сокращаем расстояние между ними, чтобы ухудшить разделение. Добавление небольшого скоса (*bevel*) смягчит контуры символов, что может добавить трудностей для робота, различающего символы, если настроено искусное освещение (выделено). Мы могли бы решить использовать другой шрифт для нашего текста, который ещё труднее для чтения ботом, и здесь как раз место для установки такого шрифта (смотри следующий информационный блок).

### Чего-то не хватает



Документация API Блендера имеет небольшой пропуск: как будто не существует способа настроить другой шрифт для объекта `Text3d`. Тем не менее, есть недокументированный метод `setFont()`, который принимает объект *Font* в качестве аргумента. Код, выполняющий изменение шрифта должен выглядеть похожим на это:

```
fancyfont=Text3d.Load('/usr/share/fonts/ttf/myfont.ttf')
text_ob.setFont(fancyfont)
```

Тем не менее, мы решили не включать этот код, частично потому что он недокументирован, но по большей части потому, что доступные шрифты существенно отличаются от системы к системе. Если у Вас есть подходящий доступный шрифт, во что бы то ни стало используйте его. Скрипт, пишущий шрифтами, которые, например, напоминают почерк, могут поднять планку сложности для компьютера ещё выше.

Последним шагом нужно обновить дисплейный список Блендера для этого объекта, чтобы наши изменения были отрендерены:

```
text_ob.setText(text)
text_ob.setExtrudeDepth(0.3)
text_ob.setWidth(1.003)
text_ob.setSpacing(0.8)
text_ob.setExtrudeBevelDepth(0.01)
ob.makeDisplayList()
```

Как только наш объект `Text3d` будет на месте, нашей следующей задачей станет отрендерить его изображение в файл. Сначала мы извлекаем контекст рендера из текущей сцены и устанавливаем *displayMode* в 0, чтобы предотвратить появление дополнительного окна рендера:

```
context = scn.getRenderingContext()
context.displayMode=0
```

Затем, мы устанавливаем размер изображения и указываем, что нам нужен формат *png*. Включением *RGBA* и установкой альфа-режима в 2 мы гарантируем, что там не будет видно никакого неба, и что наше изображение будет иметь хороший прозрачный фон:

```
context.imageSizeX(160)
context.imageSizeY(120)
context.setImageType(Blender.Scene.Render.PNG)
context.enableRGBAColor()
context.alphaMode=2
```

Даже если мы рендерим простое неподвижное изображение, мы используем метод *renderAnim()* контекста рендера, поскольку иначе результаты рендерятся не в файл, а только в буфер. Следовательно, мы устанавливаем начальный и конечный кадры анимации в 1 (точно так же, как и текущий кадр), чтобы удостовериться, что мы генерируем простой одиночный кадр. Затем мы используем метод *getFrameFilename()*, чтобы получить имя файла (с полным путём) отрендеренного кадра (выделено). Далее мы одновременно сохраняем это имя файла и возвращаем его как результат:

```
context.currentFrame(1)
context.sFrame=1
context.eFrame=1
context.renderAnim()
self.result=context.getFrameFilename()
return self.result
```

Последняя часть скрипта определяет функцию *run()*, чтобы запустить сервер *Captcha*, и вызывает эту функцию, если скрипт выполняется автономно (то есть, если он не был импортирован как модуль). Определив функцию *run()* таким образом, мы можем изолировать часто используемые параметры сервера по умолчанию, как например, номер порта, который прослушивается (выделено), но допустимо повторное использование модуля, если потребовалась другая настройка:

```
def run(HandlerClass = CaptchaRequestHandler,
        ServerClass = BaseHTTPServer.HTTPServer,
        protocol="HTTP/1.1"):
```

```
    port = 8080
```

```
    server_address = ('', port)
```

```
    HandlerClass.protocol_version = protocol
```

```
    httpd = ServerClass(server_address, HandlerClass)
```

```
    httpd.serve_forever()
```

```
if __name__ == '__main__':  
    run()
```

Полный код доступен как *captcha.py* в файле *captcha.blend*, и сервер можно запустить несколькими путями: из текстового редактора (с *Alt + P*), из меню **Scripts | render | captcha**, или запустив Блендер в фоновом режиме из командной строки. Чтобы остановить сервер снова, необходимо завершить Блендер. Обычно это можно сделать посредством нажатия **Ctrl + C** в консоли или в окне DOSbox.

#### Предупреждение



Заметьте, что этот сервер реагирует на чьи угодно запросы, а это далеко небезопасно. Как минимум он должен быть запущен через межсетевой экран, который ограничивает доступ к нему только для сервера, которому требуются вопросы Captcha. Прежде чем запускать его в любом месте, которое может быть доступно из Интернета, вы должны тщательно подумать о безопасности вашей сети!

## Итог

В этой главе мы автоматизировали процесс рендера и узнали, как выполнять множество операций с изображениями без потребности во внешнем графическом редакторе. Мы изучили:

- Автоматизацию процесса рендера
- Создание множества видов для презентации продукта
- Создание билбордов из сложных объектов
- Манипуляцию изображениями, в том числе результатами рендера, используя библиотеку обработки изображений Python Imaging Library (PIL)
- Построение сервера, создающего изображения по требованию, которые могут быть использованы как вопросы CAPTCHA

В последней главе мы взглянем на некоторые служебные задачи.

## Расширение вашего инструментария

В этой главе мы будем меньше говорить о процессе рендера, и больше о том, как сделать жизнь легче для повседневного использования Блендера, расширяя его функциональность. Мы будем использовать некоторые внешние библиотеки, которые нужно будет установить, и в определенный момент скрипты Питона, возможно, станут немного труднее для чтения начинающими. Также, с точки зрения художника, это глава может быть не настолько визуально приятной, так как работу этих скриптов не представишь в виде симпатичных иллюстраций. Тем не менее, эти скрипты на самом деле добавляют полезную функциональность, особенно для разработчика скриптов, так что, пожалуйста, продолжайте читать.



В этой главе Вы узнаете как:

- Построить список активов, например, карты изображений, и заархивировать их
- Публиковать отрендеренное изображение автоматически через FTP
- Расширить функциональность встроенного редактора поиском с регулярными выражениями
- Ускорить вычисления, используя Psyc - компилятор-на-лету
- Добавить управление версиями к вашим скриптам с помощью Subversion

### ***В Сеть и дальше - публикация готового рендера на FTP***

Мы можем сохранить отрендеренное изображение в любое место, видимое в файловой системе, но не все платформы дают возможность сделать удалённый FTP-сервер доступным через локальный каталог. Этот скрипт предлагает нам простую опцию, позволяющую загружать отрендеренное изображение на удалённый FTP-сервер, и запоминает имя сервера, имя пользователя, и (необязательно) пароль, чтобы позже использовать их снова.

**File Transfer Protocol (FTP)** (Протокол передачи файлов), который мы будем использовать, несколько сложнее, чем, например, протокол HTTP, так как он использует больше одной связи. К счастью для нас, все сложности FTP-клиента хорошо изолированы в стандартном модуле Питона *ftplib*. Мы не только импортируем класс *FTP* этого модуля, но также множество других стандартных модулей Питона, особенно для обработки путей файлов (*os.path*) и для чтения файлов стандарта *.netrc* (который позволит нам сохранять пароли за

пределами нашего скрипта, если нам нужны пароли для регистрации на FTP-сервере). Мы обсудим каждый модуль, когда понадобится.

```
from ftplib import FTP
import os.path
import re
import netrc
import tempfile
from Blender import Image, Registry, Draw
```

Питон изначально является почти платформонезависимым, но, конечно, иногда встречаются сложности, которые не полностью охвачены. Например, мы хотим использовать имена пользователя и пароли, сохраненные в файле *.netrc*, который обычно используется программами FTP (и другими), и FTP-клиент ожидает, что этот файл будет находиться в домашнем каталоге пользователя, который он надеется найти в переменной окружения *HOME*. На Windows, тем не менее, понятие домашнего каталога не так хорошо определено, и существуют различные схемы для сохранения данных, которые ограничиваются единственным пользователем; не каждая реализация Питона справляется с этим одинаковым образом.

Следовательно, мы определяем небольшую функцию-утилиту, которая проверяет наличие переменной *HOME* в окружении (она всегда есть на Unix-подобных операционных системах, и на некоторых версиях Windows). Если таковой нет, она проверяет наличие переменной *USERPROFILE* (присутствует в большинстве версий Windows, включая XP, где она обычно указывает на каталог *C:\Documents и Settings\<имя пользователя>*). Если она присутствует, функция устанавливает переменную *HOME* в значение, содержащееся в этой переменной *USERPROFILE*:

```
def sethome():
    from os import environ
    if not 'HOME' in environ:
        if 'USERPROFILE' in environ:
            environ['HOME'] = environ['USERPROFILE']
```

Наша следующая задача в том, чтобы выяснить, на какой FTP-сервер пользователь хочет загрузить результат рендера. Мы запоминаем это в ключе реестра Блендера, чтобы не надоедать пользователю с приглашением всякий раз, когда он хочет отправить свой рендер. Функция *getftphost()* принимает аргумент *reuse*

(повторное использование), который может быть использован для очистки этого ключа, если он установлен в *False* (для обеспечения возможности выбора другого FTP-сервера), но переписать интерфейс пользователя, чтобы предлагать ему такую возможность, мы оставляем в качестве упражнения читателю.

Фактический код начинает с поиска ключа реестра (с диска, если необходимо, следовательно, по умолчанию аргумент *True*, выделено). Если там нет ключа, или он не содержит имя сервера, мы запрашиваем у пользователя имя FTP-сервера посредством всплывающего окна. Если пользователь его не вводит, мы заканчиваем, возбуждая исключение. В противном случае, мы сохраняем имя хоста - сначала создаём словарь, если он ещё не существует, и сохраняем этот словарь в реестр Блендера. Наконец, мы возвращаем сохранённое имя хоста.

```
def getftphost(reuse=True):
    dictname = 'ftp'
    if reuse == False:
        Registry.RemoveKey(dictname)

    d = Registry.GetKey(dictname, True)
    if d == None or not 'host' in d:
        host = Draw.PupStrInput("Ftp hostname:", "", 45)
        if host == None or len(host) == 0 :
            raise Exception("no hostname specified")
        if d == None :
            d={}
        d['host'] = host
        Registry.SetKey(dictname, d, True)
    return d['host']
```

Нам нужна другая вспомогательная функция, чтобы убедиться, что на диск в качестве изображения Блендера сохранено последнее отрендеренное изображение, которое присутствует как изображение с именем *Render Result*, но это изображение не пишется на диск автоматически. Функция *imagefilename()* принимает изображение Блендера как аргумент, и, во-первых, проверяет, существует ли корректное имя файла, связанное с ним (выделено). Если нет, она создает имя файла из имени изображения, добавляя расширение *.tga* (изображения можно сохранять только как файлы TARGA). Затем создаётся полный путь из этого имени файла и пути к временному



каталогу. Теперь, когда у нас есть корректное имя файла, мы его сохраняем, вызывая метод `save()`, и возвращая имя файла:

```
def imagefilename(im):
    filename = im.getFilename()
    if filename == None or len(filename) == 0:
        filename = im.getName() + '.tga'
        filename = os.path.join(tempfile.gettempdir(),
                                filename)
    im.setFilename(filename)
    im.save()
    return filename
```

Когда мы загружаем файл на FTP-сервер, мы хотим убедиться, что мы не перезапишем существующий файл. Если мы обнаружим, что файл с данным именем уже существует, мы хотели бы иметь функцию, которая создаёт новое имя файла предсказуемым способом, похожим на то, как ведёт себя Блендер при создании имён для объектов Блендера. Мы хотели бы сохранять расширение файла, так что мы не можем просто прилепить к имени цифровой суффикс. Функция `nextfile()`, следовательно, сначала разделяет имя пути и часть с расширением. Она использует функции `split()` и `splitext()` из модуля `os.path`, чтобы оставить нам чистое имя.

Если имя уже заканчивается на суффикс, состоящий из точки и некоторого числа (например, `.42`), мы хотели бы увеличить это число. Это именно то, что выполняет довольно пугающая выделенная строка. Функция `sub()` модуля Питона `re` принимает регулярное выражение как первый аргумент (мы используем здесь сырую строку, так что нам не надо экранировать обратную косую черту), и проверяет, соответствует ли это регулярное выражение своему третьему аргументу (`name`, в данном случае). Регулярное выражение, используемое здесь, `(\.(ld+)$)` совпадает с точкой, за которой следуют одна или более десятичных цифр, но только, если эти цифры являются последними символами. Если есть соответствие образцу, он заменяется вторым аргументом функции `sub()`. В нашем случае, замена - это не простая строка, а лямбда-функция (то есть, безымянная), в которую мы передаём объект сопоставления, и ожидаем, что она вернёт строку.

Мы окружили часть цифр нашего регулярного выражения круглыми скобками, теперь мы можем просто извлечь эти цифры (без

первоначальной точки), вызвав метод `group()` объекта сопоставления. Мы передаем ему 1 в качестве аргумента, так как первые открывающие скобки обозначают первую группу (группа 0 является всем образцом целиком). Мы преобразуем эту строку цифр в целое, используя встроенную функцию `int()`, добавляем к ней 1, и преобразуем её обратно в строку с функцией `str()`. До того, как этот результат автоматически будет возвращён из лямбда-функции, мы снова добавляем точку, чтобы соответствовать нашему желаемому образцу.

Мы завершаем проверкой, отличается ли результирующее имя от оригинального. Если они совпадают, значит оригинальное имя не соответствовало нашему образцу, и мы просто добавляем `.1` к имени. Наконец, мы восстанавливаем полное имя файла, добавляя расширение, и вызывая функцию `join()` из модуля `os.path`, чтобы добавить путь платформо-независимым способом:

```
def nextfile(filename):
    (path,base) = os.path.split(filename)
    (name,ext) = os.path.splitext(base)
    new = re.sub(r'\.(\d+)$',
                lambda m: '.' + str(1+int(m.group(1))),
                name)
    if new == name :
        new = name + '.1'
    return os.path.join(path,new+ext)
```

Теперь мы полностью готовы заняться реальной работой загрузки файла на FTP-сервер. Сначала мы удостоверимся, что наше окружение имеет переменную `HOME`, вызывая функцию `sethome()`. Затем, мы извлекаем имя хоста FTP-сервера, на который мы хотим загрузить (вполне законно, между прочим, ввести IP-адрес вместо имени хоста):

```
if __name__ == "__main__":
    sethome()
    host = getftphost()
```

Далее, мы извлекаем данные учётной записи пользователя для выбранного хоста из файла `.netrc`, если он присутствует (выделено). Это может закончиться неудачей по различным причинам (могло не быть `.netrc`-файла, или данные хоста отсутствуют в файле); в этом случае будет возбуждено исключение. Если это случится, мы

сообщаем об этом пользователю и требуем имя пользователя и пароль с помощью всплывающего окна:

```
try:
    (user,acct,password) = \
        netrc.netrc().authenticators(host)
except:
    acct=None
    user = Draw.PupStrInput(
        'No .netrc file found, enter username:',
        "",75)
    password = Draw.PupStrInput('Enter password:', "",75)
```

Отрендеренное изображение было сохранено как объект Блендера *Image* с именем *Render Result*. Следующая вещь, которую мы делаем - извлекаем ссылку на это изображение и убеждаемся, что оно сохранено на диск. Функция *imagefilename()*, которую мы определили раньше, возвращает имя файла загруженного изображения.

Следующим шагом нужно подключиться к FTP-серверу, используя имя хоста и данные учётной записи, которые мы извлекли раньше (выделено). Как только связь будет установлена, мы извлекаем список имён файлов с помощью метода *nlst()*:

```
im = Image.Get('Render Result')
filename = imagefilename(im)

ftp = FTP(host,user,password,acct)
files = ftp.nlst()
```

*Хм, автор так аккуратно обрабатывает ситуации отсутствия файла .netrc, имени с паролем в нём, сохранённости рендеренного изображения, а о работоспособности FTP-сервера вообще не упоминает. По-моему, ситуацию отсутствия связи, а также неверности логина или пароля тоже необходимо обрабатывать через try/except. - прим. пер.*

Поскольку мы хотим убедиться, что мы не перезаписываем никаких файлов на FTP-сервере, мы удаляем путь из имени файла нашего загруженного изображения с помощью функции *basename()* и сравниваем результат со списком имён файлов, извлеченным с сервера (выделено). Если имя файла уже присутствует, мы генерируем новое имя функцией *nextfile()* и снова проверяем, и

продолжаем проверять, пока у нас, наконец, не появится имя файла, которое в данный момент отсутствует на FTP-сервере.

```
dstfilename = os.path.basename(filename)
while dstfilename in files:
    dstfilename = nextfile(dstfilename)
```

Затем, мы выгружаем наш файл изображения, вызывая метод *storbinary()*. Этот метод принимает имя целевого файла с префиксом *STOR*, как первый аргумент, и открытый файловый дескриптор как второй аргумент. Мы предоставляем последний, вызывая встроенную функцию Питона *open()* с именем нашего файла изображения в качестве единственного аргумента. (Если нужна дополнительная информация о довольно диковинном поведении модуля *ftplib*, ссылка на его документацию: <http://docs.python.org/library/ftplib.html>.) Мы грациозно заканчиваем связь в FTP-сервером, вызывая метод *quit()*, и сообщаем пользователю о завершении задачи, показывая сообщение с упоминанием имени целевого файла, так как оно может отличаться от ожидаемого, если существует файл с аналогичным именем:

```
ftp.storbinary('STOR '+dstfilename,open(filename))

ftp.quit()

Draw.PupMenu('Render result stored as "%s"%s|Ok'
             %(dstfilename,'%t'))
```

Полный код доступен как *ftp.py* в файле *ftp.blend*. Его можно запустить из текстового редактора, но в общем случае, несомненно, значительно удобнее поместить *ftp.py* в каталог скриптов Блендера. Скрипт сконфигурирован так, чтобы он был доступен в меню **Файл | Экспорт** (File | Export).

## Весенняя уборка - архивация неиспользуемых изображений

Через некоторое время у любого долгоживущего проекта набирается много хлама. Например, изображения текстур, которые Вы пытались применить, но они были отвергнуты в пользу более подходящих. Этот скрипт поможет нам найти все файлы в выбранном

каталоге, на которые нет ссылок в нашем *.blend* файле, и упаковать их в ZIP-архив.

Мы позаботимся о том, чтобы не переносить никаких *.blend* файлов в ZIP-архив (в конце концов, мы, как правило, хотим быть в состоянии рендерить), ни самого ZIP-архива (для предотвращения бесконечной рекурсии). Любой файл, который мы архивируем, мы затем попытаемся удалить, и если удаление файла оставляет пустой каталог, мы удалим также этот каталог, если он не является тем каталогом, где находится наш *.blend* файл.

Функции работы с файлами предоставляются модулями Питона *os* и *os.path*, а ZIP-файлами, которые могут использоваться как в Windows так и на открытых платформах, можно манипулировать с помощью модуля *zipfile*. ZIP-файл, в который мы перемещаем неиспользованные файлы, мы назовём *Attic.zip*:

```
import Blender
from os import walk, remove, rmdir, removedirs
import os.path
from zipfile import ZipFile
```

```
zipname = 'Attic.zip'
```

Первой задачей будет сгенерировать список всех файлов в каталоге, где находится наш *.blend*-файл. Функция *listfiles()* использует функцию *walk()* из модуля Питона *os*, чтобы рекурсивно обойти дерево каталогов и построить список файлов при обходе.

По умолчанию, функция *walk()* проходит по дереву каталогов первой глубины, что позволяет нам изменять список каталогов на лету. Эта возможность используется здесь, чтобы удалить любые каталоги, которые начинаются с точки (выделено). Это не необходимо для текущего и родительского каталогов (они представлены посредством *..* и *.* соответственно), поскольку *walk()* уже фильтрует их, но это позволяет нам, например, также отфильтровать любые *.svn* каталоги, которые могут нам встретиться.

Строка, содержащая оператор *yield*, возвращает как результат один файл за один раз, так что наша функция может быть использована как итератор. (Для дополнительной информации об итераторах, смотрите [online-документацию по адресу http://docs.python.org/reference/simple\\_stmts.html#yield](http://docs.python.org/reference/simple_stmts.html#yield)) Мы соединяем

соответствующее имя файла и путь, чтобы сформировать полное имя, и нормализуем его (то есть, удаляем двойные разделители пути и тому подобное); хотя нормализация здесь не строго необходима, поскольку *walk()* должна возвращать любые пути в нормализованной форме:

```
def listfiles(dir):
    for root,dirs,files in walk(dir):
        for file in files:
            if not file.startswith('.'):
                yield os.path.normpath(
                    os.path.join(root, file))
        for d in dirs:
            if d.startswith('.'):
                dirs.remove(d)
```

Прежде, чем мы сможем сравнить список файлов, которые используются нашим *.blend*-файлом со списком файлов, присутствующих в каталоге, мы убеждаемся, что любой упакованный файл распакован на свое первоначальное местоположение. Не строго необходимо, но позволяет удостовериться, что мы не перемещаем в архив никаких файлов, которые непосредственно не используются, но имеют копию в *.blend*-файле:

```
def run():
    Blender.UnpackAll(Blender.UnpackModes.USE_ORIGINAL)
```

Функция *GetPaths()* из модуля *Blender* выдаёт список всех файлов, используемых *.blend*-файлом (за исключением самого этого *.blend*-файла). Мы передаем ей аргумент *absolute* установленным в Истину, чтобы извлекать имена файлов с полным путём вместо относительных путей от текущего каталога для того, чтобы сравнить их должным образом со списком, произведённым функцией *listfiles()*.

Снова мы также нормализуем эти имена файлов. Выделенная строка показывает, как мы извлекаем абсолютный путь текущего каталога, передавая условное обозначение для текущего каталога Блендера (*//*) в функцию *expandpath()*:

```
files = [os.path.normpath(f) for f in
          Blender.GetPaths(absolute=True)]
currentdir = Blender.sys.expandpath('//')
```

Затем мы создаём объект *ZipFile* в режиме *write* (записи). Это отбросит любой существующий архив с тем же именем, и позволит нам добавлять файлы в архив. Полное имя архива строится соединением текущего каталога Блендера и имени, которое мы хотим использовать для архива. Использование функции *join()* из модуля *os.path* обеспечивает нам создание полного имени платформо-независимым образом. Мы установили аргумент *debug* (отладка) объекта *ZipFile* в значение 3, чтобы сообщать о чём-либо необычном на консоль при создании архива:

```
zip = ZipFile(os.path.join(currentdir, zipname), 'w')
zip.debug = 3
```

В переменную *removefiles* (удаление файлов) записываются имена файлов, которые мы хотим удалить после того, как мы создали архив. Мы можем безопасно удалить файлы и каталоги только после того, как мы создали архив, иначе может оказаться, что мы ссылаемся на каталоги, которые больше не существуют.

Архив создаётся проходом цикла по списку всех файлов в текущем каталоге Блендера и сравнением их со списком файлов, использованных нашим *.blend*-файлом. Любой файл с таким расширением, как например, *.blend* или *.blend1* пропускается (выделено), как и сам архив. Файлы добавляются к ZIP-файлу использованием метода *write()*, который принимает в качестве параметра имя файла с путём относительно архива (и, следовательно, текущего каталога). Этот путь удобнее для распаковки архива в новом месте. Любые ссылки на файлы за пределами текущего дерева каталогов не затрагиваются функцией *relpath()*. Любой файл, который мы добавляем к архиву, помечается для удаления добавлением его к списку *removefiles*. Наконец, мы закрываем архив - важный шаг, поскольку, если его опустить, мы можем остаться с заперченным архивом:

```
removefiles = []
for f in listfiles(currentdir):
    if not (f in files
            or os.path.splitext(f)[1].startswith('.blend')
            or os.path.basename(f) == zipname):
        rf = os.path.relpath(f, currentdir)
        zip.write(rf)
        removefiles.append(f)
```

```
zip.close()
```

Последней задачей будет удаление файлов, которые мы переместили в архив. Функция *remove()* из модуля Питона *os* выполнит это, но мы также хотим удалить любой каталог, который остался пустым после удаления файлов. Следовательно, для каждого файла, который мы удаляем, нам надо определить имя его каталога. Мы также удостоверяемся, этот каталог не указывает на текущий каталог, потому что мы хотим быть абсолютно уверены, что мы не удаляем его, так как это место, где находятся наши *.blend*-файлы. Хотя это маловероятный сценарий, что можно открыть *.blend*-файл в Блендере и удалить сам этот *.blend* файл, что могло бы оставить каталог пустым. Если мы удалим этот каталог, любое последующее (авто) сохранение должно потерпеть неудачу. Функция *relpath()* возвращает точку, если каталог, переданный как первый аргумент, указывает на тот же каталог, что и каталог, переданный как второй аргумент. (Функция *samefile()* является более надёжной и прямой, но не доступна в Windows.)

Если мы убедились, что мы не ссылаемся на текущий каталог, мы используем функцию *removedirs()*, чтобы удалить каталог. Если каталог не пуст, произойдёт ошибка с исключением *OSError* (то есть, файл, который мы удалили, был не последним файлом в каталоге), которую мы игнорируем. Функция *removedirs()* также удалит все родительские каталоги, ведущие к каталогу только тогда, когда они пустые, и это как раз то, что нам нужно:

```
for f in removefiles:
    remove(f)
    d = os.path.dirname(f)
    if os.path.relpath(d, currentdir) != '.':
        try:
            removedirs(d)
        except OSError:
            pass
```

```
if __name__ == '__main__':
```

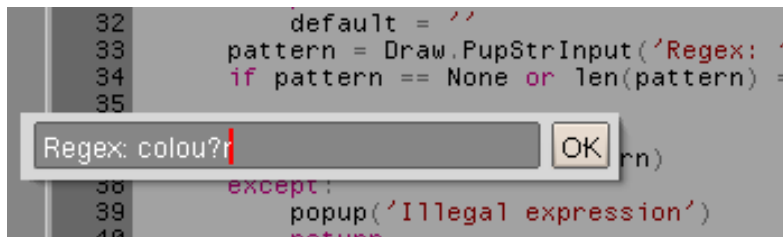
Полный код доступен как *zip.py* в файле *attic.blend*.

## Расширение редактора - поиск с регулярными выражениями

Редактор уже обеспечивает функциональность базового поиска и замены, но если Вы пользовались другими редакторами, Вы могли пропустить возможность поиска с использованием **регулярных выражений**. Этот плагин обеспечивает такую функциональность.

Регулярные выражения очень мощны и множество программистов любят их универсальность (и множество других ненавидят их ужасную неудобочитаемость). Любите Вы или ненавидите их, они очень выразительные: сопоставление любого десятичного числа можно просто выразить как, например, `\d+` (одна или более цифр). Если Вы ищете слово, которое пишется по буквам по-разному в Британском или Американском вариантах английского, как например, `colour/color`, Вы можете делать сопоставление с любым из них с помощью выражения `colou?r` (`color` с необязательным `u`).

Следующий код покажет, что встроенный редактор Блендера может быть оснащён этим полезным средством поиска просто с помощью нескольких строк кода. Представленный скрипт должен быть установлен в каталоге скриптов Блендера, и его можно будет затем вызывать из меню текстового редактора как **Text | Text Plugins | Regular Expression Search**, или комбинацией горячих клавиш `Alt + Ctrl + R`. При этом появится небольшое всплывающее поле ввода, где пользователь может ввести регулярное выражение (там будет запомнено последнее введенное регулярное выражение), и если пользователь щелкнет по кнопке ОК или нажмёт Enter, курсор будет установлен в первом из мест, которые соответствуют регулярному выражению, с выделением сопоставленного выражения.



Чтобы зарегистрировать скрипт в качестве текстового плагина с назначенной горячей клавишей, первые строки скрипта состоят из привычных заголовков, дополненных пунктом *Shortcut*: (выделено ниже):

```

#!BPY
"""
Name: 'Regular Expression Search'
Blender: 249
Group: 'TextPlugin'
Shortcut: 'Ctrl+Alt+R'
Tooltip: 'Find text matching a regular expression'
"""

```

Следующим шагом нужно импортировать необходимые модули. Питон предоставляет нам стандартный модуль *re*, который хорошо документирован (онлайн документации достаточно даже для пользователей-новичков, незнакомых с регулярными выражениями. По-русски почитать можно, например, здесь: <http://www.intuit.ru/department/pl/python/6/4.html> — прим. пер.), и мы импортируем модуль Блендера *bpy*. В этой книге мы не часто используем этот модуль, так как он помечен, как экспериментальный, но в этом случае мы нуждаемся в нём, чтобы узнать, какой текстовый буфер является активным:

```

from Blender import Draw, Text, Registry
import bpy
import re

```

Для того, чтобы сигнализировать о любых ошибках, как например, ошибочное регулярное выражение, или о том, что не нашлось ни одного сопоставления, мы определяем простую функцию *popup()*:

```

def popup(msg):
    Draw.PupMenu(msg+'%t|Ok')
    return

```

Поскольку мы хотим помнить последнее регулярное выражение, которое ввёл пользователь, мы используем реестр Блендера и, следовательно, мы определяем ключ для использования:

```

keyname = 'regex'

```

Функция *run()* связывает всю функциональность вместе; она извлекает активный текстовый буфер и завершается, если его не нашлось:

```
def run():
```

```
    txt = bpy.data.texts.active
    if not txt: return
```

Далее, она извлекает позицию курсора внутри этого буфера:

```
    row,col = txt.getCursorPos()
```

Прежде, чем показать пользователю всплывающее окно для ввода регулярного выражения, мы проверяем, есть ли уже сохраненное ранее выражение в реестре. Мы просто извлекаем его, и если это терпит неудачу, мы ставим выражением по-умолчанию пустую строку (выделено). Заметьте, что мы не передаем никаких дополнительных параметров в функцию *GetKey()*, поскольку мы хотим сохранить любую информацию на диск в этом случае. Если пользователь вводит пустую строку, мы просто делаем возврат без поиска:

```
    d=Registry.GetKey(keyname)
    try:
        default = d['regex']
    except:
        default = ''
    pattern = Draw.PupStrInput('Regex: ',default,40)
    if pattern == None or len(pattern) == 0 : return
```

Мы компилируем регулярное выражение, чтобы убедиться, что оно корректно, и если это терпит неудачу, мы показываем сообщение и выходим:

```
    try:
        po = re.compile(pattern)
    except:
        popup('Illegal expression')
        return
```

Теперь, когда мы уверены, что регулярное выражение - верное, мы проходим по всем строкам текстового буфера, начиная со строки, на которой находится курсор (выделено). С каждой строкой мы сопоставляем наше скомпилированное регулярное выражение (или с частью строки после курсора, если это первая строка).

```
    first = True
    for string in txt.asLines(row):
        if first :
            string = string[col:]
            mo = re.search(po,string)
```

Если есть сопоставление, мы отмечаем его начало в пределах строки и его длину (должным образом исправленную, если это строка первая) и устанавливаем позицию курсора на текущую строку и в начало сопоставления (выделено). Мы также устанавливаем "позицию выделения" в позицию сопоставления плюс длина сопоставления, таким образом наше сопоставление будет выделено, и затем делаем возврат. Если нет сопоставления в пределах строки, мы увеличиваем индекс строки *row* и продолжаем цикл.

Если ничего не остается для перебора, мы сигнализируем пользователю, что мы не нашли ни одного сопоставления. В любом случае, мы сохраняем регулярное выражение в реестре для использования заново:

```
        if mo != None :
            i = mo.start()
            l = mo.end()-i
            if first :
                i += col
            txt.setCursorPos(row,i)
            txt.setSelectPos(row,i+l)
            break
        row += 1
        first = False
```

```
    else :
        popup('No match')
        Registry.SetKey(keyname,{'regex':pattern})
    if __name__ == '__main__':
        run()
```

Полный код доступен как *regex.py* в файле *regex.blend*, но может быть размещён в каталоге скриптов Блендер с подходящим именем, как например, *textplugin\_regex.py*.



## Расширение редактора - взаимодействие с Subversion

При активной разработке скриптов может оказаться сложно следить за изменениями или возвращаться к предыдущим версиям. Это не уникально для написания скриптов Питона в Блендере, поэтому системы **управления версиями** развиваются уже много лет. Одна из хорошо известных, и широко используемых - это **Subversion** (<http://subversion.tigris.org>). В этом разделе мы показываем, как может быть дополнен редактор, чтобы отправлять или обновлять текстовые файлы из хранилища.

Взаимодействие с хранилищем Subversion не предусмотрено встроенными модулями Питона, так что мы должны получить эту библиотеку где-нибудь еще. Секция загрузок сайта <http://pysvn.tigris.org> содержит и исходные коды и бинарные дистрибутивы для многих платформ. Не забудьте получить правильную версию, так как поддерживаемая версия Subversion и версия Питона могут отличаться. Скрипт, который мы разрабатывали здесь, протестирован на Subversion 1.6.x и Питоне 2.6.x, но должен также работать с более ранними версиями Subversion.

Мы осуществим функциональность отправления (*commit*) текстового файла в хранилище и обновления (*update*) файла (то есть, получение самой последней исправленной версии из хранилища). Если мы пытаемся отправить файл, который пока не является частью хранилища, мы добавляем его, но мы не будем разрабатывать инструменты для создания хранилища или проверки рабочей копии. Такие инструменты, как, например, **TortoiseSVN** в Windows (<http://tortoisesvn.tigris.org/>) или множество инструментов для открытых платформ значительно лучше это делают. Мы просто принимаем подтвержденный (*checked-out*) рабочий каталог, где мы храним наши текстовые файлы Блендера. (Этот рабочий каталог может отличаться от вашего каталога проекта Блендера.)

### Отправка (commit) файла в хранилище

Отправка текстового буфера в хранилище - процесс из двух шагов. Сначала мы должны сохранить содержимое текстового буфера в файл, и затем мы отправляем этот файл в хранилище. Мы

должны проверить, имеет ли текстовый блок связанное с ним имя файла, и предложить пользователю сначала сохранить файл, если такого файла пока ещё нет. Пользователь должен сохранить файл в подтвержденный каталог для того, чтобы отправить файл в хранилище.

Так же как и расширение, позволившее нам производить поиск с помощью регулярных выражений, этот скрипт начинается с подходящего заголовка, чтобы идентифицировать его как плагин текстового редактора, и чтобы назначить клавиатурное сокращение. Мы определяем мнемосхему *Ctrl + Alt + C* для отправки (выделено), так же как мы определим *Ctrl + Alt + U* для обновления в своем сопровождающем скрипте. Мы также импортируем необходимые модули, особенно модуль *pysvn*:

```
#!/BPY
"""
Name: 'SVNCommit'
Blender: 249
Group: 'TextPlugin'
Shortcut: 'Ctrl+Alt+C'
Tooltip: 'Commit current textbuffer to svn'
"""
```

```
from Blender import Draw, Text, Registry
import bpy
import pysvn
```

```
def popup(msg):
    Draw.PupMenu(msg+'%t|Ok')
    return
```

Функция *run()* сначала пытается получить активный текстовый буфер и возвращается без брызжания, если там его нет. Затем она проверяет, существует ли имя файла, определенное для этого текстового буфера (выделено). Если нет, она напоминает пользователю, что надо сначала сохранить файл (таким образом, определяя имя файла и располагая файл в подтвержденном каталоге), и возвращается.

```
def run():

    txt = bpy.data.texts.active
```

```

if not txt: return

fn = txt.getFilename()
if fn == None or len(fn) == 0:
    popup('No filename defined: save it first')
    return

```

Следующим шагом нужно создать объект клиента *pysvn*, который позволит нам взаимодействовать с хранилищем. Метод *info()* извлекает информацию о статусе файла в хранилище (выделено). Если нет никакой информации, значит файл пока не был добавлен к хранилищу - ситуация, которую мы исправляем, вызывая метод *add()*:

```

svn = pysvn.Client()
info = svn.info(fn)
if info == None:
    popup('not yet added to repository, '+ \
        'will do that now')
    svn.add(fn)

```

Затем, мы сводим текущее содержимое текстового буфера, соединяя все строки в нём в единственный блок данных, и записываем его в файловый объект, который мы открыли для файла, связанного с буфером:

```

file=open(fn,'wb')
file.write('\n'.join(txt.asLines()))
file.close()

```

Этот файл будет отправлен в хранилище с помощью метода *checkin()*, которому мы передаем довольно неинформативное сообщение отправки. Было бы хорошей идеей предложить пользователю создать более заметное сообщение. Наконец, мы сообщаем пользователю результат отправки.



Заметьте, что номера версии в Subversion (Subversion revision) связаны не с файлом, а с хранилищем, так что это число может отличаться больше, чем на единицу от предыдущего переданного файла, если за это время была совершена передача других файлов.

```

version = svn.checkin(fn,'Blender commit')
popup('updated to rev. '+str(version))

```

```

if __name__ == '__main__':
    run()

```

Полный код доступен как *textplugin\_commit* в файле *svn.blend*, но должен быть установлен в каталоге скриптов Блендера.

## Обновление (updating) файла из хранилища

Основная цель хранилища - возможность сотрудничать, что означает, что другие пользователи могут изменить файлы, с которыми мы работаем, и мы должны быть в состоянии получить эти совершенные изменения. Это называется обновление (updating) файла и означает, что мы копируем самую последнюю версию, которая находится в хранилище в наш рабочий каталог.

Кроме проверки, сохранен ли текстовый буфер, и добавлен ли уже файл к хранилищу, мы должны также проверить, является ли наша текущая версия - новой или измененной по сравнению с версией в хранилище. Если это так, мы предлагаем пользователю выбор: отвергнуть эти изменения и вернуться к версии в хранилище, или подтвердить и отправить версию, находящуюся в текстовом буфере. (Третий вариант, объединение различий, у нас не предусмотрен; хотя Subversion, несомненно, способно сделать это, по крайней мере для текстовых файлов, но лучше это предоставить более универсальным инструментам, таким как TortoiseSVN.)

Первая часть скрипта очень похожа на скрипт отправки. Основное различие - это другое клавиатурное сокращение:

```

#!BPY
"""
Name: 'SVNUpdate'
Blender: 249
Group: 'TextPlugin'
Shortcut: 'Ctrl+Alt+U'
Tooltip: 'Update current textbuffer from svn'
"""

from Blender import Draw,Text,Registry
import bpy
import re

```

```
import pysvn
def popup(msg):
    Draw.PupMenu(msg+'%t|Ok')
    return
```

Функция *run()* тоже начинается почти также, она извлекает активный текстовый буфер (если он есть) и проверяет, имеет ли текстовый буфер связанное имя файла (выделено). Она также проверяет, было ли имя файла уже добавлено к хранилищу, и если нет, исправляет это, вызывая метод *add()*, и сообщает об этом пользователю посредством всплывающего окна:

```
def run():

    txt = bpy.data.texts.active
    if not txt: return

    fn = txt.getFilename()
    if fn == None or len(fn) == 0:
        popup('No filename defined: save it first')
        return
    svn = pysvn.Client()
    info = svn.info(fn)
    if info == None:
        popup('not yet added to repository, '+ \
            'will do that now')
        svn.add(fn)
```

После сохранения содержимого текстового буфера в связанный с ним файл, функция вызывает метод *status()*, чтобы убедиться, что файл, который мы сохранили (и, следовательно, содержание текстового буфера), изменён по сравнению с версией в хранилище (выделено). В метод *status()* можно также передавать список имён файлов, и он всегда возвращает список результатов, даже когда мы передали ему простое одиночное имя файла - поэтому применяется индекс *[0]*. Если наш текстовый буфер изменён, мы сообщаем об этом пользователю, и предлагаем выбор: или отвергнуть изменения и извлечь версию, сохранённую в хранилище, или отправить текущую версию. Также возможно отменить оба действия, щелкнув за пределами меню, в этом случае *PupMenu()* возвращает -1:

```
file=open(fn, 'wb')
file.write('\n'.join(txt.asLines()))
file.close()
```

```
if svn.status(fn)[0].text_status ==
pysvn.wc_status_kind.modified:
    c=Draw.PupMenu('file probably newer than '+ \
        'version in repository%t|Commit|Discard changes')
    if c==1:
        svn.checkin(fn, 'Blender')
        return
    elif c==2:
        svn.revert(fn)
```

После извлечения версии из хранилища мы обновляем содержание нашего текстового буфера:

```
txt.clear()
file=open(fn)
txt.write(file.read())
file.close()
```

Наконец, мы сообщаем пользователю с помощью всплывающего окна, какой номер версии содержится в текстовом буфере, снова вызывая метод *status()* и получая значение поля *commit\_revision*:

```
popup('updated to rev. '
    +str(svn.status(fn)[0].entry.commit_revision))
```

```
if __name__ == '__main__':
    run()
```

Полный код доступен как *textplugin\_svnupdate* в файле *svn.blend*, и, подобно сопряженному с ним скриптом для отправки, он должен быть размещён в каталоге скриптов Блендера.

## Работа с хранилищем

Хотя полный урок по работе с Subversion выходит за рамки этой книги, вероятно, будет полезным набросать схему рабочего процесса для проекта Блендера, в котором скриптовые компоненты пишутся через систему контроля версий.

Важно понимать, что сам проект Блендера не должен находиться под управлением системы контроля версий. Мы можем организовать наш проект Блендера любым способом, который имеет

смысл, и расположить каталог *scripts* в его пределах, он и будет находиться под управлением системы контроля версиями.

Скажем, что мы создали хранилище для скриптов на сетевом устройстве хранения, и создали каталог проекта Блендера на нашей локальной машине. Для того, чтобы перевести наши скрипты под управление версиями, мы должны выполнить следующие шаги:

1. Подтвердить (Check out) хранилище скриптов внутри нашего каталога проекта Блендера (это называется **рабочая копия** хранилища).
2. Создать скрипт в нашем *.blend* файле во встроенном редакторе.
3. Сохранить этот скрипт в рабочую копию.
4. Каждый раз, когда мы что-то изменяем, мы нажимаем *Ctrl + Alt + C*, чтобы отправить наши изменения.
5. Каждый раз, когда мы начинаем работать с нашим скриптом снова, мы нажимаем сначала *Ctrl + Alt + U*, чтобы сразу увидеть, не изменил ли кто-нибудь еще что-нибудь.

Обратите внимание, что ничто не мешает включить все активы, такие как текстуры или *.blend* файлы, чтобы они выступали в роли библиотек под контролем версий, но нам понадобится использовать отдельный клиент для фиксации изменений. Это будет интересным упражнением, создать несколько скриптов, которые отправляют или обновляют все файлы в текущем каталоге Блендера.

## ***The need for speed (жажда скорости) — использование Psycos***

Питон является интерпретируемым языком: все инструкции в скрипте интерпретируются и выполняются снова и снова, когда они встречаются. Это может звучать неэффективным, но для разработчиков программ преимущество возможности быстро разработать и протестировать программу может перевесить недостаток медленного выполнения программы. И интерпретация может быть неэффективной, но, это не идентично тому, что она всегда медленная. Питон является очень высокоуровневым языком, в

котором единственный языковой элемент может быть эквивалентом множества низкоуровневых инструкций. Кроме того, с учетом современного аппаратного обеспечения, даже медленный скрипт может закончить работу быстрее, чем пользователь ожидает результат.

Тем не менее, существуют ситуации, где любое увеличение скорости весьма приветствуется. Из всех примеров, которые мы видели в этой книге, PyNodes, вероятно, наиболее интенсивные в вычислительном отношении, так как инструкции выполняются над каждым видимым пикселем в текстуре или шейдере, и часто даже много больше времени уходит на пиксель, если принять во внимание oversampling. Экономия нескольких миллисекунд от скрипта, который тратит меньше секунды на выполнение, не даст слишком многого, но экономия 20% времени рендера составит существенную экономию времени при рендере 500 кадров.

**Ввод Psycos:** Psycos - расширение Питона, которое пытается ускорять выполнение скрипта, компилируя часто используемые части скрипта в машинные инструкции, и сохраняя их для многократного использования. Этот процесс часто называется **компиляция-на-ленту** (just-in-time compilation, JIT), и она родственна JIT-компиляторам на других языках, таких как Java. (Они аналогичны по концепции, но совершенно отличаются в реализации из-за того, что в Питоне динамическая типизация. Это никак не затрагивает разработчиков скриптов на Питоне.) Важно то, что Psycos может быть использовано в любом скрипте без каких-либо изменений в коде, за исключением добавления нескольких строк.

Psycos доступен как бинарный пакет для Windows, и может быть скомпилирован из исходных кодов на других платформах. Полные инструкции доступны на вебсайте Psycos: <http://psyco.sourceforge.net/>.

Проверьте, что Вы устанавливаете версию, которая подходит вашей версии Питона, поскольку, хотя сайт указывает, что версия, скомпилированная для Питона 2.5 должна работать также для 2.6, она все еще может потерпеть неудачу, поэтому лучше использовать версию, специально скомпилированную для 2.6.

На сайте <http://psyco.sourceforge.net/> я не смог найти бинарной версии Psycos для Питона 2.6 под Windows (в Линуксе-то его легко

самостоятельно скомпилировать). Но Гугл помог — такая версия обнаружилась [здесь: http://www.voidspace.org.uk/python/modules.shtml#psyco](http://www.voidspace.org.uk/python/modules.shtml#psyco). Вполне рабочая. Ввиду фактического умирания проекта Psyco (в википедии написано, что его преемником является PyPy – странная штука «Питон-на-Питоне», в данный момент не слишком работоспособная), сомневаюсь, что он когда-нибудь заработает для 3-го Питона, и, соответственно, для новых версий Блендера. - прим. пер.

Итак, какое увеличение скорости мы могли бы ожидать? Это трудно оценить, но достаточно легко измерить! Просто рендерите кадр и отмечайте время, которое потребовалось, затем импортируйте psyco в ваш код, рендерите снова и отмечайте различие. Если оно значимое, оставляйте в коде, в противном случае, Вы можете снова его удалить.

На следующей таблице указаны некоторые результаты для тестовой сцены, приведенной в *psyco.blend*, но ваши данные могут отличаться. Также заметьте, что тестовая сцена является довольно оптимистическим сценарием, так как большая часть оказалась покрыта текстурой, генерируемой Pynode. Если бы её было меньше, прирост в скорости бы уменьшился, но это дает оценку того, что возможно с Psyco. Показатель в два раза для важного кода легко достижим. В следующей таблице перечислены некоторые иллюстрирующие примеры времени расчёта:

Время в секундах	Без Psyco	С Psyco
Нетбук	52.7	26.3
Стационарный компьютер	14.01	6.98

## Включение Psyco

Следующий код показывает дополнительные строки, которые нужны для включения psyco в нашем ранее встречавшемся Pynode raindrops (капли дождя). Изменения указаны жирным шрифтом.

<... все остальные строки остаются прежними ...>

```
__node__ = Raindrops

try:
    import psyco
    psyco.bind(Raindrops.__call__)
    print 'Psyco configured'
except ImportError:
    print 'Psycho not configured, continuing'
    pass
```

Так что, по сути, было добавлено только несколько строк после определения Pynode. Убедитесь, что вы щелкнули на кнопке Update (обновить) на Pynode, иначе код не будет перекомпилирован, и изменения не будут видны.

Предшествующий код просто пытается импортировать модуль psyco. Если это терпит неудачу (по любой причине), в консоли выводится информационное сообщение, но, тем не менее, код будет работать правильно. Если он импортируется, мы указываем Psyco оптимизировать метод *\_\_call\_\_()*, вызывая функцию *bind()* со ссылкой на этот метод *\_\_call\_\_* в качестве аргумента, и сообщаем пользователю в консоли, что мы успешно сконфигурировали Psyco.

## **Итог**

В этой главе мы смотрели за пределы 3D и рендера, и увидели как сделать жизнь счастливее для разработчика на Питоне и художника, предоставляя некоторые скрипты, помогающие в нескольких общих служебных задачах, расширяя функциональность встроенного редактора поиском с регулярными выражениями и системой управления версиями, и показали как экономить ценное время рендера в некоторых ситуациях, используя Psyc0. В частности, мы узнали:

- Как построить список активов, таких как карты изображений, и заархивировать их
- Как опубликовать отрендеренные изображения автоматически через FTP
- Как расширить функциональность встроенного редактора поиском с регулярными выражениями
- Как ускорить вычисления, используя Psyc0 - компилятор-на-лету
- Как добавить управление версиями к вашим скриптам с помощью Subversion