VRML файлы имеют расширения **wrl** (от слова world - "мир") или wrz. В обоих случаях файл может быть либо текстовым (содержащим непосредственно код), либо gzip-архивом.

Единицы измерения

В VRML приняты следующие единицы измерения:

- Расстояние и размер: метры
- Углы: радианы
- Остальные значения: выражаются, как часть от 1.
- Координаты берутся в трехмерной декартовой системе координат (см. рис.)

Z-координата направлена из экрана. Причем объект определяется только снаружи. Изнутри вид объекта стандартом не определен и зависит от браузера.

Заголовок файла в VRML97

Каждый файл с кодом VRML97 должен начинаться строкой

```
#VRML V2.0 utf8
```

Такой заголовок обязательно должен находиться в первой строке файла, кроме того, перед знаком диеза не должно быть пробелов.

Все строки, кроме первой, начинающиеся значком #, считаются комментариями.

Операторы, из которых состоит код, называются узлами (**node**). В общем виде это выглядит примерно так "название узла {аргументы}". В ряде случаев вместо фигурных скобок могут стоят квадратные "[]".

Примитивы VRML97

Box (параллелепипед)

```
Box {size 2 2 2}
```

По умолчанию центр тяжести помещается в 0,0,0.

Таким образом код кубика 2x2x2 будет выглядеть так

```
#VRML V2.0 utf8
Shape {geometry Box {size 2 2 2}}
```

Узел Shape указывает броузеру, что идет описание формы объекта.

Сфера

```
Sphere {radius 1}
```

По умолчанию центр сферы помещается в точку 0,0,0.

Конус

```
Cone {bottomRadius 1
height 2
side TRUE
bottom TRUE }
```

По умолчанию центр его высоты цилиндра размещается в 0,0,0. Логическое утверждение TRUE/FALSE (ИСТИНА/ЛОЖЬ) для side и bottom определяет, будет ли создаваться сторона и донышко конуса.

Цилиндр

```
Cylinder {bottom TRUE
                        height 2
                         radius 1
                         side TRUE
                         top TRUE }
```

По умолчанию центр высоты цилиндра размещается в 0,0,0. Логическое утверждение TRUE/FALSE (ИСТИНА/ЛОЖЬ) для side, bottom, top определяет, будет ли создаваться сторона, нижняя и верхняя крышки цилиндра.

Объекты, строящиеся по набору вершин

PointSet (набор точек)

```
#VRML V2.0 utf8


Shape {
geometry PointSet {
coord Coordinate { point [ 0 0 0, 1 0 0, 2 0 0, 3 0 0, 4 0 0, 5 0 0,
0 1 0, 1 1 0, 2 1 0, 3 1 0, 4 1 0, 5 1 0,
0 2 0, 1 2 0, 2 2 0, 3 2 0, 4 2 0, 5 2 0,
0 3 0, 1 3 0, 2 3 0, 3 3 0, 4 3 0, 5 3 0,
0 4 0, 1 4 0, 2 4 0, 3 4 0, 4 4 0, 5 4 0,
0 5 0, 1 5 0, 2 5 0, 3 5 0, 4 5 0, 5 5 0
] }
color Color { color [ 1 0 0, 0 1 0, 0 0 1, 1 0 0, 0 1 0, 0 0 1,
0 1 0, 0 0 1, 1 0 0, 0 1 0, 0 0 1, 1 0 0,
0 0 1, 1 0 0, 0 1 0, 0 0 1, 1 0 0, 0 1 0,
1 0 0, 0 1 0, 0 0 1, 1 0 0, 0 1 0, 0 0 1,
0 1 0, 0 0 1, 1 0 0, 0 1 0, 0 0 1, 1 0 0,
0 0 1, 1 0 0, 0 1 0, 0 0 1, 1 0 0, 0 1 0
] }
}
}
```

Количество записей в разделе color должно точно соответствовать количеству точек, описанных в разделе coord!

Это чередующиеся красные, зеленые и синие точки на экране.

IndexedLineSet (линии по набору точек)

```
IndexedLineSet {
color NULL
coord NULL
colorIndex []
colorPerVertex TRUE
coordIndex []
}
```

Схема использования этого узла заключается в следующем: в разделе coord описывается набор точек, а в разделе coordIndex приводятся последовательности точек (их номера в списке раздела coord), которые соединяются отрезками. Конец последовательности обозначается приведением после номера последней точки значения -1.

Если у Вас ничего не прописано в разделе color, то ничего не увидите, т.к. отрезки не окрашены (хотя можно при этом воспользоваться emissiveColor в разделе appearance (об этом см далее))

Если colorPerVertex TRUE, то цвет из списка в разделе color приписывается ВЕРШИНАМ, а отрезки, соединяющие их окрашиваются с градиентом от цвета одной вершины к цвету другой.

Вот, например, "знак Зорро"

```
#VRML V2.0 utf8

Shape {
geometry IndexedLineSet {
colorPerVertex TRUE
coord Coordinate {point [ 0 0 0, 1 0 0, 0 1 0, 1 1 0]}
color Color {color [1 0 0, 0 1 0, 0 0 1, 1 1 1
]}
coordIndex [1 0 3 2 -1
]
}
}
```

Если colorPerVertex FALSE, то цвета из списка в разделе color приписывается всей линии, а не каждому отрезку. В этом случае всей линии будет присвоен красный цвет, а остальные цвета не нужны.

Красный "знак Зорро"

IndexedFaceSet (грани по набору точек)

```
IndexedFaceSet {
coord NULL
color NULL
normal NULL
texCoord NULL
ccw TRUE
colorIndex []
```

```
colorPerVertex TRUE
convex TRUE
coordIndex
creaseAngle 0
normalIndex []
normalPerVertex TRUE
solid TRUE
texCoordIndex []
}
```

Это узел, которым можно заменить все остальные узлы, связанные с описанием граней. При экспорте в VRML код из какой-нибудь программы моделирующей 3D, получается файл, состоящий только из `IndexedFaceSet`. Принцип работы узла очень похож на `IndexedLineSet`: описан набор координат точек (`coord`) и указано, какие из них должны образовать грань (`coordIndex`).

Должны выполняться три условия:

- каждая грань должна состоять как минимум из трех несовпадающих вершин
- вершины должны задавать ПЛОСКИЙ многоугольник
- многоугольник должен быть несамопересекающимся

Легко догадаться, что все условия автоматически выполняются для треугольника, хотя в частном случае можете задавать плоские многоугольники с любым числом вершин.

Раскраска объектов в этом узле происходит так же, как и в IndexedLineSet:

- при `colorPerVertex TRUE` цвет приписывается ВЕРШИНАМ, а грань заливается градиентом между всеми вершинами, которыми грань создана.
- при `colorPerVertex FALSE` цвет приписывается каждой ГРАНИ в порядке, соответствующем порядку цветов в разделе color

Создадим для примера кубик без одной грани средствами узла IndexedFaceSet и раскрасим, пользуясь colorPerVertex TRUE

```
#VRML V2.0 utf8

Shape {geometry IndexedFaceSet {
coord Coordinate {point [ -1 -1 -1, 1 -1 -1, 1 -1 1, -1 -1 1, -1 1 -1, 1 1 -
1, 1 1 1, -1 1 1]}
color Color {color [1 0 0, 0 1 0, 0 0 1, 1 1 0, 1 0 1, 0 1 1, 1 1 1, 0 0 0]}
coordIndex [0 1 2 3 -1 4 5 6 7 -1 0 1 5 4 -1 0 3 7 4 -1 1 2 6 5 -1 ]
solid FALSE
}}
```

Extrusion (экструзия, выдавливание)

```
Extrusion {
beginCap TRUE
ccw TRUE
```

```
convex TRUE
creaseAngle 0
crossSection [1 1, 1 -1, -1 -1, -1 1, 1 1]
endCap TRUE
orientation 0 0 1 0
scale 1 1
solid TRUE
spine [0 0 0, 0 1 0]
}
```

Узел `Extrusion` - это альтернатива `IndexedFaceSet`, позволяющая сильно сократить объем файла. Работает узел очень просто: сначала описывается многоугольник в плоскости с Y=0 (поэтому в разделе `crossSection="сечение"` только две координаты) и траектория его движения в пространстве (раздел `spine`).
Разделы`beginCap` и endCap определяют, будут ли грани-"крышки" на торцах Вашего объекта. В спецификации `solid` написано, что значением этого раздела регулируется, будут ли видны одна или обе стороны многоугольника одновременно.

В каждой точке траектории многоугольник можно:

- масштабировать (раздел `scale`). Обратите внимание, количество значений шкального множителя должно либо 1 (тогда масштабируется исходное сечение`crossSection` и далее не изменяется), либо совпадать с количеством точек в траектории (`spine`).

  Благодаря масштабированию можно делать также всякие остроконечности - нужно только задать "scale ... ,0 0, ..."

  ```
  #VRML V2.0 utf8
  Shape{
  appearance Appearance { material Material {}}
  geometry Extrusion{
  solid FALSE
  crossSection [ -1 -1, 1 -1, 1 1, -1 1, -1 -1]
  spine [0 -1 0 , 0 -0.8 0, 0 -0.5 0, 0 2.2 0, 0 2.5 0, 0 2.8 0, 0 18.8
  0,0 21 0]
  scale [0 0, 0.7 0.6, 0.5 0.2, 0.5 0.2, 3 0.2, 0.7 0.05, 0.8 0.0, 0 0]
  }
  }
  ```

  Просмотр

- вращать (раздел `orientation`). Задается направление оси вращения (первые три числа) и угол в радианах. Вращать можно как по часовой стрелке, так и против (знак угла). Количество значений в разделе orientation определяется так же, как и для масштабирования.

  ```
  #VRML V2.0 utf8
  Transform {
  children
  Shape{ appearance Appearance { material Material {}}
  ```

```
geometry Extrusion{
crossSection [ -1 0, -0.7 -0.7, 0 -1, 0.7 -0.7, 1 0, 0.7 0.7, 0 1, -0.7
0.7, -1 0]
spine [0 -1 0 , 0 1 0 ]
orientation[0 1 0 0, 0 1 0 3.14]
}
}
}
```

Просмотр

ElevationGrid (рельеф по набору точек)

```
ElevationGrid {
color NULL
normal NULL
texCoord NULL
height []
ccw TRUE
colorPerVertex TRUE
creaseAngle 0
normalPerVertex TRUE
solid TRUE
xDimension 0
xSpacing 1.0
zDimension 0
zSpacing 1.0
}
```

Наилучшее применение узла `ElevationGrid` - создание рельефа. Построение ведется следующим образом: представьте себе сетку с прямоугольными ячейками и лежащую в плоскости XZ.

Вы задаете количество ячеек по X и Z (разделы `xDimension` и `zDimension`) и для каждой точки пересечения "волокон" задаете ее "высоту"-координату по Y. Размеры всего объекта и его пропорции вытекают из величины зазоров между "волокнами" (разделы xSpacing и zSpacing).

Просмотр.

Text, FontStyle (Текст, стиль шрифта)

`Text (текст)`

Описание:

```
Text
{  string []
     fontStyle NULL
      length []
      maxExtent 0.0
}
```

По умолчанию текст располагается в плоскости `Z=0` локальной системы координат, т.е. в пределах данного раздела `children`. В разделе `string` прописывается, собственно, строка текста.

О разделе `fontStyle` необходимо сказать отдельно.

Раздел `length`: если `length` больше, чем приведенная в `string` строка, то либо масштабируется текст, либо увеличивается межбуквенное расстояние (в зависимости от браузера).

```
#VRML V2.0 utf8

Shape {
  geometry Text {string ["Hello,","world !"]}
  }
```

Просмотр.

Обратите внимание на разбивку текста в поле string на два куска в кавычках. Это дает размещение текста в две строки.

FontStyle (стиль шрифта)

Описание:

```
FontStyle {
  family "SERIF"
  horizontal TRUE
  justify "BEGIN"
  language " "
  leftToRight TRUE
  size 1.0
  spacing 1.0
  style "PLAIN"
  topToBottom TRUE
}
```

Описание раздела `FontStyle` позволяет несколько отрегулировать вид текста.

По порядку:

- `family` - определяет начертание шрифта. Возможны три значения поля family: SERIF (по умолчанию) - приблизительно соответствует Times Roman, SANS - соответствует шрифту Helvetica, TYPEWRITER - моноширинный шрифт типа Courier'a.
- `horizontal` - это очевидно, что отвечает за написание строки по горизонтали (`horizontal TRUE` - по умолчанию) или по вертикали `(horizontal FALSE)`
- `justify` - то, что в других программах еще называется `alignment` или "выравнивание". Может принимать 4 значения. Первые три: `BEGIN,`

MIDDLE, ENDсоответствуют выравниванию соответственно по левому краю, по центру, по правому краю. Четвертое значение FIRST отвечает за выравнивание по дополнительной оси, т.е. по вертикали, если текст расположен горизонтально и по горизонтали для вертикального текста.

- language - все примеры про текст были английскими фразами. Даже если переключиться в кириллицу и загнать в wrl файл русскоязычную фразу - ничего не выйдет, пока не отрегулировать значение поля language. Делается это как в интернетовских url, т.е. для России - ru

```
#VRML V2.0 utf8

  Shape {
  geometry Text {
  fontStyle FontStyle {
  language "ru"
  }
string "п_я_пёп_пчя', п_пёя_!"
  } }
```

Просмотр. (Привет, мир!)

Приведенный код нормально сработал в случае CosmoPlayer, MS VRML Viewer, Cortona и не сработал в GLView. Непонятно в какой кодировке представлен текст, это ведь вовсе не CP-1251.

- size и spacing - задают размер шрифта и межстрочное расстояние
- style - может быть PLAIN, BOLD, ITALIC, BOLDITALIC. Это всем знакомые обычный, полужирный, курсив, полужирный курсив.
- leftToRight и topToBottom - в зависимости от значения horizontal задает направления написания текста. При horizontal TRUE и leftToRight FALSE текст напишется справа налево. Аналогично по вертикали.

Вместо использования Text+FontStyle есть два более приятных варианта:

- использование текстуры. Напишите в Photoshope все что Вам вздумается, любым шрифтом, с любыми эффектами и налепите эту "листовку" на объект. Как это делается см в разделе "Текстуры"
- трехмерный текст. Удобнее всего такое делать в Internet Space Builder.

И в первом, и во втором случае никаких проблем с кодировкой, языком и т.д.!

Наложение текстур

Процедура наложения текстур состоит из двух частей, которыми занимаются соответствующие узлы.

Узлы ImageTexture, MovieTexture, PixelTextureописывают что использовать в качестве текстуры, а вспомогательные узлы TextureCoordinate и TextureTransform задают как разместить текстуру на объекте. Аналогичная ситуация уже встречалась в разделе "Текст", где узел Text задавал что написать, а FontStyle задавал как это сделать.

ImageTexture (текстура-картинка)

Описание:

```
ImageTexture {
  url [ ]
  repeatS TRUE
  repeatT TRUE
  }
```

Узел ImageTexture - очень полезный узел, предназначенный для обтягивания объектов картинками. Область использования этого приема исключительно широка: начиная от создания аватара с собственной фотографией и заканчивая привешиванием картинок на background. В разделе url указывается местоположения графического файла.

Самой текстуре присваивается локальная система координат S на T, где S соответсвует нижнему обрезу картинки, а T - левому. Значения S и T изменяются от 0 (левый нижний пиксель) до 1 (правый нижний пиксель для S и левый верхний пиксель для T). Значения параметров repeatS и repeatT определяют, будет ли текстура размножена в направлениях S и T, чтобы заполнить весь объект.

Теперь о форматах. Броузерам строго предписано поддерживать jpeg и png. Но кроме того обычно поддерживаются gif и ряд других распространенных форматов.

Следующие броузеры поддерживают:

- **MS VRML 2.0 Viewer**: GIF, BMP, JPG, RAS, PPM, PNG.
- **Cosmoplayer**: в release notes указаны только JPEG и PNG, но GIF точно поддерживается.
- **Paragraph Cortona**: вероятно BMP, GIF, JPEG, и PNG.
- **GLView**: DIB,BMP, GIF, TGA, JPEG, PPM and RGB

MovieTexture (текстура-видеоролик)

Описание:

```
MovieTexture {
  loop FALSE
  speed 1.0
  startTime 0
  stopTime 0
```

```
  url []
  repeatS TRUE
  repeatT TRUE
  }
```

Данный узел в качестве текстуры задает файл в формате MPEG. Поскольку этот формат позволяет хранить как видео-, так и аудиоинформацию, соответственно узелMovieTexture может появляться и в разделе texture узла Appearance, и в разделе source узла Sound (о нем позже). В последнем случае, естественно, рендеринг изображения не проводится, а обрабатывается только саундтрек файла.

- loop, как обычно задает, будет ли файл прокручиваться бесконечно.
- Параметр speed позволяет задавать скорость проигрывания MPEG файла. В случае speed <0 проигрывание идет в обратном порядке. В случае speed=0 будет статично отображаться 0-й фрейм MPEG файла.
- startTime и stopTime определяют обработку проигрывания во времени MPEG файла и обычно используются для запуска/остановки в определенный момент или после определенного события.
- Параметры url, repeatS и repeatT занимаются тем же, что и в узле ImageTexture

PixelTexture (пиксельная текстура)

Описание:

```
PixelTexture {
  image 0 0 0
  repeatS TRUE
  repeatT TRUE
  }
```

Механизм работы данного узла такой же, как и у ImageTexture, кроме того, что Вы указываете не готовую картинку из файла, а должны ручками раскрасить каждый указанный Вами пиксель. Ситуации, когда Вам такой выход покажется оптимальным, единичны. Что такое repeatS и repeatT читайте выше.

В параметре image первые два числа задают размер текстуры в пикселях, третье - способ раскрашивания: 1 - градации серого, 2 - градации серого плюс прозрачность, 3 - цветное изображение, 4 - цветное изображение плюс прозрачность.

Таким образом, запись

```
PixelTexture {image 2 2 1 0 255 255 0}
```

задает текстуру размером 2x2, раскрашенную в шахматном порядке белым и черным.

```
#VRML V2.0 utf8

Shape{
appearance Appearance {texture PixelTexture { image 2 2 1
0 255 255 0}}
geometry Box{}
}
```

Просмотр.

А запись

```
PixelTexture {image 2 2 4 0xff000080 0x00ff0080
0x0000ff80 0x00000080}
```

задает текстуру в полупрозрачную клеточку красного, зеленого, синего и черного цвета.
Можете посмотреть кубик с такой текстурой. Для иллюстрации полупрозрачности текстуры внутри расположен белый непрозрачный шарик. Если Вы его не видите в своем броузере, переключитесь в режим `wireframe` и убедитесь, что он там есть. Кстати, даже если Вы видите этот шарик сквозь полупрозрачные стенки кубика, не надейтесь разглядеть противоположные грани кубика. Как уже говорилось, вид примитивов изнутри не определен.

```
#VRML V2.0 utf8
Shape{
appearance Appearance {texture PixelTexture {image 2 2 4 0xff000080
0x00ff0080 0x0000ff80 0x00000080}}
geometry Box{}
 }
Shape{
   geometry Sphere{}
  }
```

Просмотр.

TextureTransform (трансформация текстур)

Описание:

```
TextureTransform {
center 0 0
rotation 0
scale 0
translation 0 0
}
```

Как упоминалось в начале этой страницы,
узел `TextureTransform` применяется для указания КАК разместить текстуру на объекте. Названия параметров узла говорят сами за себя: `center` задает

точку, относительно которой происходит вращение (rotation) и масштабирование (scale). Translation сдвигает текстуру по поверхности объекта. Все действия проводятся в локальной ST-системе координат текстуры, о которой смотри выше. Узел TextureTransform, безусловно, жизненно необходим, но пользоваться им путем "ручного" исправления параметров очень неудобно. В любом приличном VRML редакторе трансформация текстур проводится визуально перетаскиванием либо самой текстуры, либо слайдеров, задающих численые параметры rotation, scale, translation (например,CosmoWorlds).

Источники освещения

DirectionalLight (направленный параллельный свет)

Описание:

```
DirectionalLight {
  ambientIntensity 0
  color 1 1 1
  direction 0 0 -1
  intensity 1
  on TRUE
  }
```

Узел DirectionalLight задает освещение параллельными лучами в указанном направлении. По умолчанию это 0 0 -1, что означает направление точно от Вас в экран. В связи с тем, что источник предполагается бесконечно удаленным, не нужно указывать координаты его координаты.

Аналогом этого узла в окружающем мире для нас является Солнце, но в отличие от него освещение от узла DirectionalLight достается не всем предметам в сцене, а только находящимся в том же parent узле, что и сам узел.

Видели когда-нибудь, чтобы на солнечной поляне под безоблачным небом два камня лежали освещенными, а третий нет, а в VRML запросто... Вот нечто подобное.

Просмотр.

Учтите, что размещение узла вне всех grouping узлов равносильно участию его во всех узлах. Легче показать на примере. Вот те же три шарика. Добавлена только одну строчка с еще одним DirectionalLight узлом вне всех Transform узлов. Обратите внимания, что при таком способе освещения нет теней, средняя и правая сферы вовсе не заслонены левой.

Просмотр.

Параметр intensity задает яркость освещения (от 0 до 1), а ambientIntensity (от 0 до 1) задает насколько велик вклад данного

источника в общее освещение сцены за счет отражения и рассеяния от объектов. Параметр `color` задает RGB окраску света.

`PointLight (точечный источник)`

Описание:

```
PointLight {
  ambientIntensity 0
  attenuation 1 0 0
  color 1 1 1
  intensity 1
  location 0 0 0
  on TRUE
  radius 100
  }
```

Узел `PointLight` служит для размещения в сцене точечного источника света, который излучает по всем направлениям (что-то вроде лампочки или свечи). Соответственно, Вы должны указать координаты источника (параметр `location x y z`). Что такое `ambientIntensity`, `intensity`, `color` смотри выше. Параметр `radius` задает радиус сферы освещения, а `attenuation` задает, как быстро будет падать интенсивность по мере удаления от центра. Три числа, указываемые для `attenuation`, используются в формуле для вычисления интенсивности на расстоянии r от центра:

Вот как будет выглядеть график зависимости `I(r)` для некоторых наборов трех чисел в параметре `attenuation`. Посчитано в пределах принятого по умолчанию радиуса сферы освещения 100 м.

Очевидно, что при `attenuation 1 0 0` интенсивность менятся не будет:

При `attenuation 1 1 0`

При `attenuation 1 1 1`

При `attenuation 1 0.1 0`

Учтите, что PointLight является children узлом, поэтому может находиться внутри узла Transform, который повлияет на положение (через параметрtranslation) или на радиус освещенности (через параметр scale).

Просмотр.

Обратите внимание на два момента: во-первых, действительно, поигравшись с параметрами attenuation, можно добиться видимого эффекта (в данном случаеattenuation 0 0.4 0); во-вторых, так же, как и в случае DirectionalLight нет теней: средняя и правая сферы не заслонены левой!

SpotLight (направленный расходящийся свет)

Описание:

```
SpotLight {
ambientIntensity 0
attenuation 1 0 0
beamWidth 1.570796
color 1 1 1
cutoffAngle 0.785398
direction 0 0 -1
intensity 1
location 0 0 0
on TRUE
radius 100
}
```

Глядя на список параметров узла SpotLight, нетрудно догадаться, что он является расширенной комбинацией двух предыдущих способов освещения: общими для всех являются color, ambientIntensity, intensity, on; от DirectionalLight досталось direction, а от PointLight radius и attenuation. В результате получилось что-то вроде фонарика или прожектора: источник имеет положение и светит в определенном направлении.

Поскольку о приведенных параметрах уже говорилось в соответствующих разделах, то здесь я остановлюсь только на незнакомых beamWidth и cutoffAngle. Помимо затухания света при удалении ОТ ИСТОЧНИКА, которое регулируется параметром attenuation, можно также задать размывание по краям светового пятна. Для этого придуманы два конуса с углами раствора beamWidth и cutoffAngle.

Во внутреннем конусе (с углом `beamWidth`) интенсивность в направлении перпендикулярно лучу ПОСТОЯННА, равная параметру `intensity`. Снаружи внешнего конуса (с углом `cutoffAngle`) интенсивность равна 0, а в зазоре между ними интенсивность спадает линейно. В общем виде множитель к исходной интенсивности, указанной в параметре `intensity`, выражается так:

Или в графическом виде (для `beamWidth=45, cutoffAngle=90` градусов):

Обратите внимание, что по умолчанию выставлено `beamWidth > cutoffAngle`, что дает пятно с неразмытыми краями! И не забывайте, что при размещении`SpotLight` внутри `Transform`,
параметры `scale` и `translation` последнего повлияют на все параметры самого `SpotLight`.

Навигация и информация о мире в целом

Background (панорама)

Описание:

```
Background {
groundAngle [ ]
groundColor [ ]
backUrl [ ]
bottomUrl [ ]
frontUrl [ ]
leftUrl [ ]
rightUrl [ ]
topUrl [ ]
skyAngle [ ]
skyColor 0 0 0
}
```

Узел `Background` предназначен для создания в Вашей сцене панорамы, т.е. изображения или цвета на бесконечном удалении. В Вашем распоряжении только два варианта: либо Вы раскрашиваете фон, либо используете в качестве фона картинки. Надо сказать, что оба варианта реализованы очень убого. В первом случае Вы находитесь в сфере бесконечного радиуса и пользуетесь параметрами `groundAngle`, `groundColor`, `skyAngle`, `skyColor`; во втором случае Вы находитесь в кубе бесконечного размера и пользуетесь параметрами `backUrl`, `bottomUrl`, `frontUrl`, `leftUrl`, `rightUrl`, `topUrl`.

1. Раскрашивание фона. В этом случае Вы приписываете цвет угловым интервалам на сфере бесконечного радиуса, и этим цветом заливаются концентрические сферические кольца.
Примерно, как на вот этой картинке.

Или то же самое можете посмотреть в VRML .

У Вас есть две пары
параметров: `skyAngle/skyColor` и `groudAngle/groundColor`. Первая пара - это как бы "небо" и раскрашивается сверху вниз (т.е.`skyAngle=0` - вертикально над Вами), а вторая пара - это как бы "земля" и раскрашивается снизу вверх (т.е. `groundAngle=0` - вертикально под Вами). Смотри рис.

Как видно из рисунка, `skyAngle` изменяется от 0 до Пи, а `groundAngle` - от 0 до Пи/2, и при этом интервалы их изменения перекрываются. Зачем вообще введена пара `groundAngle/groundColor` непонятно, поскольку можно обойтись одной только парой `skyAngle/skyColor`. Возможно, это сделано, чтобы легче было организовать четкую линию горизонта между "небом" и "землей", хотя, горизонт можно сделать и раскрашивая одно только "небо"! Количество значений параметра`Color` должно быть на 1 больше числа значений параметра `Angle`. Если Вы указали соседние кольца разных цветов, то на их границе цвет интерполируется для плавного перехода.

Вот наглядный пример.

```
#VRML V2.0 utf8

Background {
skyColor [0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1,0 0 0,1 1 1]
skyAngle
[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,
2.0,2.1, 2.2,2.3,2.4, 2.5,2.57]
}
```

Просмотр.

2.Использование в качестве фона картинок. Оглядите комнату в которой находитесь: четыре стены, потолок и пол. Точно так же устроена модель фона для размещения картинок - это кубик, у которого есть "потолок" (`topUrl` - указываете адрес картинки для "потолка"), "пол" (`bottomUrl`) и четыре стены - (`leftUrl` - слева, `frontUrl` - перед Вами, `rightUrl` - справа, `backUrl` - позади Вас.).

`NavigationInfo (характеристики аватара)`

Описание:

```
NavigationInfo {
avatarSize [0.25 1.6 0.75]
headlight TRUE
speed 1.0
type ["WALK","ANY"]
visibilityLimit 0.0
}
```

Аватар это представление Вас (или любого другого, просматривающего сцену) в виртуальном мире. (Но не путайте с аватарами в многопользовательских мирах, где на Вас могут посмотреть СО СТОРОНЫ, ИЗНУТРИ виртуального мира, там "аватар" означает модель человека или объект, который Вас обозначает и которым Вы управляете).

Так вернемся к Вашему представлению при просмотре Вами сцены.

Раздел `avatarSize` задает размеры аватара.

Размер по горизонтали (на картинке - c, по умолчанию 0.25) влияет на столкновения Вас с другими объектами, по вертикали (на картинке h, по

умолчанию 1.6) определяет, насколько свысока Вы смотрите на сцену (если только в явном виде не указано положение Viewpoint - см ниже), а третий параметр (на картинке - s, по умолчанию 0.75) определяет насколько высокие объекты Вы можете "перешагнуть" сверху, не уткнувшись в них.

Подводя итог с учетом того, что размеры указаны в метрах, можно сказать, что по умолчанию у Вас странная комплекция: цилиндрическое тело диаметром 25 сантиметров, Ваши глаза находятся на уровне 1 метр 60 сантиметров, а ногу Вы можете оторвать от пола на 75 сантиметров.

Еще одна величина, приведенная на картинке, - v - соответствует параметру `visibilityLimit` в описании узла и определяет, как далеко Вы видите. Рендеринг за пределами `visibilityLimit` НЕ ПРОВОДИТСЯ. Выставленный по умолчанию `visibilityLimit=0` соответствует бесконечному пределу.

Параметр `speed` задает скорость перемещения по миру, но обычно броузеры содержат свои настройки на этот счет, которые игнорируют содержимое файла. Скорось дается в м/с и единственная интересная вещь в этом параметре - это `speed=0`, когда Вы сможете только крутиться на месте и никуда не уйдете.

Оставшийся параметр `type` определяет, какими кнопками навигации Вы сможете пользоваться при просмотре. Возможные значения параметра type: `"ANY", "WALK", "EXAMINE", "FLY", "NONE"`. Собственно, способа исследования сцены всего 3: "`WALK`" - "ходьба", "`FLY`" - "полет" (отличается от "ходьбы" отсутствием гравитации, т.е. траектория движения аватара не повторяет рельеф, НАД которым он движется, "`EXAMINE`" - "изучение" (Вы не движетесь, а движением мышки вращается сцена, этот способ удобен, для осмотра одного отдельного объекта). Очень важное слово "`ANY`" задает, можно ли пользоваться для навигации кнопками броузера. Если в списке параметра type есть "`ANY`", то в любой момент Вы можете просто переключиться с одного способа навигации на другой кнопкой броузера, если в списке нет "`ANY`", тогда остаются только те способы, которые перечислены в параметре `type`. Так, можно запретить "ходить" и "летать" в сцене, а оставить зрителю только возможность покрутить объект. Так должно быть, но обычно кнопки броузера все-таки продолжают работать. Единственно, что можно извлечь из этого параметра, - какой из способов навигации будет выставлен после загрузки сцены.

Но что точно работает, так это значение "`NONE`", когда выключается целиком панель управления броузера. Это применяется, когда у Вас в сцене предусмотрена собственная система навигации: `anchor`'ы, `viewpoint`'ы и т.д.

`Viewpoint (точки обзора)`

Описание:

```
Viewpoint {
fieldOfView 0.785398
jump TRUE
orientation 0 0 1 0
position 0 0 10
description " "
}
```

Узел `Viewpoint` создает в сцене "точки обзора", или, как это называется в ISB, "камеры".

Прежде всего, учтите, что параметры по умолчанию работают при наличии в файле самого узла. То есть пустой узел `Viewpoint {}` - это вовсе не одно и то же, что отсутствие узла. В последнем случае это еще большой вопрос, с какого места броузер начнет показывать Вашу сцену. Поэтому ВСЕГДА прописывайте в файле хотя бы один `Viewpoint`!

Параметр `fieldOfView` (по умолчанию 45 градусов) определяет угол обзора из данной точки, или, другими словами, задает тип объектива. Вот сравните вид сцены при разных `fieldOfView`. Для переключения между Viewpoint'ами в броузере - поищите стрелочки влево/вправо.

```
#VRML V2.0 utf8

Shape {geometry Box {}}
Viewpoint {
jump FALSE
position 0 0 9
orientation -1 0 0 0
fieldOfView 0.78
description "fieldOfView=45"
}
Viewpoint {
jump FALSE
position 0 0 -9
orientation 0 1 0 3.14
fieldOfView 1.57
description "fieldOfView=90"
}
```

Просмотр.

Параметр `jump` определяет, будет ли перемещение между точками обзора дискретным или непрерывным.

Параметры `position` и `orientation` задают положение и ориентацию "камеры" в пространстве.

В параметре `description` можно присвоить "камерам" имена и/или краткое описание, которое отображается в списки `Viewpoint`'ов в броузере.

`WorldInfo (информация о мире: название, автор и т.д.)`

Описание:

```
Worldinfo {
info [ ]
title " "
}
```

Чисто описательный узел, никак не влияет на отображение или поведение сцены. Если Вы претендуете на копирайты и др., то пишите все в разделе `info`. Содержимое раздела `title` должно отображаться броузерами так же, как содержимое тегов `< title >< /title >` в html файлах

Анимация. Сенсоры, маршруты, интерполяторы

Общие замечания

Под анимацией понимаются не только визуальные проявления (изменение у объектов координат, размера, цвета и т.д.), но и любые другие виды динамического изменения сцены (например, включение звука). И первое, что Вам нужно запомнить - слово "**event**", или "событие". Это сообщение о том, что произошло некоторое событие. При этом у каждого **event** есть свой "**timestamp**", т.е. пометка времени, когда произошло событие. Эта пометка служит, во-первых, чтобы события обрабатывались в хронологическом порядке, а во-вторых пометки можно обрабатывать скриптами. Как уже говорилось во Введении, у узлов могут быть параметры`EventIn` (принимают сообщения о событиях), `EventOut` (посылают сообщения о событиях) и `ExposedField` (делает и то, и другое, и к тому же имеет некоторое значение).

Если абстрактно задаться вопросом "Что необходимо для того, чтобы создать анимацию?", то неизбежно возникает несколько стадий решения, а именно:

- как активировать анимацию
- как указать, какой именно объект сцены должен изменяться
- как должен изменяться объект

Ответами на каждый из трех вопросов в VRML занимаются соответствующие средства.

Активацией событий занимаются узлы-сенсоры (генерируют **EventOut**), указанием на конкретный объект занимаются "**ROUTE**"ы, или "маршруты" (транспортируют**EventOut** к объекту), изменением объектов занимаются "**Interpolator**"ы, или "интерполяторы" (обрабатывают **EventIn** и отправляют через очередной **ROUTE** объекту новые параметры).

Разберем по порядку всех участников.

Сенсоры

Основное назначение сенсоров - сгенерировать **EventOut** после срабатывания. Срабатывание сенсора может быть вызвано разными причинами: наступление определенного времени, клик мышкой, наведение курсора, приближение к объекту, столкновение с объектом и т.д.)

К классу сенсоров относятся следующие узлы: **Anchor, Collision, CylinderSensor, PlaneSensor, ProximitySensor, SphereSensor, TimeSensor, TouchSensor, VisibilitySensor**

Anchor

Описание:

```
Anchor {
children [ ]
description " "
parameter [ ]
url [ ]
bboxCenter 0 0 0
bboxSize -1 -1 -1
}
```

Несмотря на отсутствие в явном виде раздела **EventOut**, узел **Anchor** относится к сенсорам, поскольку **EventOut** все-таки генерируется, только маршрут для сообщения строго определен и не может быть изменен. Результатом является обращение к документу, указанному в разделе url. Это может быть не только VRML сцена, но и (гипер)текстовый документ и т.д.

Пощелкайте мышкой по объектам.

Просмотр.

Кроме того, работают (по крайней мере ДОЛЖНЫ работать) **anchor**'ы в виде url [имя_файла#viewpoint для VRML файлов и имя_файла#name для HTML файлов. Напомню, что **Anchor** является **grouping** узлом, поэтому ссылкой становятся все объекты в пределах раздела **children**.

Параметр **description** - чисто описательный. При наведении курсора на объект с **anchor**'ом в статус-лайне обычно отображается содержимое **description**.

В дополнение к параметру url есть параметр **parameter**, это механизм, аналогичный HTML, для отображения файла в определенном фрейме. Если фрейм не указан, создается новое окно.

Параметр **bbox** (или **bounding box**) - предназначенна для ускорения рендеринга. При этом **children** узлы ищутся уже не по всему пространству а в пределах этого самого ящика (**bounding box**) с размерами **bboxSize** и центром в **bboxCenter**.

Необходимо, чтобы **children** узлы ДЕЙСТВИТЕЛЬНО попадали в пределы **bounding box**, иначе результат не определен. Поэтому проще никогда не связываться с**bbox** и оставить ее по умолчанию бесконечно большой (**bboxSize -1 -1 -1**)

Collision

Описание:

```
Collision {
children [ ]
collide TRUE
bboxCenter 0 0 0
bboxSize -1 -1 -1
proxy NULL
eventOut collideTime
}
```

Узел **Collision** выполняет две задачи: во-первых, регистрирует факт столкновения (и время этого столкновения) аватара (т.е. Вас) с объектом, указанным в разделе**children**, а во-вторых регулирует, пройдете ли Вы СКВОЗЬ объект (**collide TRUE**) или нет (**collide FALSE**). В момент столкновения генерируется **eventOutcollideTime,** что позволяет создать последовательность последующих событий. (Учтите, что невозможно организовать столкновения с **IndexedLineSet, PointSet, Text**.)

Если в Вашем VRML файле нет ни одного узла **Collision**, то "проницаемость" объектов регулирует броузер. Наверняка Вы видели в настройках броузера что-нибудь вроде **collision ON/OFF, collider Auto/Always/Never** и т.д.

Про **bboxCenter** и **bboxSize** читайте в разделе **Anchor**.

Параметр **proxy** - это очень полезная вещь для ускорения рендеринга. Если в разделе у Вас не **NULL**, а какой-нибудь ОБЪЕКТ, то он не изображается в сцене (невидим), но столкновение на самом деле будет регистрироваться с ним, т.е. объектом, указанным в разделе **proxy**, а не в разделе **children**.

К примеру, Вам необходимо сделать зарегистрировать столкновение с объектом сложной формы, что-нибудь вроде:

Просмотр.

Зачем напрягать броузер расчетом положения аватара относительно всех этих граней? Просто прописываете в разделе **proxy** СФЕРУ чуть большего радиуса, чем "иглы" звезды и все. Не забудьте выставить у броузера **collide ON** !

ВЫВОД: как и для всех узлов, значения которых может проигнорировать броузер (**NavigationInfo: avatarSize и headlight ON/OFF**), ценность узла**Collision** резко снижена! Вместо пользуйтесь, например, **ProximitySensor.**

ProximitySensor

Описание:

```
ProximitySensor {
center 0 0 0
size 0 0 0
enabled TRUE
eventOut isActive
eventOut position_changed
eventOut orientation_changed
eventOut enterTime
eventOut exitTime
}
```

Вот этот сенсор я очень люблю, работает безотказно, просто и со многими возможностями. Представьте себе невидимый параллелепипед, размещаемый где угодно в пространстве. При пересечении его границ сенсором генерируются сообщения:

- о самом факте пересечения границ (**eventOut isActive**)
- о времени входа (**eventOut enterTime**), если Вы попали внутрь параллелепипеда или времени выхода (eventOut exitTime), если Вы выбрались из него

А когда Вы уже находитесь внутри параллелепипеда генерируются следующие сообщения:

- если Вы движетесь (**eventOut position_changed**)
- если Вы поворачиваетесь (**eventOut orientation_changed**)

Сколько бы у Вас в сцене ни было **ProximitySensor**'ов, все они работают независимо друг от друга. При этом они могут пересекаться в пространстве, быть вложенными один в другой или даже полностью совпадать (при этом при пересечении их общей границы Вы сработают оба).

Пример.

TimeSensor

Описание:

```
TimeSensor {
cycleInterval 1
enabled TRUE
loop FALSE
```

```
startTime 0
stopTime 0
eventOut cycleTime
eventOut fraction_changed
eventOut isActive
eventOut time
}
```

Исключительно важный узел. В 90 процентах случаев Вы не обойдетесь без него при организации анимации, поскольку именно здесь можно регулировать СКОРОСТЬ протекания процессов.

Обратите внимание, что по умолчанию выставлено **enabled TRUE**. Это означает, что если Вы не указали в явном виде **enabled FALSE** (а также не меняли поле**startTime 0**), то после загрузки сцены **TimeSensor** сразу начинает генерировать разнообразные **eventOut'ы** ! Т.е. по умолчанию **TimeSensor** НЕ ПРИХОДИТСЯ АКТИВИРОВАТЬ, и этим он принципиально отличается от остальных узлов-сенсоров.

Не забудьте, что кроме поля **enabled**, есть поля s**tartTime** и **stopTime**, которые имеют больший приоритет, чем **enabled**. Но об этом - чуть позднее.

А пока разберемся с остальными полями.

Цикл работы **TimeSensor**'а может быть либо один (**loop FALSE**) либо при **loop TRUE** циклов будет много, (бесконечно, если **stopTime** меньше, чем**startTime**, иначе пока не наступит **stopTime**). Длительность цикла задается полем **cycleInterval**.

Что же происходит хронологически?

Если **startTime=0**, то как только **TimeSensor** активирован (при **enabled TRUE** - сразу после загрузки, а при **enabled FALSE** - после получения сообщения**TimeSensor.enabled**), генерируются следующие **eventOut**'ы:

- **eventOut isActive** Посылается значение TRUE до тех пор, пока TimeSensor не будет деактивирован
- **eventOut cycleTime** Посылается в момент наступления startTime и в начале каждого последующего цикла при loop TRUE.
- **eventOut time** Посылает абсолютное значение времени, прошедшего с начала цикла.
- **eventOut fraction_changed** В отличие от eventOut time посылает ОТНОСИТЕЛЬНОЕ значение степени завершенности цикла: в начале цикла fraction_changed=0, в конце цикла fraction_changed=1. Это один из наиболее часто используемых eventOut'ов. При loop TRUE почти всегда используется волшебная связка
  .fraction_changed - .set_fraction - .value_changed,

где `.fraction_changed` - это часть от TimeSensor, .set_fraction - часть от интерполятора, .value_changed - часть от узла, у которого изменяются свойства.

Теперь поговорим о **`startTime`** и **`stopTime`**.

Как известно, в сетевых технологиях счет времени ведется с 00:00:00 1 января 1970 года, и значения полей **`startTime`** и **`stopTime`** в том числе. Но поскольку эти значения надо указать в секундах (т.е. сколько прошло с 00:00:00 1 января 1970 года), то чтобы пользоваться этими замечательными средствами управления**`TimeSensor`**'а не обойтись без маленького скрипта.

Просмотр. Пришлось перевести миллисекунды в секунды (поделить на 1000),

ВЫВОД: никогда не связывайтесь с вычислением в явном виде абсолютных значений при измерении времени (**`time, startTime, stopTime`**) ! А если Вам нужно запустить анимацию в определенное время, пользуйтесь средствами, работающими с относительными значениями (**`cycleInterval`**).

И тут есть два противоположных случая: первый - если после запуска в определенное время далее анимация циклично повторяется (loop TRUE), второй - если после запуска в определенное время анимация должна совершиться ОДИН РАЗ.

1. Первая задача решается очень легко. Посмотрите пример и поймете сами: Просмотр. Шарик начнет двигаться через 4 секунды после открытия файла и будет двигаться бесконечно!
2. Вторая задача несколько сложнее, поскольку нам необходимо получить сигнал **`set_startTime`**. Кто разберется в примере, надеюсь оценит идею.
   Просмотр. Шарик начинает двигаться через 4 секунды и движется 5 секунд.

`TouchSensor`

Описание:

```
TouchSensor {
enabled TRUE
eventOut hitNormal_changed
eventOut hitPoint_changed
eventOut hitTexCoord_changed
eventOut isActive
eventOut isOver
eventOut touchTime
}
```

Хороший сенсор, обычно используемый для большей интерактивности: чтобы что-нибудь открылось/закрылось/заработало и т.д. приходится навести курсор или щелкнуть на чем-нибудь мышкой.

Поле у узла всего одно **enabled TRUE/FALSE**, а сам сенсор привязывается ко всем объектам, объединенным с **TouchSensor**'ом в одну **parent** группу.

Работа сенсора начинается с момента наведения курсора мыши (или другого манипулятора) на объект, к которому привязан **TouchSensor**. При этом начинает генерироваться **eventOut isOver TRUE**. А кроме того при перемещении курсора над поверхностью объекта генерируются **eventOut hitNormal_changed**(отслеживается положение вектора нормали к поверхности объекта), **eventOut hitPoint_changed** (отслеживаются координаты точки на поверхности объекта, над которой находится курсор), **eventOut hitTexCoord_changed** (отслеживается положение точки на поверхности объекта, над которой находится курсор в координатах **texture map**).

Оставшийся два **eventOut**'а **eventOut isActive** и **eventOut touchTime** генерируются при участии кнопки мыши: isActive TRUE генерируется только до тех пор пока Вы держите кнопку мыши НАЖАТОЙ (а после отпускания **isActive FALSE**), а **eventOut touchTime** наоборот генерируется КАК ТОЛЬКО Вы ОТПУСКАЕТЕ кнопку мыши.

Наиболее употребительными мне представляются **isOver, isActive и touchTime**

Просмотр.

VisibilitySensor

Описание:

```
VisibilitySensor {
center 0 0 0
enabled TRUE
size 0 0 0
eventOut enterTime
eventOut exitTime
eventOut isActive
}
```

Этот сенсор определяет, находится ли в поле зрения область пространства внутри параллелепипеда с центром в **center** и размером **size**. Как только область становится видимой, генерируется **eventOut enterTime** и **isActive** становится **TRUE**. Как только Вы "отвернулись" и не

видите содержимое параллелепипеда, генерируется **eventOut exitTime** и **isActive** становится **FALSE**.

Сенсор может быть полезен для оптимизации Вашей сцены. К примеру, можно прекращать часть анимации, который в данный момент не видна. Останавливая таким способом обработку анимации, можно увеличить число fps.

Dragging Sensors

Название этого подраздела обусловлено тем, что для активирования описываемых здесь сенсоров можно не только кликнуть мышью (нажать кнопку и ОТПУСТИТЬ), а нажать и НЕ ОТПУСКАЯ перемещать. Наверняка Вам знакомо понятие "drag and drop", вот эти сенсоры из такой серии.

PlaneSensor

Описание:

```
PlaneSensor {
autoOffset TRUE
enabled TRUE
maxPosition -1 -1
minPosition 0 0
offset 0 0 0
eventOut isActive
eventOut trackPoint_changed
eventOut translation_changed
}
```

Этот сенсор отслеживает перемещения курсора в плоскости с Z=0 локальной системы координат (по умолчанию в плоскости экрана).

Как только над объектом, к которому привязан PlaneSensor, происходит нажатие кнопки мыши, генерируется eventOut isActive. После этого при перемещении курсора при нажатой кнопке мыши отслеживаются текущие значения координат курсора (eventOut trackPoint_changed) и вектора перемещения (eventOut translation_changed).

Поле autoOffset определяет, будут ли суммироваться смещения (autoOffset TRUE) или каждое смещение будет отсчитываться от исходного положения объекта (autoOffset FALSE).

Просмотр. Для обоих шариков проделайте следующее: сдвиньте, отпустите кнопку мыши и попробуйте сдвинуть снова. Тот, что слева начнет двигаться с того места, на котором Вы его бросили (autoOffset TRUE), а тот что слева начнет двигаться с того места, в котором он был изначально (autoOffset FALSE)

Поле **enabled TRUE/FALSE** разрешает/запрещает работу сенсора.

Пара полей **maxPosition** и **minPosition** позволяют организовать двумерное движение в ограниченном регионе (как это было в предыдущем примере) или даже свести движение к одномерному. Просмотр.
Если **maxPosition** меньше, чем minPosition, то движение не ограничивается.

Поле **offset** определяет первоначальное смещение, относительно исходного положения объекта, а значит и точку, с которой будет каждый раз начинать движение объект при **autoOffset FALSE**.

CylinderSensor

Описание:

```
CylinderSensor {
autoOffset TRUE
diskAngle 0.262
enabled TRUE
maxAngle -1
minAngle 0
offset 0 0 0
eventOut isActive
eventOut rotation_changed
eventOut trackPoint_changed
}
```

Этот сенсор отслеживает движение курсора мыши в цилиндрической системе координат невидимого цилиндра с осью вращения параллельной локальной оси Y.

Большинство полей (autoOffset, enabled, offset) и eventOut'ов этого сенсора (isActive, trackPoint_changed) такие же, как и у PlaneSensor, где Вы и можете про них прочитать.

Пара полей maxAngle и minAngle аналогична maxPosition и minPosition PlaneSensor. Если maxAngle меньше, чем minAngle, то вращение не ограничивается.

eventOut rotation_changed аналогично eventOut translation_changed PlaneSensor.

Единственное действительно отличительное поле - это diskAngle.

В спецификации предусмотрено два способа описание движения курсора через CylinderSensor. Представьте себе отдельно взятое велосипедное колесо. Если Вы всунете палец между спиц, то сможете вращать колесо бесконечно НЕ ОТРЫВАЯ руку, которая будет описывать конус вращения. А если Вы схватитесь за обод, то для поддержания бесконечного вращения Вам придется постоянно ПЕРЕХВАТЫВАТЬ руку.

Аналогично, в VRML можно хвататься через CylinderSensor за ТОРЕЦ цилиндра или за его БОКОВУЮ СТОРОНУ.

Для регулирования, когда используется какой способ и введен **diskAngle**. Если угол между **bearing** вектором и осью цилиндра МЕНЬШЕ **diskAngle**, то Вы сможете, зацепив мышью цилиндр, вращать его бесконечно (при этом курсор будет описывать на экране круги). Если угол между вектором и осью цилиндра БОЛЬШЕ**diskAngle**, то придется, провернув немного цилиндр, отпускать кнопку мыши и перетаскивать курсор (при этом курсор на экране будет двигаться дискретно-прямолинейно). {**bearing** вектор - это вектор, проходящий сквозь курсор на экране.}

И кстати, не забывайте, что **CylinderSensor** можно привязывать к объектам любой формы, а не только к телам вращения. Просмотр. Потяните параллелепипед за боковые грани - вращение будет продолжаться, пока вы не дотянете ПРЯМОЛИНЕЙНО курсор до границы экрана. Теперь поверните ее к себе верхней гранью и крутите ее, пока не надоест, перемещая курсор ПО КРУГУ.

SphereSensor

Описание:

```
SphereSensor {
autoOffset TRUE
enabled TRUE
offset 0 0 0
eventOut isActive
eventOut rotation_changed
eventOut trackPoint_changed
}
```

После описания **PlaneSensor** и **CylinderSensor** здесь нечего рассказывать. Разве что пример привести. Просмотр.

Маршруты

Механизм route можно уподобить проводам, по которым передаются сигналы от **eventOut** к **eventIn** узлов или скриптов, можно от одного eventOut рассылать сообщения о событиях нескольким eventIn. И наоборот, к одному eventIn могут быть проложены маршруты от нескольких eventOut'ов. Но последней ситуации желательно избегать, поскольку, если на eventIn ОДНОВРЕМЕННО поступает два и более сообщений, результат не определен.

Просмотр. Щелкните на любом из объектов. Это пример, когда от одного **eventOut** сообщения рассылаются нескольким **eventIn**.

Интерполяторы

Как уже говорилось в начале страницы, интерполяторы выдают объекту численное значение какого-либо его параметра (цвет, положение, размер и т.д.) в данный момент времени в течение **cycleInterval**. За

каждый **cycleInterval** интерполятор пробегает все значения полей **key** и **keyValue**.

Все узлы-интерполяторы (**ColorInterpolator, CoordinateInterpolator, NormalInterpolator, OrientationInterpolator, PositionInterpolato, ScalarInterpolator**) записываются одинаково:

```
eventIn set_fraction
key [набор контрольных точек]
keyValue [набор значений, соотвествующий точкам в поле key]

eventOut value_changed
```

Отличия заключаются только в ТИПЕ значения, отсылаемого через **eventOut value_changed**.

Если число значений в поле **keyValue** не соответствует количеству контрольных точек в поле **key**, результат не определен.

Важно помнить, что значения **keyValue** в ПРОМЕЖУТОЧНЫХ точках между указанными контрольными точками интерполируется ЛИНЕЙНО ! Т.е. если Вы, скажем, хотите организовать поступательное движение объекта по дуге, нужно быть внимательным, вводя большое количество точек в поле **key** (и соответственно в поле**keyValue**), поскольку движение будет аппроксимироваться ломаной.

ColorInterpolator

Описание:

```
ColorInterpolator { eventIn set_fraction
key [ ]
keyValue [ ]
eventOut value_changed
}
```

Этот интерполятор позволяет отсылать объектам (через узел **Material**) RGB значения цвета.

Просмотр.

ColorInterpolator

Описание:

```
CoordinateInterpolator { eventIn set_fraction
key [ ]
keyValue [ ]
eventOut value_changed
}
```

Этот интерполятор позволяет отсылать наборы координат узлу Coordinate, который встречается в узлах **IndexedFaceSet, IndexedLineSet и PointSet**. Это удобно использовать, когда для деформации объекта не подходит анизотропное масштабирование.

Просмотр.

NormalInterpolator

Описание:

```
NormalInterpolator { eventIn set_fraction
key [ ]
keyValue [ ]
eventOut value_changed
}
```

Само понятие нормали (normal) возникает в VRML в уравнениях расчета освещенности каждой точки поверхности объекта (в частности в **diffuse** и **specular color**). А узел **NormalInterpolator** дает возможность динамически регулировать распределение освещенности по объекту за счет изменения направления векторов нормалей к граням объекта.

Просмотр.

OrientationInterpolator

Описание:

```
OrientationInterpolator { eventIn set_fraction
key [ ]
keyValue [ ]
```

```
eventOut value_changed
}
```

Нетрудно догадаться, что **`OrientationInterpolator`** позволяет организовать вращение. Только не забывайте, что значения ПРОМЕЖУТОЧНЫЕ между**`keyValues`** вычисляются из ЛИНЕЙНОЙ интерполяции.

Просмотр.