Технологии обработки, анализа и визуализации данных
Лабораторная работа 6. Низкоуровневая визуализация в Processing

Цель работы:

Необходимо на основе динамических или сериальных данных (по выбранному варианту) сформулировать задачу для визуализации и разработать интерактивное приложение в Processing для решения этой задачи и его интерпретации.

Задачи:

1. Изучить исходные данные, сформулировать задачу для визуализации данных, обосновать выбор методов для подготовки и визуализации данных.
2. Подготовить данные для визуализации.
3. Визуализировать данные в приложении, создать отдельный класс визуализации.
4. Внедрить в приложение методы интерактивного управления визуализацией.
5. Интерпретировать полученную визуализацию и оценить насколько она решает поставленную задачу.

Рекомендуемая литература:

1. Fry Ben – Visualizing Data – 2007 – O Reilly
2. Chen Chun Houh – Handbook of Data Visualization – 2008 – Springer
3. Yau Nathan – Visualize This – 2011 – Wiley
4. Edward R. Tufte – Visual Explanations: Images and Quantities, Evidence and Narrative – 1997 – Graphucs Press

Методические указания

1. Прежде всего, определитесь с целями и задачами предстоящей работы. Именно они во многом могут повлиять на ход моделирования, степень детализации, разрешение выходного изображения, глубину цвета, поля и т. д.
2. Определите тип представления визуализации - статичный, анимированный или интерактивный. В соответствии с выбранным типом, назначьте ключевые элементы визуализации, их контраст цвета или формы.
3. Используйте метафоры для отображения символов. Старайтесь, как можно использовать лексические формы представления комментариев, т.е. создавайте приложение, ориентированное на многоязычную аудиторию.

4. Упрощайте визуальное представление - минимум цветовых оттенков, отображают способ, которым пользователь может взаимодействовать с минимум образов и форм, максимум информации передаваемой этими элементами.

Processing — это подязык программирования, основанный на Java с простым и понятным си-подобным синтаксисом. Processing дает возможность быстро и легко создавать мультимедиа приложения (в терминологии processing — скетчи). Благодаря поддержке OpenGL, Processing позволяет создавать двухмерные и трёхмерные аппликации, в том числе и игры.

Procesing является, открытым и кроссплатформенным ПО. Исходный архив включает в себя java-машину, сам интерпретатор, мини-IDE, и несколько десятков примеров. После загрузки и распаковки архива, необходимо найти в корневом каталоге исполняемый файл, запускающий IDE, в котором и пишется код программы. [1].

Любая программа в Processing по умолчанию из двух блоков: setup(), который выполняется один раз перед выполнением другого метода по умолчанию draw(), который уже выполняется с частотой, совпадающей с заданной частотой кадров. Данные для чтения/записи хранятся в папке data, расположенной рядом с файлами подпрограмм.

Processing был выбран так как довольно прост в освоении, имеет вполне широкие возможности работы в трёхмерном пространстве и предоставляет всё необходимое для работы из поставляемого дистрибутива

Данные Processing — это подязык программирования, основанный на Java с простым и понятным си-подобным синтаксисом. Processing дает возможность быстро и легко создавать мультимедиа приложения (в терминологии processing — скетчи). Благодаря поддержке OpenGL, Processing позволяет создавать двухмерные и трёхмерные аппликации, в том числе и игры.

Procesing является, открытым и кроссплатформенным ПО. Исходный архив включает в себя java-машину, сам интерпретатор, мини-IDE, и несколько десятков примеров. После загрузки и распаковки архива, необходимо найти в корневом каталоге исполняемый файл, запускающий IDE, в котором и пишется код программы. [1].

Любая программа в Processing по умолчанию из двух блоков: setup(), который выполняется один раз перед выполнением другого метода по умолчанию draw(), который уже выполняется с частотой, совпадающей с заданной частотой кадров. Данные для чтения/записи хранятся в папке data, расположенной рядом с файлами подпрограмм.

Processing был выбран так как довольно прост в освоении, имеет вполне широкие возможности работы в трёхмерном пространстве и предоставляет всё необходимое для работы из поставляемого дистрибутива

Данные

Ирисы Фишера - самый популярный в статистической литературе набор данных, часто используемый для иллюстрации работы различных алгоритмов классификации. При всем желании мы не смогли без него обойтись, поскольку в современных реальных приложениях редко встречаются такие компактные наборы данных, позволяющие построить хороший классификатор при минимуме исходных признаков.

Выборка состоит из 150 экземпляров ирисов трех видов, для которых измерялись четыре характеристики: длина и ширина чашелистика (Sepal.Length и Sepal.Width), длина и ширина лепестка (Petal.Length и Petal.Width). [3]

Набор ирисов Фишера предоставлен нам в формате файла CSV, для начала работы с которым, его необходимо прочитать и подготовить для визуализации.

Для этого написан был написан метод csvToArray(), предназначенный для чтения CSV-файла сразу в массив готовых для отображения объектов. Метод сначала создаёт массив объектов класса Bubble, затем в таблицу считывает CSV файл с ирисами с помощью метода loadTable().

После этого через цикл мы проходим по каждой строке таблицы, получая пространственные параметры цветков, в качестве координат для плоскости и видовую принадлежность цветка, чтобы потом присвоить отдельному объекту цвет.

Также координаты умножаются на 1/10 часть глобальной переменной size, для сохранения масштаба, так как size отвечает за реальный размер будущего графика. Из полученных данных создаётся объект Bubble, который заносится в массив и цикл переходит на новую итерацию.

В конце метод возвращает массив класса Bubble, которые уже можно передать на отображение. Код метода csvToArray():

```
int size = 500;

ArrayList<Bubble> csvReadToArray()
{
  ArrayList<Bubble> bubbles = new ArrayList<Bubble>();
  Table table = loadTable("iris.csv", "header"); for
  (TableRow row: table.rows())
  {
    float x = (size/10)*row.getFloat("sepal.length");
    float y = (size/10)*row.getFloat("sepal.width");
    float z = (size/10)*row.getFloat("petal.length");
    String variety = row.getString("variety"); color
    colour;
    colour = color(255, 255, 255);
    switch(variety)
    {
      case "Setosa":
        colour = color(194, 76, 0, 76);
        break;
```

```
      case "Versicolor":
        colour = color(0, 194, 136, 76);
        break;
      case "Virginica":
        colour = color(194, 8, 76);
        break;
    }
    bubbles.add(new Bubble(x, y, z, colour, 5));
  }
  return bubbles;
}
```

В итоге полученный массив будет присвоен переменной bubbles в методе setup(), но так как мы будем использовать её ещё и в методе draw(), необходимо определить эту переменную в глобальном пространстве:

```
ArrayList<Bubble> bubbles;
```

Для работы в трёхмерном пространстве необходимо в методе setup() при помощи метода size() установить размер окна и указать, что мы работаем в трёхмерном пространстве с помощью встроенной константы P3D:

```
size(500, 500, P3D);
```

Реализация класса визуальных объектов

Для того, чтобы мы могли отображать элементы из набора данных на графике, мы должны для начала создать эти самые объекты. В прошлой части был упомянут класс Bubble, который и является реализацией отдельного визуального объекта.

Чуть ранее мы указали программе, что собираемся работать в трёхмерном пространстве, а значит, наши объекты, которые мы будем отображать на графике должны быть так же трёхмерными, а потому в его полях указаны его координаты x,y,z, а так же поле c, отвечающее за цвет самого шара.

```
class Bubble
{
  float x;
  float y;
  float z;
  color c;

  Bubble(float xcord, float ycord, float zcord, color clr)
  {
    x = xcord;
    y = ycord;
    z = zcord;
    c = clr;
  }

  void display()
  {
     pushMatrix();
     noStroke();
     lights();
     translate(x, y, z);
     fill(c);
     sphere(r);
     popMatrix();
  }
```

```
}
```

При создании сферы, значения введенных координат и цвета записываются в вещественных полях x, y, z и переменной цвета c.

Для размещения в пространстве в классе реализован метод display(), в котором указывается точка размещения методом translate(x, y, z), применяется окрашивание методом fill(c) и создаётся уже собственно готовая сфера радиусом 5 с помощью метода sphere(), который рисует, саму сферу с заданным радиусом.

Уже в методе draw() основной программы для вывода сфер на экран мы воспользуемся циклом for и методом display() из Bubble:

```
for (int i = bubbles.size()-1; i>=0; i--)
{
   Bubble bubble = bubbles.get(i);
   bubble.display();
}
```

Результат выполнения программы представляет собой представление данных в виде цветных сфер, расположенных в пространстве (Рис.1):

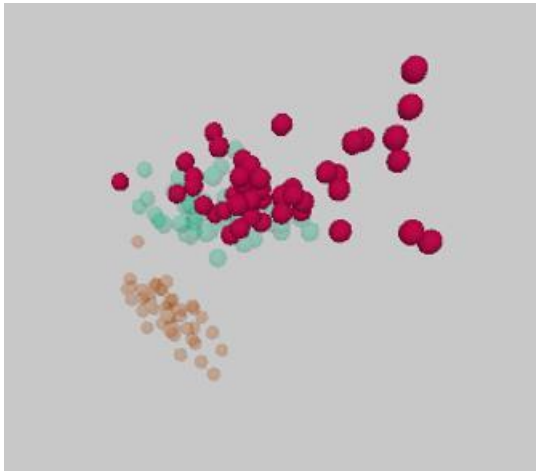

Рисунок 1 – Вывод ирисов Фишера в пространстве

Создание координатных плоскостей

Координатная плоскость (декартова) — плоскость, на которой задана система координат. Самый простой примитив, который можно расположить на плоскости — точка. Её положение определяется двумя координатами, и в математике она записывается так: (x, y), где первое число — координата по оси x, а второе — по оси y. В коде её можно представить, как массив, состоящий из двух элементов. [2]

В данной работе мы работаем в трёхмерном пространстве, поэтому каждая точка уже будет иметь координаты, а сами точки не обязательно будут принадлежать одной из плоскостей. Тем не менее, для того, чтобы мы могли визуально рассмотреть закономерности, координатные плоскости, на которые мы можем ориентироваться, необходимы, а потому построим их.

Сначала мы создадим три линии осей при помощи методов stroke() для задания цвета контура линии, построенной методом line(), где линии откладываются от точки (0, 0, 0) в точку (0, 0, size), и методом fill() для окрашивания уже самой линии в красный, зеленый и синий цвета для каждой из осей:

```
stroke(255, 0, 0, 100);
line(0, 0, 0, 0, 0, size);
fill(255, 0, 0);
stroke(0, 255, 0, 100);
line(0, 0, 0, 0, size, 0);
fill(0, 255, 0);
stroke(0, 0, 255, 100);
line(0, 0, 0, size, 0, 0);
fill(0, 0, 255);
```

Затем мы сделаем подписи к осям с помощью метода text() на серединах осей для наглядности:

```
stroke(255, 0, 0, 100);
line(0, 0, 0, 0, 0, size);
fill(255, 0, 0);
stroke(0, 255, 0, 100);
line(0, 0, 0, 0, size, 0);
fill(0, 255, 0);
stroke(0, 0, 255, 100);
line(0, 0, 0, size, 0, 0);
fill(0, 0, 255);
```

Для построения собственно координатной сетки мы воспользуемся циклом, в котором в качестве итератора будет шаг в 1/10 от упомянутой переменной size, и каждый шаг на каждой из плоскостей строится вертикальная и горизонтальная линия и подписывается на оси её значение в сантиметрах с помощью переменной delta:

```
stroke(200);
int iter = size/10;
int delta = 1;
for(int i = iter; i <=size; i = i + iter)
{
  line(0, 0, i, 0, size, i);
  line(i, 0, 0, i, 0, size);
  line(0, i, 0, 0, i, size);
  line(0, 0, i, size, 0, i);
  pushMatrix();
  textSize(14);
  text(delta, i, 0, 0);
  text(delta, 0, i, 0);
  text(delta, 0, 0, i);
  popMatrix();
  delta++;
}
```

Плоскость xOy мы зальем серым фоном для удобства методом rect():

```
fill(200);
rect(0, 0, size, size);
```

В итоге получаем более удобную для анализа визуализацию, с которой уже можно работать:

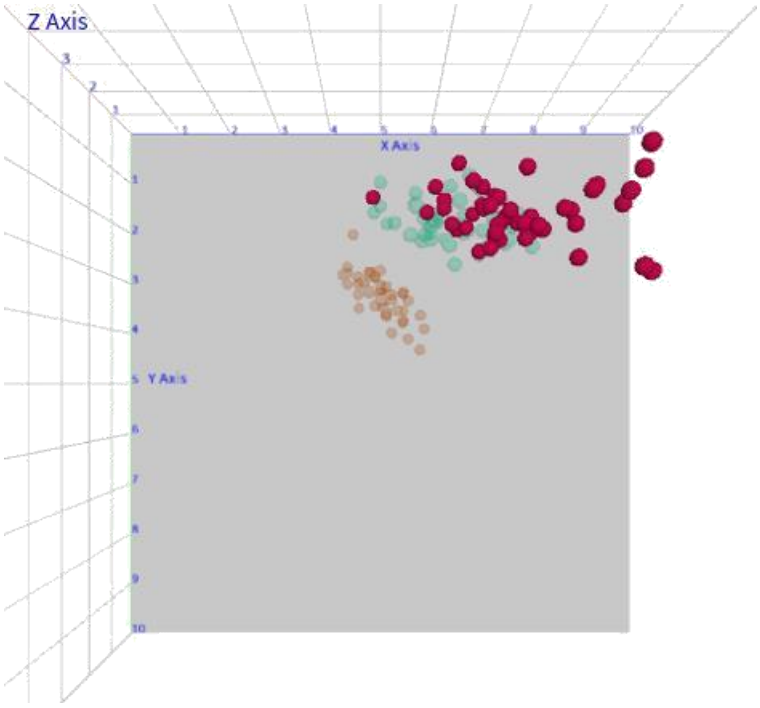

Рисунок 2 – Визуализация с координатными плоскостями **Реализация интерактивных возможностей**

Трёхмерные графики неудобно анализировать, если рассматривать их только с одной перспективы, поэтому мы сделаем возможность вращать график с помощью мыши, чтобы можно было более наглядно рассмотреть зависимости.

Для реализации интерактивного взаимодействия с помощью мыши мы воспользуемся mouseDragged() – встроенным в Processing методом который выполняет код внутри своего тела с частотой, с какой выполняется метод draw(), при зажатии любой кнопки мыши. В нашем случае мы будем считывать текущие координаты мыши по X и Y с помощью системных переменных mouseX и mouseY соответственно и на их основе рассчитывать смещения по осям X и Y, которые затем пригодятся для вращения фигуры. Для этого так же заранее определим переменные xcord, ycord, newXcord, newYcord в глобальном пространстве:

```
float xcord, ycord;
float newXcord, newYcord;

void mouseDragged()
{
   newXcord  =  mouseX/float(width)  *  TWO_PI;
   newYcord = mouseY/float(width) * TWO_PI;

   float diff = xcord-newXcord;
   if(abs(diff)>0.01)
   {
     xcord -= diff/2.0;
   }

   diff = ycord-newYcord;
   if(abs(diff)>0.01)
   {
     ycord -= diff/2.0;
   }
```

```
}
```

В методе draw() полученные координаты используются для вращения фигуры по осям методами rotateX и rotateY:

```
rotateX(-ycord);
rotateY(-xcord);
```

Для изменения приближения фигуры мы воспользуемся методом mouseWheel(), возвращающим 1 или -1 при прокрутке колеса мыши в вверх или вниз соответственно, и так мы будем изменять переменную scale, отвечающую за масштаб и передаваемую в метод scale(), масштабирующий изображение:

```
float scale;

scale(scale);

void mouseWheel(MouseEvent e)
{
   float diff = e.getCount();
   scale = scale + diff/10;
}
```

Так как иногда может понадобиться сброс перспективы на первоначальную, мы реализуем данный функционал на нажатие левой кнопки мыши с помощью присвоения начальных координат в теле метода mousePressed():

```
void mousePressed()
{
  if(mouseButton == LEFT)
  {
     xcord = 0;
     ycord = 0;
     scale = 1;
  }
}
```

В итоге мы получаем возможность свободно вращать изображение по осям X и Y, изменять масштаб и при необходимости возвращать всё в изначальный вид.