

Практическое занятие № 1

Тема: Язык VRML. Построение примитивов и управление положением объектов в VRML

Цель: Ознакомить со средой VRML. Изучить приемы построения примитивов

Задание:

- 1 Ознакомиться с теоретическим материалом.
- 2 Создайте документ с изображением кубика размерами 2x2x2.
- 3 Создайте документ с изображением сферы радиусом 3.
- 4 Создайте документ с изображением конуса высотой 4, радиусом 3 без дна.
- 5 Создайте документ с изображением цилиндра высотой 4, радиусом 3, с верхней, но без нижней крышки.
- 6 Создайте документ с изображением красного квадрата размерами 3x6x1.
- 7 Создайте документ с изображением синего цилиндра высотой 4, радиусом 3.
- 8 Создайте документ с изображением зеленого конуса высотой 3.5, радиусом 2.
- 9 Создайте документ с изображением полупрозрачной сферы радиусом 4.
- 10 Создайте документ с изображением сферы и конуса.
- 11 Создайте изображение эллипсоида.
- 12 Создайте документ с изображением конуса.

Необходимые приборы: ПК, текстовый редактор Блокнот, браузер

Методические рекомендации к практической работе:

Методические рекомендации к выполнению задания 1

Vrml - платформно-независимый объектно-ориентированный расширяемый язык программирования, который в самых общих принципах напоминает хорошо известный проектировщикам Web-страниц язык HTML. VRML-документ запрашивается с Web-сервера и поступает пользователю в виде исходного текста, точно так же, как HTML-документ. Поскольку большинство браузеров не имеет встроенных средств поддержки VRML, для просмотра RML-документов необходимо подключить вспомогательную программу- VRML-браузер. Один и тот же VRML-документ может выглядеть по-разному в разных браузерах.

Язык VRML (Virtual Realty Modelling Language) предназначен для описания трехмерных изображений и оперирует объектами, описывающими геометрические фигуры и их расположение в пространстве.

Vrml-файл представляет собой обычный текстовый файл, интерпретируемый браузером. Поскольку большинство браузеров не имеет

встроенных средств поддержки vrml, для просмотра Vrml-документов необходимо подключить вспомогательную программу - Vrml-браузер, например, Live3D или Cosmo Player.

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. Оси располагаются на экране следующим образом: X - горизонтально в плоскости экрана, Y - вертикально в плоскости экрана, Z - смотрит прямо на Вас из экрана.

Важно запомнить, что VRML код регистрозависим, т.е. имеет значение, прописные буквы или строчные. Так, записи Vox, vox, BOX вовсе не одно и то же, и только первая из них правильная.

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.

Каждый файл с кодом VRML97 должен начинаться строкой

#VRML V2.0 utf8

Запись utf8 означает тип кодировки. Вообще могут быть utf7, utf8, utf16, что означает соответственно семи-, восьми- и шестнадцатититную кодировку.

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 VRML

В VRML определены четыре базовые фигуры: куб (прямоугольный параллелепипед), сфера, цилиндр и конус.

Эти фигуры называются примитивами (primitives). Набор примитивов невелик, однако комбинируя их, можно строить достаточно сложные трехмерные изображения.

Рассмотрим подробно каждый из примитивов.

1. Box (параллелепипед)

Описание прямоугольного параллелепипеда:

Box {size 2 2 2}

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

Узел Shape указывает браузеру, что идет описание формы объекта.

2. Sphere (сфера)

Описание сфер выглядит так

Sphere {radius 1}

По умолчанию центр сферы помещается в точку 0,0,0. Таким образом, если вы будете указывать только узлы Sphere, то получите все сферы с центрами в 0,0,0, как матрешки.

Из-за того, что не указан радиус сферы, она по умолчанию получилась радиусом 1 метр.

3. Cone (конус)

Описание конуса:

```
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 дно конуса. При side TRUE и bottom FALSE, получится конус без доньшка; при side FALSE и bottom TRUE, получите только доньшко; при side FALSE и bottom FALSE получите конус, невидимый и неосязаемый при движении сквозь него, в общем это будет отсутствие конуса.

4. Cylinder (цилиндр)

Описание цилиндра:

```
Cylinder {bottomTRUE
height 2
radius 1
sideTRUE
topTRUE }
```

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.

Цвет фигуры

```
Appearance {
exposedField SFNode material NULL
exposedField SFNode texture NULL
exposedField SFNode textureTransform NULL }
```

Все, что нас интересует - это поле material, куда помещается объект класса Material, задающий световые характеристики материала. Цвет фигуры, определяется с помощью объекта Material.

```
Material {
ambientColor 0.2 0.2 0.2
diffuseColor 0.8 0.8 0.8
specularColor 0 0 0
emissiveColor 0 0 0
transparency 0 }
```

- задает свойства материала. Все поля принимают значения от 0.0 до 1.0.

Параметры ambientColor, diffuseColor, specularColor и emissiveColor

управляют цветами и указываются в палитре RGB (красный, зеленый и голубой), причем первая цифра определяет интенсивность красного цвета, вторая - зеленого, а третья - синего.

diffuseColor (диффузный цвет) - определяет нормальный отраженный свет. Чем прямее попадание луча от источника света на поверхность, тем больше (ярче) отраженный свет.

emissiveColor - предназначен для моделирования самосветящихся объектов.

transparency - прозрачность объекта. 1.0 - полностью прозрачен, 0.0 - абсолютно непрозрачен.

ambientIntensity - другая форма отражения света. Параметр определяет силу отражения так называемого окружающего (**ambient**) света. **Ambient light** распространяется по всем направлениям и сила отражения не меняется с изменением угла падения луча.

specularColor (зеркальный цвет) и **shininess** (яркость) задают зеркальные свойства объекта. Когда угол между источником света и поверхностью близок к углу между поверхностью и наблюдателем, **specularColor** добавляется к вычисленному свету (посчитанному с учетом остальных параметров). Малая яркость порождает мягкое свечение, расплывчатые зеркала; высокая - резкое и яркое отражение света.

Методические рекомендации к выполнению задания 2

Создайте документ с изображением кубика размерами 2x2x2:

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

```
Shape {geometry Box {size 2 2 2}}
```

Методические рекомендации к выполнению задания 3

Создайте документ с изображением сферы радиусом 3:

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

```
Sphere {radius 3}
```

Методические рекомендации к выполнению задания 4

Создайте документ с изображением конуса высотой 4, радиусом 3 без дна:

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

```
Cone {bottomRadius 3
```

```
height 4
```

```
side TRUE
```

```
bottom FALSE }
```

Методические рекомендации к выполнению задания 5

Создайте документ с изображением цилиндра высотой 4, радиусом 3, с верхней, но без нижней крышки:

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

```
Cylinder {bottom TRUE
```

```

height 4
radius 3
side TRUE
top FALSE }

```

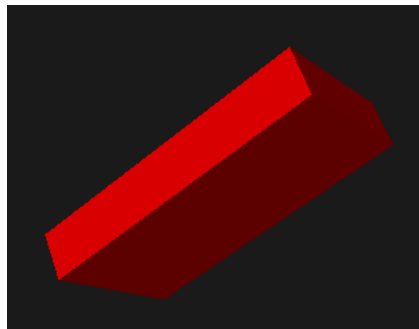
Методические рекомендации к выполнению задания 6

Создайте документ с изображением красного квадрата размерами 3x6x1
#VRML V2.0 utf8

```

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 0 0
    }
  }
  geometry Box {
    size 3 6 1
  }
}

```



Методические рекомендации к выполнению задания 7

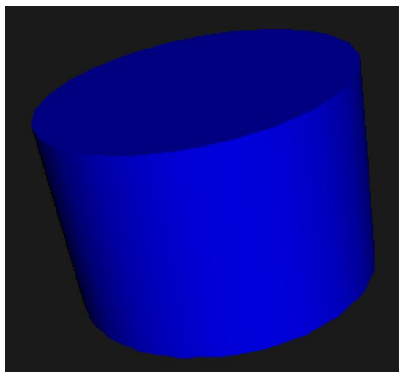
Создайте документ с изображением синего цилиндра высотой 4,
радиусом 3

#VRML V2.0 utf8

```

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0 0 1
    }
  }
  geometry Cylinder {bottom TRUE
height 4
radius 3
side TRUE
top TRUE } }

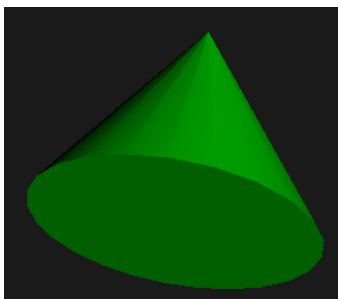
```



Методические рекомендации к выполнению задания 8

Создайте документ с изображением зеленого конуса высотой 3.5, радиусом 2

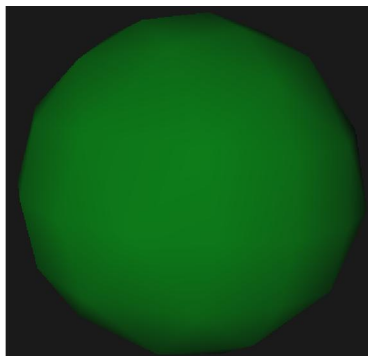
```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0 0 1
    }
  }
  geometry Cone {bottomRadius 2
height 3.5
side TRUE
bottom TRUE } }
```



Методические рекомендации к выполнению задания 9

Создайте документ с изображением полупрозрачной сферы радиусом 4:

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0 0.9 0.1
      transparency 0.5
    }
  }
  geometry Sphere {radius 4} }
```

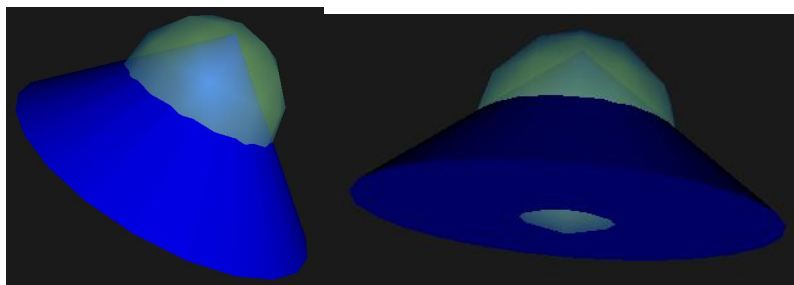


Методические рекомендации к выполнению задания 10

Создайте документ с изображением сферы и конуса:

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

```
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.6 0.7 0.1
      emissiveColor 0 0.2 0.4
      ambientIntensity 0.7
      transparency 0.4
      specularColor 0 0.2 0.6
    }
  }
  geometry Sphere {radius 4} }
  Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0 0 1
      }
    }
    geometry Cone {bottomRadius 8
height 7
side TRUE
bottom TRUE } }
```



Преобразование системы координат

```
Transform {
  addChildren
  removeChildren
  center 0 0 0
```

```

children []
rotation 0 0 1 0
scale 1 1 1
scaleOrientation 0 0 1 0
translation 0 0 0
bboxCenter 0 0 0
bboxSize -1 -1 -1 }

```

Геометрические классы позволяют создавать только правильные фигуры. Часто применяется преобразование координат, т.е. масштабирование (**scale**) координаты x, y, z с разными коэффициентами.

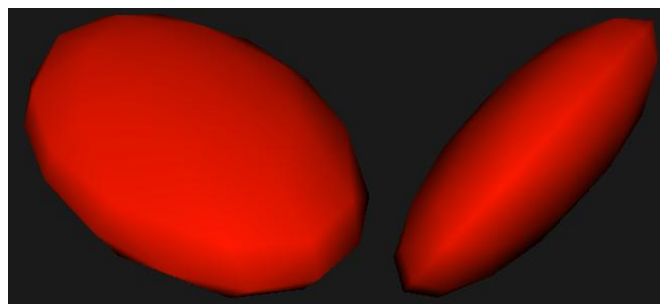
Методические рекомендации к выполнению задания 11

Создайте изображение эллипсоида:

```

#VRMLV2.0 utf8
Transform {
  scale 2 1 3
  children Shape {
    appearance Appearance {
      material Material {
        diffuseColor 1 0.1 0    }}
    geometry Sphere {} } }

```



Поля **translation**, **rotation**, **scale**, **scaleOrientation**, **center** дают возможность осуществить любую геометрическую трансформацию (аффинное преобразование координат):

center задает центральную точку, относительно которой будут производиться другие трансформации (например вращение - **rotation**);

translation задает перенос системы координат (СК);

rotation - поворот СК вокруг заданной оси на заданный угол;

scale - масштабирование;

scaleOrientation - определяет "виртуальный" поворот СК перед выполнением масштабирования, т.е. позволяет проводить масштабирование НЕ вдоль стандартных осей **Ox**, **Oy**, **Oz**.

Настоятельно рекомендуется использовать **Transform** (фактически это установка локальной системы координат) для каждого более-менее законченного геометрического объекта. Если, например, вам понадобится увеличить ваш объект в два раза, то это будет легко сделать - просто напишите

scale 2 2 2 в объекте **Transform**.

Transform может являться родителем другого объекта **Transform**.

Методические рекомендации к выполнению задания 12

Создайте документ с изображением конуса

```
#VRMLV2.0 utf8
```

```
Transform {
```

```
  scale 3 5 5
```

```
  children Shape {
```

```
    appearance Appearance {
```

```
      material Material {
```

```
        diffuseColor 1 1 0}}
```

```
        geometry Cone {bottomRadius 2
```

```
          height 1
```

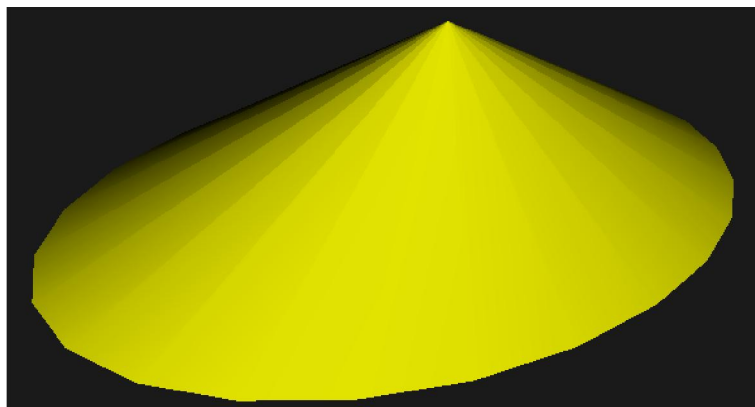
```
          side TRUE
```

```
          bottom TRUE
```

```
        }
```

```
      }
```

```
    }
```



Вопросы для самоконтроля:

- 1 Для чего предназначен язык Vrm1?
- 2 Каждый ли браузер поддерживает технологию Vrm1?
- 3 Что представляет собой Vrm1-файл?
- 4 Какие единицы измерения приняты в Vrm1?
- 5 В какой системе координат берутся координаты?
- 6 Какие примитивы используются в Vrm1?
- 7 Назовите команды создания основных базовых фигур в среде Vrm1:
куб, сфера, цилиндр и конус.

Практическое занятие № 2

Тема: Текст, стиль шрифта. Наложение текстур

Цель: Изучить приемы работы с текстом, его форматированием и наложением текстур

Задание:

- 1 Ознакомиться с теоретическим материалом.
- 2 Создайте документ по образцу.
- 3 Создайте документ, содержащий два куба.
- 4 Создайте документ с изображением стола.
- 5 Самостоятельно создайте документ, содержащий изображение стула.

Необходимые приборы: ПК, текстовый редактор Блокнот, браузер

Методические рекомендации к практической работе:

Методические рекомендации к выполнению задания 1

Text (текст)

Описание:

```
Text {  
  string ["Hallo!","Hallo!!!"]  
  fontStyleNULL  
  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 строка текста.

Раздел **length**: если length больше, чем приведенная в string строка, то либо масштабируется текст, либо увеличивается расстояние между буквами.

Обратите внимание на разбивку текста в поле string на два куска в кавычках ["Hallo!","Hallo!!!"]. Это дает размещение текста в две строки.

В случае записи ["Hallo! Hallo!!!"] - текст разместится в одну строку.

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.

- **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`, `END` соответствуют 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** – задание языка.

- **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. Аналогично по вертикали.

Наложение текстур

Процедура наложения текстур состоит из двух частей, которыми занимаются соответствующие узлы. Узлы `ImageTexture`, `MovieTexture`, `PixelTexture` описывают, 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 в "...".

Самой текстуре присваивается локальная система координат `S` на `T`, где `S` соответствует 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.

Браузерам СТРОГО ПРЕДПИСАНО поддерживать jpeg и png. Но кроме того обычно поддерживаются gif и ряд других распространенных форматов.

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 MPG. Поскольку этот формат позволяет хранить как видео-, так и аудиоинформацию, соответственно узел MovieTexture 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.

Параметр **loop**, как обычно задает, будет ли файл прокручиваться бесконечно.

Параметр **speed** позволяет задавать скорость проигрывания MPG файла. В случае speed < 0 проигрывание идет в обратном порядке. В случае speed = 0 будет статично отображаться 0-й фрейм MPG файла.

Параметры **startTime** и **stopTime** определяют обработку проигрывания во времени MPG файла и обычно используются для запуска/остановки в определенный момент или после определенного события.

Параметры **url**, **repeatS** и **repeatT** занимают тем же, что и в узле

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.

А запись

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

задает текстуру в полупрозрачную клеточку красного, зеленого, синего и черного цвета.

TextureTransform (трансформация текстур)

Описание:

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

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 визуальным перетаскиванием либо самой текстуры (например, ISB), либо слайдеров, задающих численные параметры rotation, scale, translation

Методические рекомендации к выполнению задания 2

Создайте документ, содержащий текст:

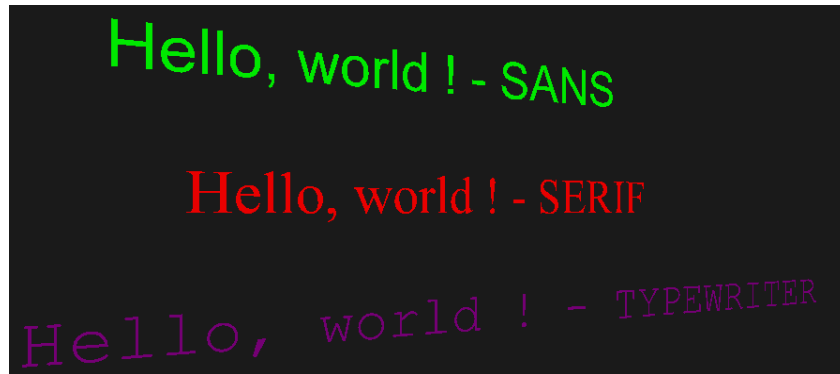
```
#VRMLV2.0 utf8
```

```
Shape {  
  appearance Appearance {  
    material Material {  
      diffuseColor 1 0 0    }}  
  geometry Text {string ["Hello, world ! - SERIF"]  
    fontStyle FontStyle {family "SERIF"}} }  
  Transform {  
    translation -1 2 0  
    children [Shape {  
      appearance Appearance {  
        material Material {  
          diffuseColor 0 1 0    }}  
      geometry Text {string ["Hello, world ! - SANS"]  
        fontStyle FontStyle {family "SANS"}}  
    }]  
  }  
}
```

```

translation -2 -2 0
children [Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.5 0 0.5  }}
  geometry Text {string ["Hello, world ! - TYPEWRITER"]}
  fontStyle FontStyle {family "TYPEWRITER"}}
}]

```



Методические рекомендации к выполнению задания 3

Создайте документ, содержащий два куба

#VRML V2.0 utf8

#Описание куба с именем MyBox

```

Transform {
  translation 0 0 0
  children [
    DEF MyBox Shape {
      appearance Appearance {
        texture ImageTexture {
          url ["image2.jpeg"]
        }
      }
    }
  ]
}

```

```

geometryBox {
  size 2 2 2
}

```

```

] }

```

#Описание второго куба типа MyBox при помощи оператора USE

```

Transform {
  translation 0 3 0
  children [
    USE MyBox
  ]
}

```

Методические рекомендации к выполнению задания 4

Создайте документ, содержащий изображение стола:

#VRML V2.0 utf8

#Описание поверхности

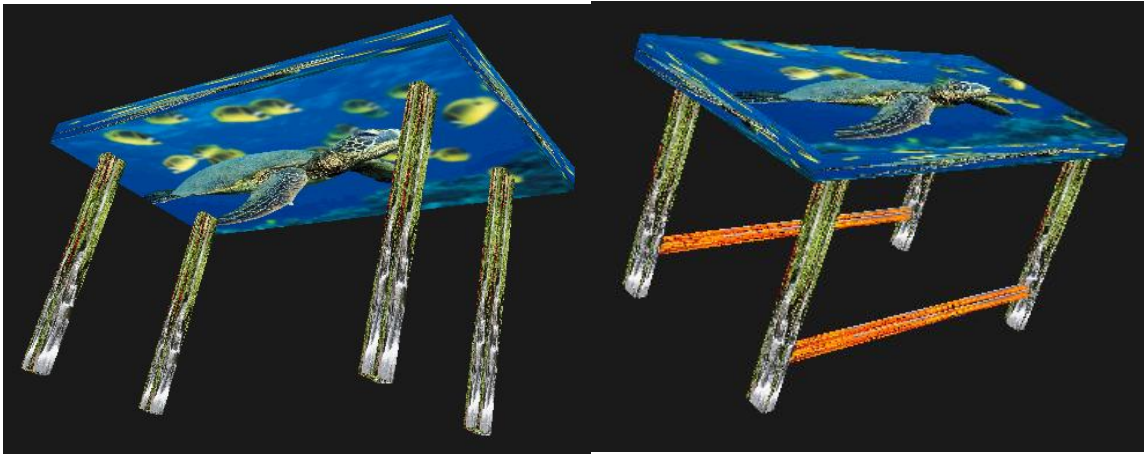
```

Transform {
translation 0 0 0
  children [
    DEF MyBox Shape {
      appearance Appearance {
        texture ImageTexture {
          url ["image1.jpeg"]
        }
      }
    }

    geometry Box {
      size 20 1 10
    }
  ] }
#Описаниевторогокуба
Transform {
translation 8 -5 3
  children [
    DEF MyBox1 Shape {
      appearance Appearance {
        texture ImageTexture {
          url ["image2.jpeg"]
        }
      }
    }

    geometry Box {
      size 0.5 10 1
    }
  ]
}
]}
Transform {
translation 8 -5 -3
  children [
    USE MyBox1
  ] }
  Transform {
translation -8 -5 -3
  children [
    USE MyBox1
  ] }
}
Transform {
translation -8 -5 3
  children [
    USE MyBox1
  ]
}
}
Добавьте самостоятельно нижние перекладины.

```



Вопросы для самоконтроля:

- 1 Назовите команды добавления текста?
- 2 Перечислите основные параметры текста.
- 3 Для чего предназначена команда length?
- 4 Каким образом задается стиль шрифта?
- 5 Опишите процесс наложения текстур.
- 6 Перечислите основные виды текстур.
- 7 Для чего предназначен параметр loop?
- 8 Что определяет запись PixelTexture {image 2 2 1 0 255 255 0}?

Практическое занятие № 3

Тема: Объекты, строящиеся по набору вершин

Цель: Изучить приемы и методы построения объектов по набору вершин

Задание:

- 1 Создайте документ, содержащий набор точек.
- 2 Создайте документ, содержащий четыре линии.
- 3 Создайте документ, содержащий линии разных цветов.
- 4 Создайте документ, содержащий линии разных градиентных цветов.
- 5 Создайте документ, содержащий куб, построенный при помощи полилиний.
- 6 Создайте документ с изображением трех кубов.
- 7 Создайте платоново тело - икосаэдр (он имеет 20 граней, 30 ребер, 12 вершин).

Необходимые приборы: ПК, текстовый редактор Блокнот, браузер

Методические рекомендации к практической работе:

Методические рекомендации к выполнению задания 1

Для описания объектов необходимо задать координаты вершин, на основе которых строится объект.

PointSet (набор точек)

Описание:

```
PointSet {  
  coordNULL  
  colorNULL }
```

Поскольку по умолчанию coord=color=NULL, то вставка в код узла PointSet без аргументов не приведет ни к чему, не будет создано ни одной точки. Количество записей в разделе **color** должно точно соответствовать количеству точек, описанных в разделе **coord**.

Создайте документ, содержащий набор точек

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

```
Background{      # цветфона  
  skyColor 1 0 0}
```

```
PointSet {  
  coord Coordinate {  
    point [  
      0 1 2 #1 точка  
      1 0 0 #2  
      0 1 0 #3  
      0 0 1 #4  
      1 1 1 #5  
      0 1 0 #6
```

```

1 0 1 # 7
1 0.5 0.7 #8
0.5 1 0 #9
0.5 1 1 #10
] }
color Color { color [ 1 1 1, 0 0.5 0, 0 0.7 1, 0 0.2 0.9, 0 0.5 1, 1 1 1, 0 0.5
0, 0 0.7 1, 0 0.2 0.9, 0 0.5 1 ] } }

```

IndexedLineSet (линии по набору точек)

Описание:

```

IndexedLineSet {
color NULL
coord NULL
colorIndex []
colorPerVertexTRUE
coordIndex []
}

```

Данный узел создает трехмерный объект из точек соединенных полиниями. Набор точек задается в поле *coord* с помощью узла *Coordinate*. Поле *coordIndex* это массив из номеров трехмерных вершин, которые соединяются отрезками. Одна так называемая **полилиния** может состоять из большого количества отрезков и для того, чтобы разделить между собой эти линии снова используется признак "-1".

Методические рекомендации к выполнению задания 2

Создайте документ, содержащий четыре линии

```
#VRMLV2.0 utf8
```

```
Background{
skyColor 1 0.5 1
}

```

```

Group {
children[
Shape {
appearanceAppearance {
# нужно назначить излучающий свет материал,
# чтобы линии были видны, т.к. без этого они не
# отображаются
material Material {
emissiveColor 0 0 0
}
}
}
geometry IndexedLineSet {
# координаты точек
coord Coordinate {
point [
0 0 0 #1 точка

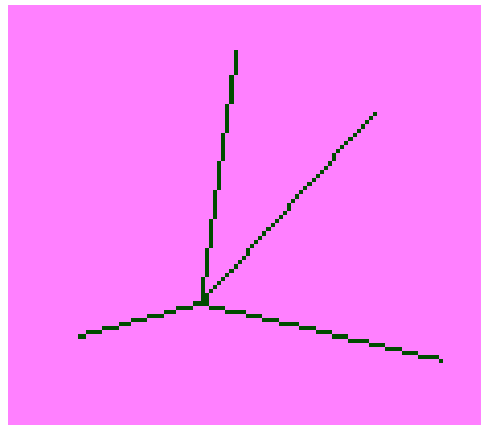
```

```

1 0 0 #2
0 1 0 #3
0 0 1 #4
1 1 1 #5
]
}
# соединения точек, первая линия из точки 0 в точку 1,
# вторая линия из точки 0 в точку 2 и т.д.
# разделителем для полилиний является "-1"
coordIndex [ 0 1 -1 0 2 -1 0 3 -1 0 4 ]
} # конец IndexedLineSet
} # конец Shape
] # конец children
} # конец Group

```

В результате данного примера можно увидеть такую картинку:



Узел *IndexedLineSet* имеет богатые возможности по окрашиванию. Схема использования этого узла заключается в следующем:

- в разделе *coord* описывается набор точек,
 - 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.
 - Для этого узла важно разобраться со способами раскраски отрезков. 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*).
 - 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.
- Если *colorPerVertex* FALSE, то цвета из списка в разделе *color* приписывается каждой полилинии, а не каждому отрезку.

Методические рекомендации к выполнению задания 3

IndexedFaceSet (границы по набору точек)

Прежде чем разбираться с узлом IndexedFaceSet надо разобраться с узлом Coordinate. Определяется он следующим образом:

```
Coordinate {  
  exposedFieldMFVec3f point []  
}
```

Этот узел предельно прост в определении. Его единственный параметр-массив *point[]* определяет набор точек в трехмерном пространстве.

Создайте документ, содержащий линии разных цветов

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

```
Background {  
  skyColor 10.9 0.8}
```

```
Group {  
  children [  
    Shape {  
      geometry IndexedLineSet {
```

```
# координаты точек
```

```
coord Coordinate {
```

```
  point [
```

```
    0 0 0          #1 точка
```

```
    1 0 0          #2
```

```
    0 0.5 0        #3
```

```
    1 0.5 0        #4
```

```
    0 1 0          #5
```

```
    1 1 0          #6
```

```
    0 1.5 0        #7
```

```
    1 1.5 0        #8
```

```
    0 2 0          #9
```

```
    1 2 0          #10
```

```
  ] }
```

```
  colorColor { color [ 1 0 0, 0 1 0, 0 0 1, 0.5 0.2 0.9, 0 0.5 1, 0 0.9 1, 0.8 0.5 0.5,  
    0 1 0, 0 0 1, 0 0 0.8 ] }
```

```
# задание сплошной заливки для каждой линии
```

```
colorPerVertex FALSE
```

```
# соединение точек, из точки 0 в точку 1,
```

```
# из точки 2 в точку 3 и так далее
```

```
# разделением для полилиний является "-1"
```

```
coordIndex [ 0 1 -1 2 3 -1 4 5 -1 6 7 -1 8 9 ]
```

```
  } # конец IndexedLineSet
```

```
  } # конец Shape
```

```
] # конец children
```

```
} # конец Group
```



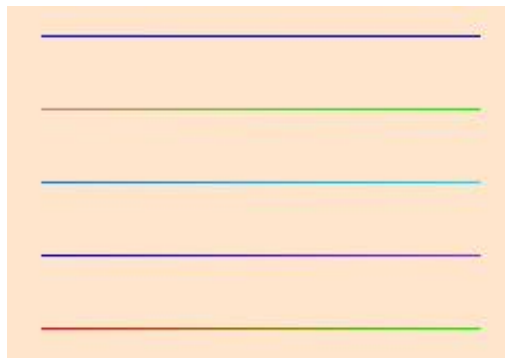
Методические рекомендации к выполнению задания 4

Создайте документ, содержащий линии разных градиентных цветов. Для этого измените предыдущую программу, вместо строки

`colorPerVertex FALSE`

введите строку

`colorPerVertex TRUE`



Методические рекомендации к выполнению задания 5

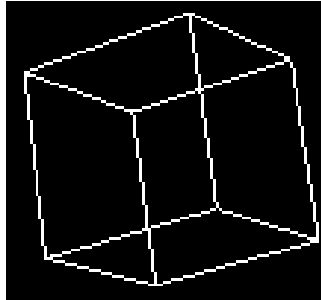
Создайте документ, содержащий куб, построенный при помощи полилиний

```
Background{
  skyColor 0 0 0
}
Group {
  children[
    Shape {
      geometry IndexedLineSet {
        coord Coordinate {
          point [
            0 0 0      #1 точка
            1 0 0      #2
            1 0 1      #3
            0 0 1      #4
            0 1 0      #5
            1 1 1      #6
            0 1 1      #7
```

```

1 1 0 #8 ]}
colorPerVertex FALSE
coordIndex [ 0 1 -1 1 2 -1 2 3 -1 0 3
-1 0 4 -1 3 6 -1 2 5 -1 1 7 -1
4 6 -1 4 7 -1 7 5 -1 6 5
]} ]}]

```



Узел IndexedFaceSet позволяет построить поверхность (а вернее практически трехмерное тело) по заданному набору точек.

Описание:

```

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 []
}

```

Узел, которым можно заменить все остальные узлы, связанные с описанием граней. Принцип работы узла очень похож на IndexedLineSet: описан набор координат точек (*coord*) и указано, какие из них должны образовать грань (*coordIndex*).

Для того, чтобы получить что-то вразумительное, должны выполняться три условия:

1. каждая грань должна состоять как минимум из трех несовпадающих вершин
2. вершины должны задавать ПЛОСКИЙ многоугольник
3. 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.

В поле *color* должен содержаться объект класса *Color*, задающий цвета:

- если *colorPerVertex* == цвет приписывается *вершинам*, а грань заливается градиентом между всеми вершинами, которыми грань создана; при *colorPerVertex* false цвет приписывается каждой *грань* в порядке, соответствующем порядку цветов в разделе *color*

- если не указано поле *colorIndex*, то цвета на многоугольники, определенные в поле *coordIndex*, назначаются просто по порядку следования их описаний (первому описанному многоугольнику соответствует первый цвет из поля *color*, и т.п.);

- если *colorIndex* указан, то цвета на многоугольники назначаются по индексам, указанным в *colorIndex*.

Методические рекомендации к выполнению задания 6

Создайте документ с изображением трех кубов

#VRML V2.0 utf8

Shape { # просто набор многоугольников

appearance Appearance {

material Material { diffuseColor 0.5 0.5 0.5 } }

geometry

IndexedFaceSet {

coord Coordinate { point [0 0 0, -1 1 1, 2 2 2, -3 3 3, 4 -4 4, 5 5 5] }

coordIndex [0, 1, 2, 3, -1, 1, 2, 4, 5, -1]

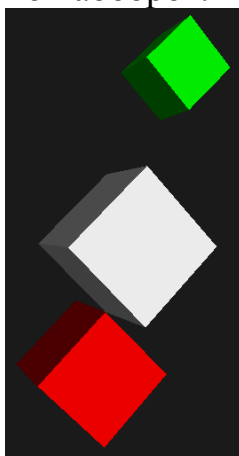
colorPerVertex FALSE

color Color { color [0 1 0, 1 1 1] }

colorIndex [1, 0]

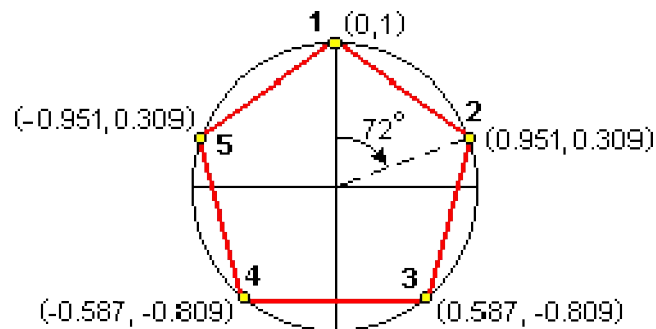
}} # 2-й четырехугольник будет зеленого цвета, а первый белого.

Если не указать colorIndex, то наоборот.

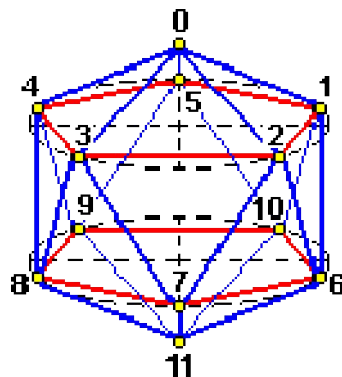


Методические рекомендации к выполнению задания 7

Необходимо задать координаты для 12 точек. Сначала берется окружность и делится на 5 равных частей, при этом получаются точки разбиения, которые будут точками икосаэдра. Вот так выглядит эта поделенная окружность:



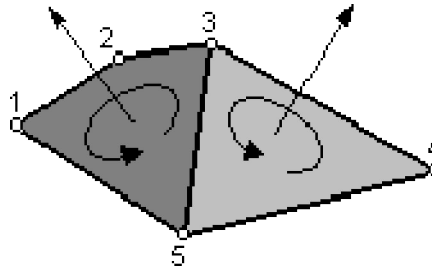
Для того, чтобы получить "объем" берется вторая окружность поделенная таким же образом, но только повернутая в противоположную сторону. Добавляются две вершины: сверху и снизу. Все эти точки соединяются ребрами. Получается следующая фигура:



Плоскости, в которых лежат окружности, расположены в плоскостях $Y=-0.5$ и $Y=+0.5$, при этом центр икосаэдра совпадает с центром координат. На рисунках специально указаны все координаты и пронумерованы вершины, для того, чтобы можно было не запутаться при создании массива точек.

Прежде чем создавать грани экосаэдра надо понять процесс задания этих граней. Грань (плоскость) в VRML задается перечислением как минимум трех точек лежащих на этой плоскости. Всем известно, что через три точки в пространстве можно провести только одну плоскость. Вы можете строить плоскости хоть из 10 вершин, но дело в том, что точность задания координат каждой вершины может привести к тому, что некоторые точки не будут лежать на задаваемой плоскости и иметь мельчайшие отклонения. Поэтому, чтобы не ломать голову расчетами уточненных координат плоскости лучше всегда задавать только тремя точками (вершинами). По нормали программа высчитывает освещенность плоскости, ее видимость для обозревателя и многие другие параметры. Чтобы указать направление нормали в VRML применяется

порядок задания точек. Три точки определяющие плоскость задаются по правилу "буравчика". Давайте сначала посмотрим на рисунок:



На рисунке изображены две плоскости и нормали к ним, направленные таким образом, чтобы эти плоскости были видимыми, т.е. нормальные вектора должны быть направлены в сторону наблюдателя. Еще вы видите направления вращения руки при вкручивании "буравчика". Теперь задание левой плоскости будет таким: 5 3 2 1, т.е. точки перечислены в порядке "вращения" вокруг нормали. Конечно же, перечисление точек может начаться с любой точки, т.е. записи 3 2 1 5 и 1 5 3 2 задают одну и ту же плоскость, самое главное какая точка за какой идет по ходу вращения. Правая плоскость задается, как 3 5 4, и для нее используется рекомендованное количество точек - три. В связи с тем, что количество точек может варьироваться от плоскости к плоскости, разработчики VRML придумали признак-разделитель "-1", который следует после определения каждой плоскости. Например, определение двух этих плоскостей выглядело бы так: 3 2 1 5 -1 3 5 4.

Текст кода выглядит так:

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

```
Shape {
```

```
  appearance Appearance {
```

```
    material Material { diffuseColor 1 0 0 } }
```

```
}
```

```
WorldInfo {
```

```
  info ["IndexedFaceSet example"]
```

```
  title "Step 15 Example"
```

```
}
```

```
Background{
```

```
  skyColor 0 0 0
```

```
}
```

```
Group {
```

```
  children[
```

```
    IndexedFaceSet {
```

```
    # определяем набор из 12 вершин с помощью узла Coordinate
```

```
      coord Coordinate {
```

```
        point [
```

```
          0 1 0 #0
```

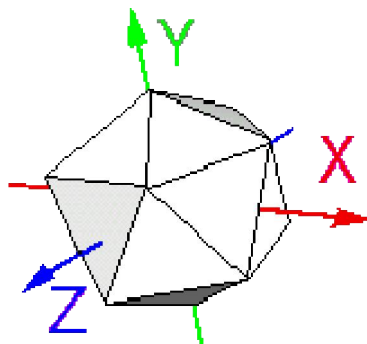
```

        .951 .5 -.309 #1
        .587 .5 .809 #2
        -.587 .5 .809 #3
        -.951 .5 -.309 #4
        0 .5 -1 #5
        .951 -.5 .309 #6
        0 -.5 1 #7
        -.951 -.5 .309 #8
        -.587 -.5 -.809 #9
        .587 -.5 -.809 #10
        0 -1 0 #11
    ]
}
#Определяем 20 граней икосаэдра
coordIndex [
    0 2 1 -1 0 3 2 -1 0 4 3 -1 0 5 4 -1 0 1 5 -1
    1 2 6 -1 2 7 6 -1 2 3 7 -1 3 8 7 -1 3 4 8 -1
    4 9 8 -1 4 5 9 -1 5 10 9 -1 5 1 10 -1 1 6 10 -1
    7 11 6 -1 7 8 11 -1 9 11 8 -1 9 10 11 -1 10 6 11 -1
]
}
]
}

```



Как Вы можете увидеть из примера, вершины перечисляются в параметре **coord** при помощи узла **Coordinate**. Грани же задаются в массиве **coordIndex**, где каждая грань отделяется от других **-1**. Получается такое вот тело:



С помощью **creaseAngle** можно сделать тело сглаженным. Но если мы будем сглаживать икосаэдр, то просто получим из него нечто очень сильно похожее на обычную сферу. После работы с нормальными телами можно также понять смысл поля **solid**. В переводе это "целое" (твердое, прочное и т.д.), т.е. когда этот признак равен **TRUE**, то тело считается цельковым и значит в нем нет пустоты. По этому нормали всех граней тела должны быть направлены из его центра наружу. При этом считается, что грани имеют правильно направленные нормали, и если вдруг наблюдатель окажется внутри тела, то ни одной грани он не увидит, так как векторы нормали направлены от него. Если же вдруг поле **solid** будет равно **FALSE** и у тела не будет определена какая-то одна грань, то через эту "дырку" можно будет видеть все то, что творится с другой стороны, так как в таком случае грани будут считаться двух сторонними (иметь две нормали в обе стороны) и тело будет считаться пустым внутри.

Совместно с **solid** можно также и разобраться с полем **convex**, которое означает, что трехмерное тело выпуклое (**TRUE**). Это означает, что все грани плоские, не пересекают друг друга и внутренние углы между гранями тела меньше 180 градусов, иначе тело будет считаться вогнутым или очень сложным.

Вопросы для самоконтроля:

- 1 Что нужно сделать для описания объектов?
- 2 Для чего предназначен раздел coord?
- 3 Каким образом строятся линии по набору точек?
- 4 Каким образом определяется узел Coordinate?
- 5 Что называют полилиниями?
- 6 Каким образом производится рисование полилиний?
- 7 Каким образом производится построение икосаэдра?

Практическое занятие № 4

Тема: Экструзия, рельеф по набору точек

Цель: Изучить приемы создания и настройки параметров экструзии, рельефа по определенному набору точек

Задание:

- 1 Ознакомиться с теоретическими сведениями.
- 2 Создайте документ по образцу.
- 3 Измените поверхность, созданную в задании 2.
- 4 Нарисуйте изображение домика.

Необходимые приборы: ПК, текстовый редактор Блокнот, браузер

Методические рекомендации к практической работе:

Методические рекомендации к выполнению задания 1

Extrusion (экструзия, выдавливание)

Описание:

```
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  
solidTRUE  
spine [0 0 0, 0 1 0]  
}
```

Разделы **beginCap** и **endCap** определяют, будут ли грани-"крышки" на торцах Вашего объекта. По умолчанию **solid** выставлен TRUE: значения этого раздела регулируются, будут ли видны одна или обе стороны многоугольника одновременно.

В каждой точке траектории многоугольник можно:

- масштабировать (раздел **scale**). Количество значений шкального множителя должно либо быть равным 1 (тогда масштабируется исходное сечение **crossSection** и далее не изменяется), либо совпадать с количеством точек в траектории (**spine**). Если количество значений в разделе **scale** больше единицы, но меньше количества значений в разделе **spine**, то результат спецификацией VRML не определен (обычно получается непонятный объект). Если количество значений в разделе **scale** больше единицы, и больше количества значений в разделе **spine**, то лишние игнорируются.

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

- 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.

ElevationGrid (рельеф по набору точек)

Описание:

```
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 (хоть рыболовецкую) 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 "волоконками" (разделы **xSpacing** и **zSpacing**).

Он задает в пространстве сетку, лежащую в плоскости **XZ** (т.е. на поверхности "земли"), в которой каждая точка пересечения поднята или опущена по отношению к плоскости **Y=0** на высоту определенную в массиве **height[]**.

Данный массив имеет размерность, заданную в количестве элементов сетки по оси **X** и **Z** соответствующими параметрами **xDimension** и **zDimension**. Если мы хотим создать сетку **5x5** элементов, то мы должны задать массив **height** из 25 элементов.

Параметры **xSpacing** и **zSpacing** задают расстояние между линиями сетки в метрах.

Методические рекомендации к выполнению задания 2

Создайте документ:

```
#VRMLV2.0 utf8
```

```
WorldInfo {
```

```
  info ["ElevationGrid Example !!!"]
```

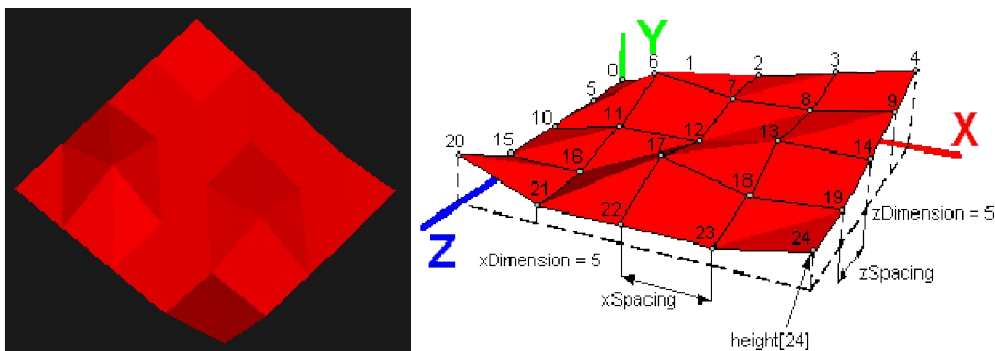
```
  title "ElevationGrid"}
```

```

Group {
  children[
    Shape {
      appearance Appearance {
#          назначаем красный цвет для
#          поверхности
        materialMaterial {
          diffuseColor 1 0 0
        }
      }
      geometry ElevationGrid {
        xDimension 5
        zDimension 5
        xSpacing 0.5
        zSpacing 0.5
        height[
          .1 .2 .3 .4 .5
          .1 .4 .3 .3 .4
          .1 .2 .2 .3 .3
          .1 .1 .3 .2 .2
          .3 .1 .1 .1 .2 ] } } } }

```

После этого браузер отобразит нечто похожее на следующий рисунок:

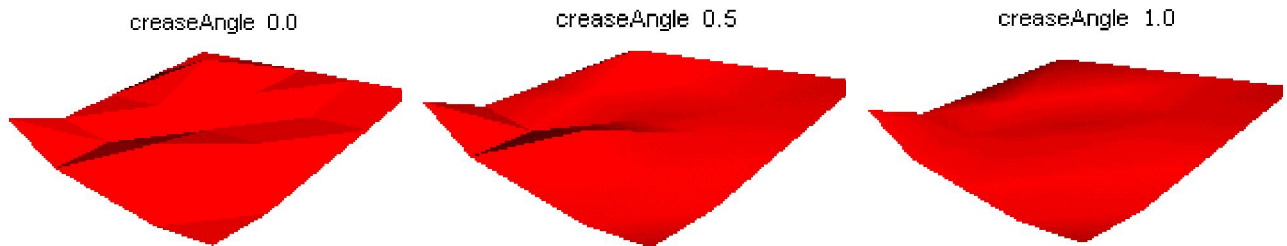


Из рисунка видно, что выпуклая поверхность помещается первой точкой **height[0]** в центр координат. Это отличает данный узел от других, так как рассматриваемые нами узлы до этого устанавливались своим центром в точку **(0,0,0)**.

Также из рисунка можно наглядно понять, как влияет количество точек **xDimension** и **zDimension**, а также расстояние между ними **xSpacing** и **zSpacing** на отображаемую поверхность. Чем больше точек вы зададите, тем более качественную (т.е. сглаженную) поверхность получите, естественно при этом и расстояние между точками надо уменьшать.

Поле **solid**, как и раньше, влияет на количество видимых сторон поверхности. Если поле **solid** установлено в **FALSE**, то поверхность видна с двух сторон, если **TRUE** (значение по умолчанию), то поверхность видна только сверху.

Поле **creaseAngle** влияет на гладкость поверхности. Значение этого поля задается в радианах. Между нормальными двух соседних граней существует геометрический угол, значение этого угла сравнивается со значением поля **creaseAngle**. Если значение поля больше угла между нормальными, то между этими гранями переход отображается сглаженным, иначе угол будет "острым". Следующие три рисунка демонстрируют этот эффект:



Данный массив имеет размерность, заданную в количестве элементов сетки по оси **X** и **Z** соответствующими параметрами **xDimension** и **zDimension**. Если мы хотим создать сетку **5x5** элементов, то мы должны задать массив **height** из 25 элементов.

Параметры **xSpacing** и **zSpacing** задают расстояние между линиями сетки в метрах.

Методические рекомендации к выполнению задания 3

Измените поверхность, созданную в задании 2

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

```
WorldInfo { info ["ElevationGrid Example !!!"]
  title "ElevationGrid" }
```

```
Group {
  children[
```

```
  Shape {
```

```
    appearance Appearance {
      material Material {
        diffuseColor 1 0 0}}
```

```
    geometry ElevationGrid {
      creaseAngle 1
      solid FALSE
```

```
      xDimension 5
```

```
      zDimension 5
```

```
      xSpacing 0.5
```

```
      zSpacing 0.5
```

```
      height[
```

```
        .1 .2 .3 .4 .5
```

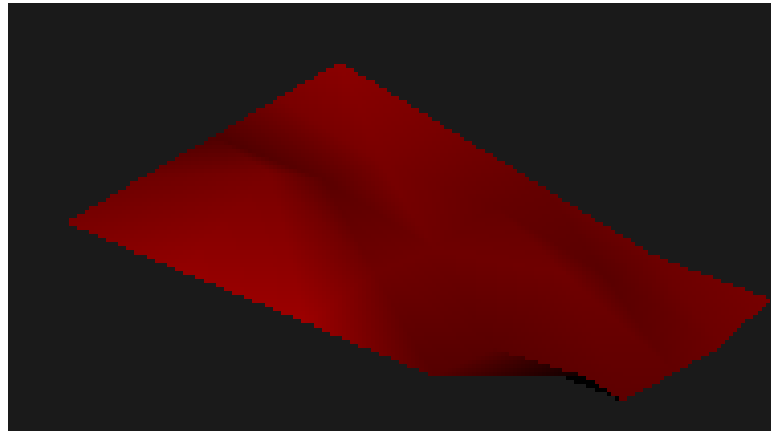
```
        .1 .4 .3 .3 .4
```

```
        .1 .2 .2 .3 .3
```

```
        .1 .1 .3 .2 .2
```

```
.3 .1 .1 .1 .2 ]} ]}]}
```

Послеэтогобраузеротобразитследующийрисунок:



Методические рекомендации к выполнению задания 4

Нарисуйте изображение домика:

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

```
WorldInfo { info ["Example !!!"] title "ElevationGrid"}
```

```
Background {
    skyColor 0.3 0.3 0}
```

```
Group {
```

```
children[
```

```
    Shape {
```

```
        appearance Appearance {
            material Material {
                diffuseColor 1 0 0 }}
```

```
        geometry ElevationGrid {
```

```
            creaseAngle 1
```

```
            solid FALSE
```

```
            xDimension 7
```

```
            zDimension 5
```

```
            xSpacing 0.5
```

```
            zSpacing 0.5
```

```
            height[
```

```
                0 0 0 0 0 0 0
```

```
                0 0.3 0.3 0.3 0.3 0.3 0
```

```
                0 0.6 0.6 0.6 0.6 0.6 0
```

```
                0 0.3 0.3 0.3 0.3 0.3 0
```

```
                0 0 0 0 0 0 0 ]}]}
```

```
        Transform { translation 1.5 -0.75 1
```

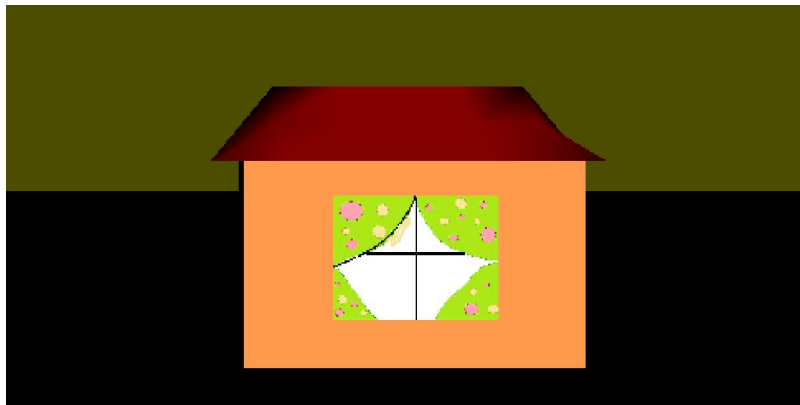
```
            children [
```

```
                Shape { appearance Appearance {
                    material Material {
                        diffuseColor 1 0.6 0.3 }}
```

```

    geometry Box {
        size 2.5 1.5 1.5 } } ] }
    Transform { translation 1.5 -0.7 1.75
    children [
        Shape { appearance Appearance {
            texture ImageTexture {
                url ["im.jpeg"] } }
        geometry Box {
            size 1.2 0.9 0.1 } } ] }
    Transform { translation -50 -1.5 -50
children[ Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0.1 0.1 0.3 } }
    geometry ElevationGrid {
        creaseAngle 1
        solid FALSE
        xDimension 300
        zDimension 300
        xSpacing 0.5
        zSpacing 0.5
        height [ 0 ] } } ] }

```



Самостоятельно добавьте дополнительные окна и дверь.

Вопросы для самоконтроля:

- 1 Что понимают под понятием экструзия?
- 2 Для чего предназначена экструзия?
- 3 Каким образом организовывается построение рельефа по набору точек?

Практическое занятие № 5

Тема: Интерполяторы

Цель: Изучить приемы работы с интерполяторами

Задание:

- 1 Ознакомиться с теоретическими сведениями.
- 2 Создайте документ по образцу с использованием интерполятора цвета.
- 3 Создайте документ с использованием интерполятора нормалей.
- 4 Создайте документ с использованием интерполятора ориентации.
- 5 Создайте документ с использованием скалярного интерполятора.

Необходимые приборы: ПК, текстовый редактор Блокнот, браузер

Методические рекомендации к практической работе:

Методические рекомендации к выполнению задания 1

Интерполяторы играют очень важную роль в создании анимации, но при этом крайне просты в использовании. Суть заключается в том, что они меняют некоторую величину с течением времени. В зависимости от того, какую именно величину вы хотите менять, различают шесть узлов-интерполяторов. Поведение их во многом подобно, и механизмы действия очень схожи.

Механизм действия

Интерполяторы применяются там, где нужно менять какую-то величину с течением времени. Они принимают сигналы времени с узла TimeSensor или подобного ему и производят линейную интерполяцию между множествами значений, именуемых ключевыми. Для каждого ключевого значения существует ключ (key), дробь в интервале от 0 до 1. Таймер генерирует события fractionchanged со скоростью, на которую только способен. Это событие может быть маршрутизировано на входящее событие setfraction интерполятора для того, чтобы установить правильную точку цикла. Так, если вы маршрутизируете на интерполятор таймер с интервалом цикла, равным 10, интерполятор будет проходить по циклу свои ключевые значения каждые 10 с. Всякий раз, когда интерполятор получает входящее событие, он генерирует исходящее событие с параметром, равным интерполированному значению величины. Оно, в свою очередь, может быть маршрутизировано на любое поле подходящего типа. Узлы могут быть помещены в любое место сцены, будь то корень или узел Transform. Поскольку они являются узлами, отвечающими только за "моторчик" интерполяции, то их конкретное место в иерархии сцены значения не имеет. Имена полей у всех интерполяторов одинаковы. Разница между ними заключается лишь том, что в полях keyValue (ключевое значение) и исходящих событиях value_changed находятся разные типы данных.

Итак, сначала идет входящее событие set_fraction. Оно получает события от таймера. Это дает информацию о текущем моменте цикла анимации и указывает, как далеко мы зашли в своем интерполяционном процессе.

Следующее поле - поле key (ключ). Это последовательность значений от 0 до 1, соответствующих ключевым точкам анимации. Если вы хотите, чтобы в нулевой момент времени, в середине и в конце объект занимал определенное положение, то вы создадите поле key со значением [0, 0.5, 1]. Каждое значение в поле key должно иметь соответствующее ему значение в поле keyValue, так что, если следовать приведенному выше примеру и использовать ScalarInterpolator, который интерполирует величины типа SFFloat, мы можем создать поле keyValue [0.0, 10.0, 0.0]. В момент времени 0 исходящее значение будет 0.0, в середине цикла - 10.0, а в 1 (конец цикла) - опять 0.0. Между этими моментами времени значения будут интерполироваться линейно, т. е. в момент 0.25 цикла посылаемое значение будет равно 4. Значения передаются в качестве параметра исходящего события value_changed всякий раз, когда интерполятор получает событие fraction_changed.

Методические рекомендации к выполнению задания 2

Узлы-интерполяторы позволяют задавать ключевые кадры анимации и предоставляют браузеру позаботиться обо всем остальном. Таким образом, каркас анимации задается сжато и просто. Нужно только помнить что для того, чтобы анимация текла плавно, без рывков, в нуле и единице поле keyValue должно иметь одно и то же значение. А теперь посмотрим на конкретные узлы-интерполяторы.

Первый на нашем пути - интерполятор цвета.

```
ColorInterpolator {
  eventIn    SFFloat setfraction# (-inf.inf)
  exposedField MFFloat key    [ ]    # (-inf.inf)
  exposedField MFColor keyValue [ ]    #[0,1]
  eventOut   SFColor value_changed }
```

Можно догадаться, что он интерполирует RGB-тройку цветов. Цвета декларируются в поле keyValue точно так же, как и в любом поле типа MFColor. Если мы хотим, чтобы цвет изменялся от ярко-зеленого через синий к красному, то напомним следующее:

```
key [0,0.5,1]           keyValue [010, 001, 100]
```

Заметьте, что здесь все-таки будет скачок цвета в конце интервала цикла. В приводимом ниже примере цвет сферы и текста под ней равномерно изменяется под воздействием таймера:

Создайте документ:

```
#VRMLV2.0 utf8 WorldInfo { title "VRMLbookExample 26_1"}
NavigationInfo { headlight FALSE}
DEF T1 TimeSensor{
  cycleInterval 10
  loop TRUE}
#Colour Interpolator
Group { children [ DirectionalLight { ambientIntensity 0.75}
  DEF CI1 ColorInterpolator {
    key [0, 0.25, 0.50, 0.75,1]
```

```

keyValue [ 1 0 0,1 0,0 0 1,0 0,1 1 ] }
Transform { translation 0 2 0 children [ Shape{
appearance Appearance { material DEF M1 Material {
  diffuseColor 1 0 0 } } geometry Sphere {} } ] } Shape{
appearance Appearance { material USE M1}
geometry Text {
string ["Color Interpolator"] fontStyle FontStyle {justify "MIDDLE" } } } ]}
ROUTE T1 .fraction_changed TO CI1.set_fraction ROUTE
CI1.value_changed TO M1.set_diffuseColor

```



Заметьте, что мы выключили автономный источник света у путешественника, но объединили оба геометрических узла в одну группу и добавили к ней источник направленного света.

Методические рекомендации к выполнению задания 3

Интерполятор нормалей - NormalInterpolator.

```

NormalInterpolator {
eventIn SFFloat set_fraction # (-inf.inf)
exposedField MFFloat key [ ] # (-inf.inf)
exposedField MFVec3f keyValue [ ] # (-inf.inf)
eventOut MFVec3f value_changed }

```

Этот узел может посылать множественные значения, в данном случае нормалей, так что вы можете манипулировать нормальями ко всем граням объекта одновременно. В приводимом примере мы используем значения условных нормалей для расчета, а затем плавного изменения освещенности квадрата.

Создайте документ:

```

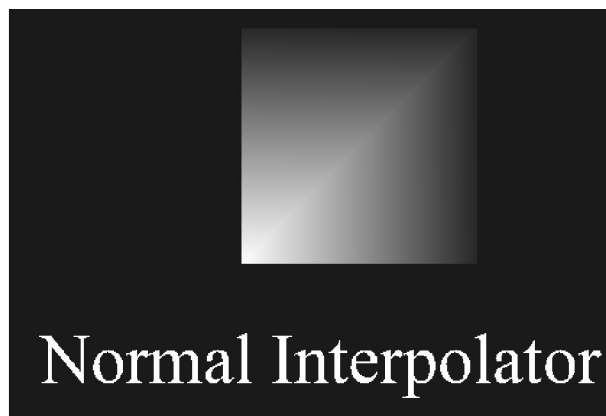
#VRMLV2.0 utf8
NavigationInfo { headlightFALSE}
DEF T1 TimeSensor {
cycleInterval 10
loop TRUE}
Group { children [ DirectionalLight { ambientIntensity 0.75 }
DEF N11 NormalInterpolator { key [0, 0.25, 0.5, 0.75, 1] keyValue [
0 0 1, 0 0 1, 1 0 0, 0 0 1,

```

```

0 0 1,      1 0 0.5, 1 0 0,      1 0 0.5,
0 0 1,      1 0 0,  1 0 0,      1 0 0,
0 0 1,      1 0 0.5, 1 0 0,      1 0 0.5,
0 0 1,      0 0 1,  1 0 0,      0 0 1]]}
Transform { translation 1 -1.0 0 children [ Shape{
appearance Appearance { material Material {
emissiveColor 1 1 1 } }
geometry Text { string ["Normal Interpolator"] fontStyle FontStyle {
justify "MIDDLE" } } } ] }
Transform { translation 0 0.5 0 scale 3 3 2 children [ Shape{
appearance Appearance { material Material {
diffuseColor 1 1 1 #1 0.5 0.5 } }
geometry IndexedFaceSet { coord Coordinate {
point [ 0 0 0, 0 1 0, 1 1 0, 1 0 0 ]}
coordIndex [ 0 1 2 3,]
normal DEF N1 Normal { vector [ 0 0 1, 1 0 0.5, -1 0 0, 1 0 0.5]}
normalPerVertex TRUE normalIndex [3012] ccw FALSE } } ] } }
ROUTE    T1    .fraction_changed    TO    N11.set_fraction    ROUTE
N11.value_changed TO N1.set_vector

```



В этом примере направление обхода грани - по часовой стрелке, на что указывает значение `FALSE` в поле `ccw` узла `IndexedFaceSet`. Кроме того, порядок условных нормалей переопределен в поле `normalIndex`.

Методические рекомендации к выполнению задания 4

Далее следует интерполятор ориентации - `OrientationInterpolator`.

```

OrientationInterpolator {
eventIn  SFFloat set_fraction  # (-inf,inf)
exposedField MFFloat key  [ ] # (-inf,inf)
exposedField MFRotation keyValue  [ ] # [-1,1],(-inf,inf)
eventOut  SFRotation value_changed } '

```

Интерполятор ориентации производится сглаживанием между значениями поворота (ось + угол) и может быть маршрутизировано событие `setrotation` узла `Transform`.

Таким образом, мы уже можем создавать анимированные точки обзора. Если мы поместим точку обзора внутрь узла Transform, а сам узел Transform заставим изменять угол поворота, путешественник будет вращаться вокруг своей оси, не меняя точки, на которой он стоит.

Создайте документ:

```
#VRMLV2.0 utf8
```

```
WorldInfo {
```

```
  title "VRMLtraok Example 26_4" }
```

```
  Background {
```

```
    groundColor      [0 0.5 0, 0 0.9 0]
```

```
    groundAngle      [1.50]
```

```
    skyAngle [1.00,1.57]
```

```
    skyColor [1 1 1, 0.9 0.9 0.9, 0 0 1]
```

```
    backUrl ["image2.jpeg"]
```

```
    frontUrl ["inage1.jpeg"]
```

```
    leftUrl  ["image3.jpeg"]
```

```
    rightUrl ["image1.jpeg"]
```

```
  } Viewpoint {
```

```
    description "Entry"
```

```
    position 0 0 0
```

```
    fieldOfView .50}
```

```
  DEF T1 TimeSensor { cycleInterval 10 loop TRUE}
```

```
  #Orientation Interpolator Example
```

```
  DEF N1 OrientationInterpolator { key [0, 0.25, 0.5, 0.75,1]
```

```
    keyValue [ 0 10 0, 0 10 1.57, 0 10 3.14, 0 10 4.71, 0 1 0 6.28 ]}
```

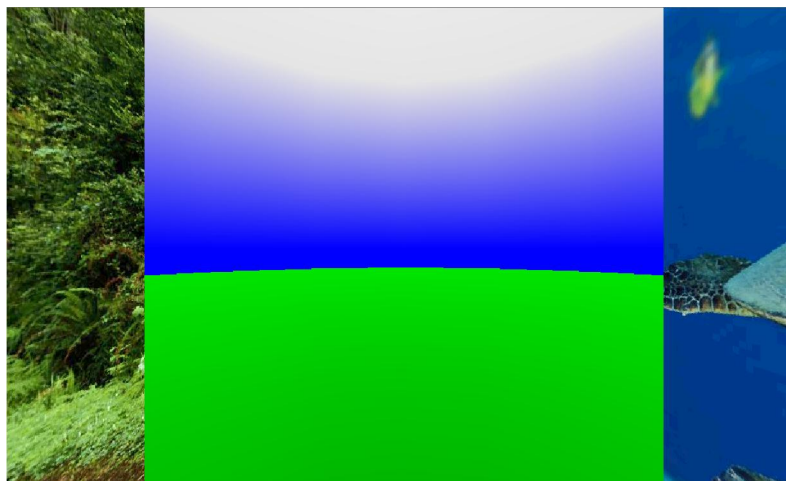
```
  DEF R1 Transform { translation 0 3 -10 children [ Viewpoint {
```

```
    description "Rotation" position 0 0 0 fieldOfView 1.50 } ] }
```

```
  ROUTE T1 .fraction_changed TO N1.set_fraction ROUTE
```

```
N1.value_changed TO R1.set_rotation
```

Сразу после загрузки мира вращаться еще ничего не будет. Для того чтобы запустить движение, нужно переключиться на точку наблюдения Rotation.



```

ScalarInterpolator
Scalar-Interpolator {
  eventIn SFFloat set_fraction # (-inf.inf)
  exposedField MFFloat key [ ] # (-inf.inf)
  exposedField MFFloat keyValue [ ] # (-inf.inf) eventOut SFFloat
value__changed }

```

Скалярный интерполятор соединяет действительные значения. С его помощью можно влиять на интенсивность света, прозрачность объекта или любую другую скалярную величину.

Можно "обратить" таймер, установив интерполяцию между 1 в момент 0 и 0 в момент 1.

Код программы:

```

#VRML V2.0 utf8
WorldInfo { title "VRMLbook Example 26_5"}
NavigationInfo { headlight FALSE}
DEF T1 TimeSensor
{ cycleInterval 10
loop TRUE}#Scalar Interpolator
Transform { rotation 0 1 0 0 children [
DEF DL1 DirectionalLight { ambientIntensity 0.75}
DEF SI1 ScalarInterpolator
{ key [0, 0.25, 0.5, 0.75, 1]
keyValue [ 0, 0.5, 0, 1,0]}
Transform { translation 0 2 0 children [ Shape {
appearance DEF A1 Appearance { material Material {
diffuseColor 1 1 0 } } geometry Cylinder {} } ] }
Transform { translation 0 -0.5 0 children [ Shape{
appearance USE A1 geometry Text { string ["Scalar Interpolator"] fontStyle
FontStyle {
justify "MIDDLE" } } } ] } ]
}
ROUTE T1 .fraction_changed TO SI1.set_fraction ROUTE
SI1.value_changed TO DL1 .set_intensity

```



Теперь, комбинируя вместе все изученные приемы, вы способны создавать миры внушительной сложности.

Вопросы для самоконтроля:

- 1 Что понимают под понятием интерполятор?
- 2 Для чего предназначена интерполяторы?
- 3 Каким образом организовывается построение интерполяторов?
- 4 Перечислите виды интерполяторов. В чем их отличия?