XXXX

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Программирование графики с использованием Java 2D

Методические указания к лабораторной работе

Рязань 2010

УДК 681.3

Программирование графики с использованием Java 2D: методические указания к лабораторной работе / Рязан. гос. радиотехн. ун-т.; сост. А.А. Митрошин, А.В.Бакулев. – Рязань, 2010. – 16 с.

Содержат описание лабораторной работы, используемой в курсе «Инженерная и компьютерная графика».

Предназначены для студентов дневной и заочной форм обучения направления «Информатика и вычислительная техника».

Табл. 1. Ил. 6. Библиогр.: 2 назв.

Программирование, графика, Java, Java 2D.

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра САПР вычислительных средств Рязанского государственного радиотехнического университета (зав. кафедрой засл. деят. науки и техники РФ В.П.Корячко)

Программирование графики с использованием Java 2D

Составители: Митрошин Александр Александрович
Бакулев Александр Валерьевич

Редактор Р.К. Мангутова
Корректор С.В. Макушина
Подписано в печать 00.00.0000. Формат бумаги 60×84 1/16.
Бумага газетная. Печать трафаретная. Усл. печ. л. 1,0.
Уч-изд. л. 1,0. Тираж 100 экз. Заказ
Рязанский государственный радиотехнический университет.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.

Лабораторная работа № 3

В предыдущей лабораторной работе мы познакомились с объектом `Graphics2D`, методом это объекта `setRenderingHints()`, позволяющим управлять качеством и скоростью визуализации двумерного изображения, пером (stroke) и способами управления параметрами пера, заливками (paint) и графическими примитивами. В этой лабораторной работе будут рассмотрены другие возможности Java 2D API, такие как определение фигур отсечения, двумерные преобразования и композиция.

Преобразование координат

Часто при программировании вывода некоторого изображения удобнее использовать не координаты, заданные в пикселях, а реальные физические размеры рисуемых объектов. Например, при программировании вывода плана здания удобнее указывать длины стен, выраженные в метрах.

В Java 2D для решения такой задачи может использоваться метод `scale()` класса `Graphics2D`, который выполняет преобразование координат графического контекста. При этом пользовательские координаты (единицы измерения, указанные пользователем) преобразуются в аппаратные координаты (пиксели).

```
// Указываем, что один метр объекта надо рисовать
// отрезком в 25 пикселей при откладывании длины
// по оси X и 25 пикселей по оси Y
g2.scale(25,25);
// Рисуем отрезок, который моделирует реальный объект,
// который начинается в точке с физическими координатами
// (50,50) – в метрах и заканчивается в точке (100,100) – метрах
```

```
g2.draw(new Line2D.Double(50,50,100,100));
```

В Java 2D доступно четыре основных типа преобразования координат:

- Масштабирование: растяжение или сжатие всех расстояний от фиксированной точки (рис. 1). Математически масштабирование описывается следующим матричным выражением:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} Sx \cdot x \\ Sy \cdot y \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix},$$

где (*x* ,*y*) – координаты исходной точки, а (*x'* ,*y'*) – координаты точки после преобразования, *Sx* и *Sy* – коэффициенты масштабирования по осям *x* и *y*.

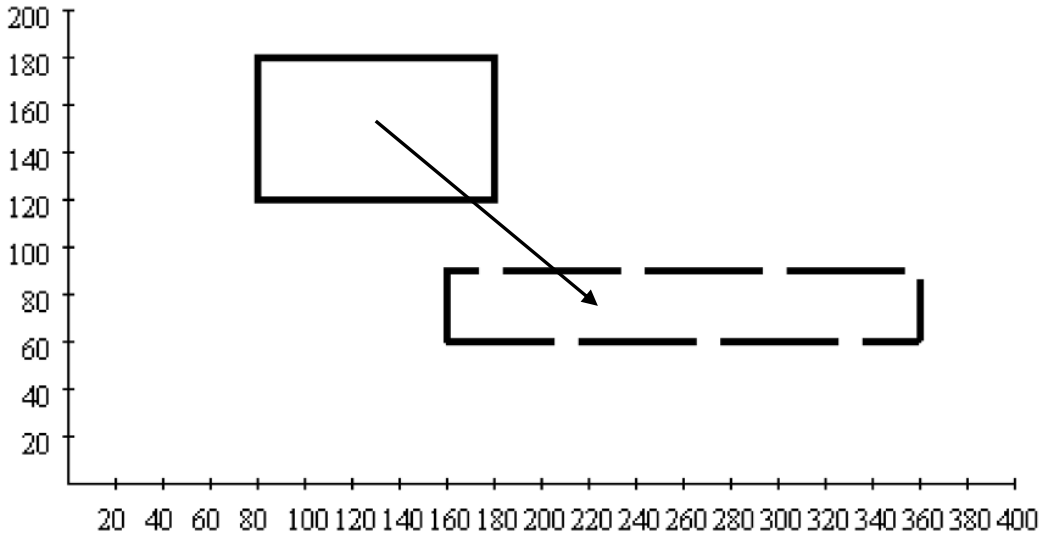


Рис. 1. Масштабирование

- Поворот: вращение всех точек фигуры вокруг фиксированной точки (рис. 2.).

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \cdot cos\varphi - y \cdot sin\varphi \\ x \cdot sin\varphi + y \cdot cos\varphi \end{bmatrix} = \begin{bmatrix} cos\varphi & -sin\varphi \\ sin\varphi & cos\varphi \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix},$$

где (*x* ,*y*) – координаты исходной точки, φ - угол поворота, а (*x'* ,*y'*) – координаты точки после преобразования.

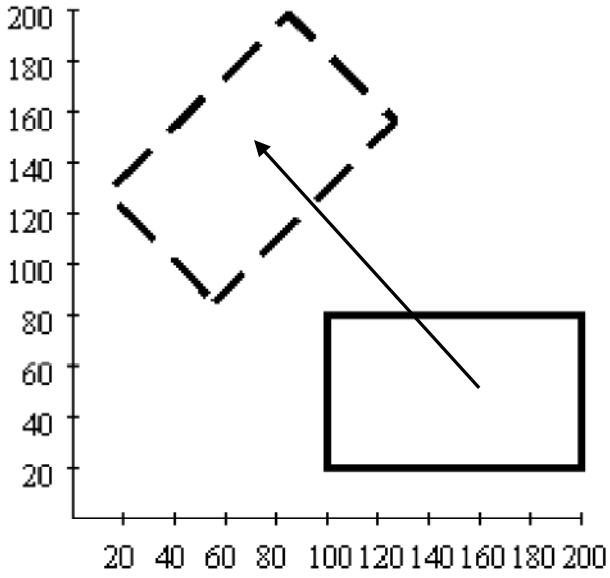


Рис. 2. Поворот

- Перемещение: смещение всех точек фигуры на фиксированное расстояние (рис. 3).

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix},$$

где *dx* и *dy* – величины переноса по осям *x* и *y*.

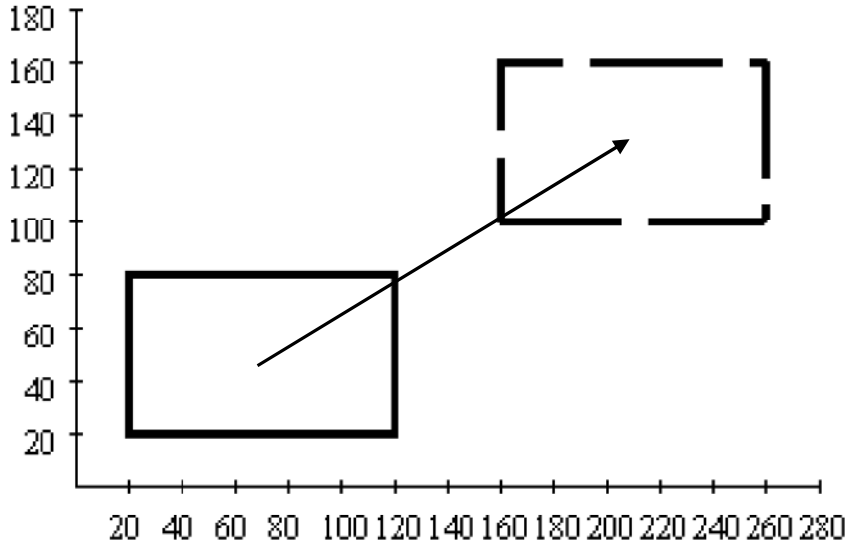


Рис. 3. Перемещение

- Сдвиг: фиксация одной линии и перемещение всех параллельных ей линий фигуры на расстояние, пропорциональное расстоянию от данной линии до фиксированной линии (рис. 4).

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x + Sx \cdot y \\ y + Sy \cdot x \end{bmatrix} = \begin{bmatrix} 1 & Sx \\ Sy & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

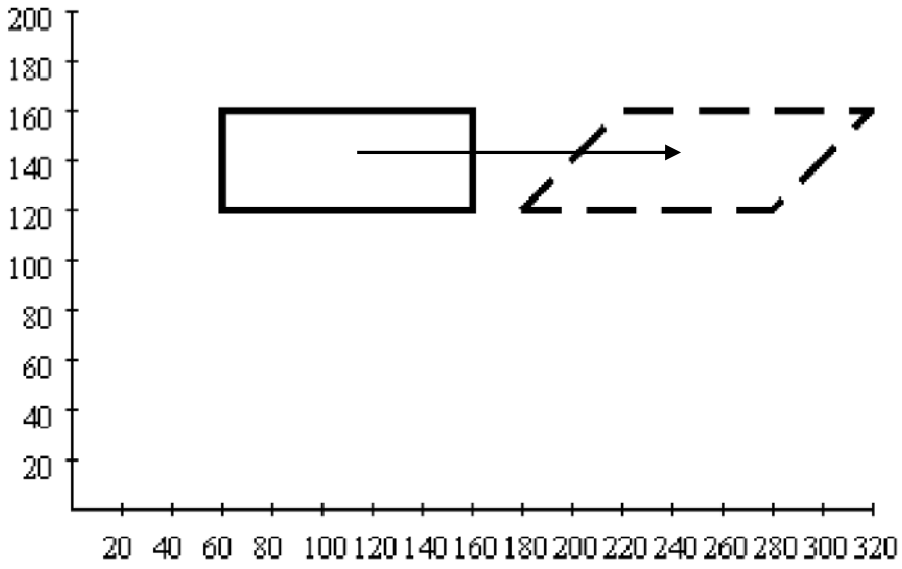


Рис. 4. Сдвиг

Описанные выше элементарные преобразования часто приходится комбинировать. Математическое описание комбинации с использованием описанных выше выражений не всегда удобно, поэтому вводятся, так называемые *однородные координаты*. В однородных координатах любая комбинация описанных ранее преобразований может быть описана с помощью единственной матрицы вида $\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$.

Все эти преобразования называются *аффинными*. Аффинные преобразования и их композиции широко используются в компьютерной

графике благодаря следующим очень важным свойствам, позволяющим эффективно их реализовывать в графических системах [3].

- Аффинные преобразования сохраняют прямые линии, то есть прямая преобразуется в прямую, треугольник в треугольник и т.д.
- Аффинные преобразования сохраняют параллельность, то есть параллельные прямые отображаются в параллельные прямые.
- Аффинные преобразования сохраняют эллипсы, но круг не всегда отображается в круг.
- Аффинные преобразования сохраняют кривые Безье.

Правило преобразования координат в графическом контексте устанавливается с помощью метода `setTransform()`. Параметром этого метода служит объект класса `AffineTransform` из пакета `java.awt.geom`. Как следует из названия класса, он предназначен для описания аффинных преобразований.

Аффинное преобразование задаётся двумя основными конструкторами класса `AffineTransform`:

```
AffineTransform(double a, double b, double c, double d, double e, double f);
AffineTransform(float a, float b, float c, float d, float e, float f).
```

При этом точка с координатами (*x, y*) в пространстве пользователя преобразуется в точку с координатами `(a*x+c*y+e, b*x+d*y+f)` в пространстве графического устройства.

Следующие два конструктора используют в качестве параметра массив из шести элементов коэффициентов преобразования *{ a, b, c, d, e, f }* или массив из четырех элементов *{a, b, c, d}*, если *e=f=0*.

```
AffineTransform(double[] arr);
AffineTransform(float[] arr);
```

Существует также конструктор, который создаёт тождественное аффинное преобразование (которое ничего не изменяет):

```
AffineTransform();
```

Эти конструкторы не очень удобны при задании конкретных преобразований. Во многих случаях удобнее создавать преобразование статическими методами класса `AffineTransform`, возвращающими объект класса `AffineTransform`:

- `getRotateInstance(double angle)` – возвращает преобразование поворота относительно начала координат на угол `angle`, заданный в радианах. Если оси координат пользователя не менялись преобразованием отражения, то положительное значение `angle` задаёт поворот по часовой стрелке.
- `getRotateInstance(double angle, double x, double y)` – возвращает преобразование поворота относительно точки с координатами (x, y).
- `getRotateInstance(double vx, double vy)` – возвращает поворот, заданный вектором с координатами (vx, vy) относительно начала координат. Метод эквивалентен методу `getRotateInstance(Math.atan2(vx,vy))`.
- `getRotateInstance(double vx, double vy, double x, double y)` – аналогичен предыдущему методу, но определяет поворот не относительно начала координат, а относительно точки с координатами (x, y).
- `getQuadrantRotateInstance(int n)` – определяет поворот n-раз по 90° относительно начала координат.
- `getQuadrantRotateInstance(int n, double x, double y)` - аналогичен предыдущему методу, но определяет

поворот не относительно начала координат, а относительно точки с координатами (x, y).

- `getScaleInstance(double sx, double sy)` – изменяет масштаб по оси Ox в `sx` раз, а по оси Oy – в `sy` раз.
- `getTranslateInstance(double tx, double ty)` – сдвигает каждую точку (x, y) в точку (x+tx, y+ty).
- `createInverse()` – возвращает преобразование, обратное текущему преобразованию.

После того, как преобразование `at` создано, его необходимо сделать текущим с помощью метода

`setTransform(AffineTransform at).`

После этого всё, что выводится, будет подвергаться текущему (то есть установленному в данный момент) преобразованию.

После того, как текущее преобразование установлено, его можно *заменить* с помощью одного из следующих методов, синтаксис и семантика большинства из которых понятны, по аналогии с описанными выше методами.

```
setTransform(double a, double b, double c, double d,
                                   double e, double f);
```

`setToIdentity()` – устанавливает тождественное преобразование, то есть преобразование, которое ничего не преобразовывает;

```
setToRotation(double angle);
setToRotation(double angle, double x, double y);
setToRotation(double vx, double vy);
setToRotation(double vx,double vy,double x,double y);
setToQuadrantRotation(int n);
setToQuadrantRotation(int n, double x, double y);
setToScale(double sx, double sy);
```

`setToTranslate(double tx, double ty).`

Часто полезно использовать композицию некоторого аффинного преобразования с текущим преобразованием, то есть сначала выполнить некоторое преобразование, а к его результатам применить текущее преобразование графического контекста. Для выполнения такого рода композиций можно использовать следующие методы:

```
concatenate(AffineTransform at);
rotate(double angle);
rotate(double angle, double x, double y);
rotate(double vx, double vy);
rotate(double vx, double vy, double x, double y);
quadrantRotate(int n);
quadrantRotate(int n, double x, double y);
scale(double sx, double sy);
shear(double shx, double shy);
translate(double tx, double ty).
```

Преобразование, задаваемое методом `preConcatenate(AffineTransform at)` осуществляется *после* текущего преобразования. Это кажется необычном, поскольку префикс pre обычно означает то, что происходит раньше. Дело в том, что графический контекст выполняет преобразования в порядке, обратном тому, в котором они были указаны [1]. Например, последовательность команд

```
g2.rotate(angle);
g2.scale(2,2);
g2.draw(…);
```

выполняется следующим образом: сначала осуществляется масштабирование по каждой из осей, затем осуществляется поворот, потом осуществляется преобразование, которое является текущим для

графического контекста, затем выводится фигура, преобразованная таким образом. Теперь понятно, почему преобразование, задаваемое с помощью метода `preConcatenate()` выполняется после текущего преобразования контекста: здесь pre надо трактовать как то, что в последовательности команд преобразования предшествует командам преобразования контекста, то есть выполняется позже.

Области отсечения

Для выполнения любых графических операции внутри заданной области в Java 2D используются так называемые *фигуры отсечения* (clipping shape) графического контекста. На рис. 5 представлено некоторое изображение (слева) и два изображения, которые получены применением отсечения: в центре фигура отсечения – треугольник, справа фигура отсечения – окружность.

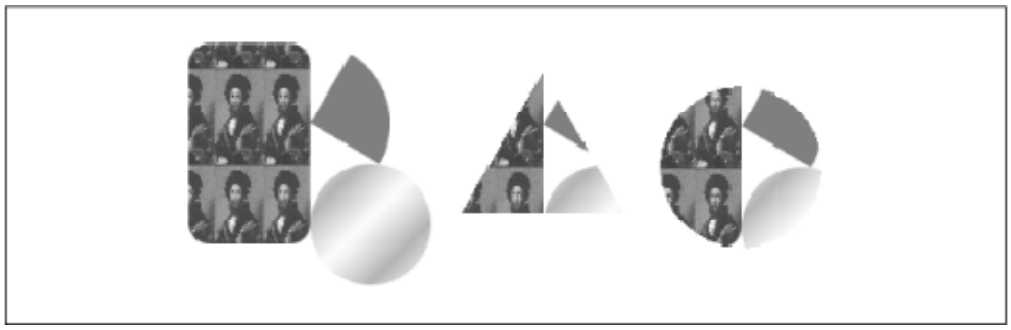


Рис. 5. Отсечение

Для определения фигуры отсечения необходимо вызвать метод `clip()` графического контекста и передать в качестве параметра этого метода объект, класс которого реализует интерфейс `Shape()`. Например, следующий код создает область отсечения в виде эллипса

```
g2.clip(new Ellipse2D.Double(1,1,50,75)).
```

Прозрачность и композиция

При описании цвета (например, в цветовой модели RGB) могут указываться не только составляющие цвета, но альфа-канал, определяющий степень прозрачности цвета. Полагается, что при значении альфа-канала для пикселя равному 0 пиксель совершенно прозрачен, при значении альфа-канала для пикселя равному 1 пиксель совершенно не прозрачен. При наложении изображения на уже существующее изображение частично прозрачные пиксели накладываемого изображения не закрывают пикселей, находящихся под ними, а смешиваются с нижележащими пикселями. Например, так описывается цвет со значением альфа-канала 0.5:

```
Color color = new Color(100, 70, 200, 0.5).
```

При наложении двух фигур смешиваются (компонуются) цвета и значения альфа-каналов соответствующих пикселей. Т. Портер и Т. Дафф сформулировали 12 возможных правил композиции, которые реализованы в Java 2D. Рассмотрим эти правила, называемые правилами Портера-Даффа. Будем называть *источником* изображение, которое рисуется, а *целевым* изображение, являющиеся фоном (то, по которому рисуют). Допустим, что пиксель источника имеет значение альфа-канала a_S, целевой пиксель значение альфа-канала a_D. На рис. 6 показана диаграмма, поясняющая правила композиции для этих значений.

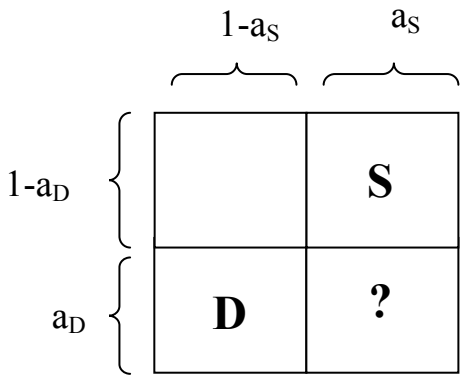


Рис. 6. Правила композиции

Портер и Дафф полагают, что значение альфа-канала выражает вероятность использования цвета пикселя при описании изображений. Для пикселя источника с вероятностью a_S будет использоваться исходный цвет, а с вероятностью $1-a_S$ – не будет. Это справедливо и для целевого цвета. Предположим, что при объединении цветов эти вероятности являются независимыми. Если нужно преимущественно использовать цвет пикселя источника, а не целевого пикселя (эта ситуация обозначена буквой S на рис. 1), то вероятность этого события ровна $a_S(1-a_D)$. Аналогично вычисляется вероятность $a_D(1-a_S)$ преимущественного использования целевого пикселя, а не пикселя источника (эта ситуация обозначена символом D на рис. 1). Что же делать, если источник и целевое изображение считают свой цвет преимущественным. В этой ситуации и используются правила Портера-Даффа. Если предпочтение отдаётся цвету источника, то в таком случае правый нижний угол диаграммы помечается буквой S, а само правило называется SRC_OVER. В нём комбинируются цвет источника с весом a_S и целевой цвет с весом $a_D(1-a_S)$. Визуальным эффектом применения этого правила является смешение пикселя источника и целевого пикселя с приоритетом для пикселя источника. Если a_S =1, то цвет целевого пикселя не учитывается, если a_S =0, то пиксель источника полностью прозрачен и цвет целевого пикселя не изменяется.

В зависимости от того, как расставлены буквы на диаграмме, можно сформулировать и правила, которые перечислены в табл. 1.

Табл. 1. Правила композиции

CLEAR	Изображение-источник удаляет пиксели целевого изображения	
SRC	Изображение-источник переписывает пиксели целевого изображения и пустые пиксели	ColorR= ColorS AlphaR= AlphaS+(1-AlphaS)* AlphaD

Табл. 1. Правила композиции (окончание)

DST	Изображение-источник не влияет на целевое изображение	
SRC_OVER	Изображение-источник смешивается с целевым изображением и переписывает пустые пиксели	ColorR= ColorS+(1-AlphaS)*ColorD AlphaR= AlphaS+(1-AlphaS)* AlphaD
DST_OVER	Изображение-источник не влияет на целевое изображение и переписывает пустые пиксели	ColorR= ColorS*(1-AlphaD)+ColorD AlphaR= AlphaS*(1-AlphaD)+AlphaD
SRC_IN	Изображение-источник переписывает только пиксели целевого изображения	ColorR= ColorS*AlphaD AlphaR= AlphaS*AlphaD
SRC_OUT	Изображение-источник удаляет пиксели целевого изображения и переписывает пустые пиксели	ColorR= ColorS*(1-AlphaD) AlphaR= AlphaS*(1-AlphaD)
DST_IN	Значение альфа-канала источника модифицирует пиксели целевого изображения	ColorR= ColorD*AlphaS AlphaR= AlphaD*AlphaS
DST_OUT	Дополнение до значения альфа-канала источника модифицирует пиксели целевого изображения	ColorR= ColorD*(1-AlphaS) AlphaR= AlphaD*(1-AlphaS)
SRC_ATOP	Часть изображения-источника, лежащая внутри целевого изображения, смешивается с целевым изображением	
DST_ATOP	Изображение-источник смешивается с целевым изображением	
XOR	Дополнение до значения альфа-канала источника модифицирует целевое изображение. Пустые пиксели перезаписываются.	

По умолчанию используется правило композиции SRC_OVER.

Работа двух правил композиции показана на рис. 7.

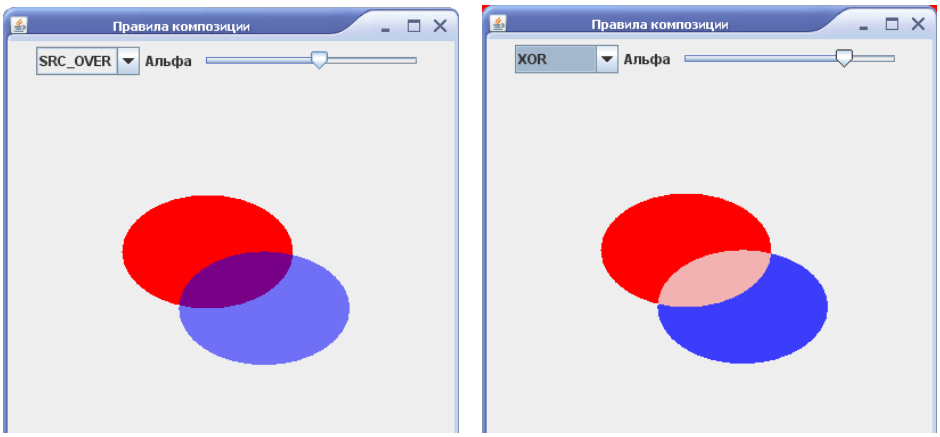


Рис. 7. Правила композиции SRC_OVER и XOR

Для установки правила композиции используется метод `setComposite()` класса `Graphics2D`, которому в качестве параметра передаётся объект, который реализует интерфейс `Composite`. В Java 2D содержится только один такой класс – `AlphaComposite`, который реализует правила Портера-Даффа.

Для создания экземпляра правила типа `AlphaComposite` нужно указать для метода `getInstance()` этого класса правило композиции и значение альфа-канала для пикселей изображения-источника:

```
int rule = AlphaComposite.SRC_OVER;
float alpha=0.5f;
g2.setComposite(AlphaComposite.getInstance(rule,alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle).
```

В этом примере прямоугольник заполняется синим цветом и для него задаётся значение альфа-канала 0.5. Поскольку установлено правило композиции SRC_OVER, его пиксели смешиваются с учетом коэффициента прозрачности с уже существующим изображением.

Порядок выполнения работы

1. Изучите теоретический материал.

2. Выполните задание преподавателя, связанное с разработкой программы, строящей изображение с использованием Java 2D.

3. Ответьте на контрольные вопросы.

Контрольные вопросы

1. Расскажите об аффинных преобразованиях. Какие вы знаете элементарные аффинные преобразования?

2. Почему вводятся однородные координаты?

3. Прокомментируйте конструкторы класса `AffineTransform`.

4. Для чего используются методы `getXXXInstance`?

5. Что такое фигуры отсечения и как они определяются?

6. Что такое альфа-канал и для чего он используется?

7. Прокомментируйте правила композиции Портера-Даффа.

8. Как определяются правила композиции в Java 2D API?

Библиографический список

1. Хорстман К., Корнелл Г. Java 2. Библиотека профессионала, том 2. Тонкости программирования. – М.: ООО «И.Д. Вильямс», 2009.

2. Хабибуллин И. Самоучитель Java. – СПб.: БХВ-Петербург, 2008.

3. Юань Фень. Программирование графики для Windows. – СПб.: Питер, 2002.

4. Хортон А. Java 2 (в двух томах), т. 2. – М.: Лори, 2008.

Варианты заданий

Вариант 1
Вариант 2
Вариант 3
Вариант 4
Вариант 5
Вариант 6
Вариант 7
Вариант 8
Вариант 9
Вариант 10
Вариант 11
Вариант 12

Приложение 1

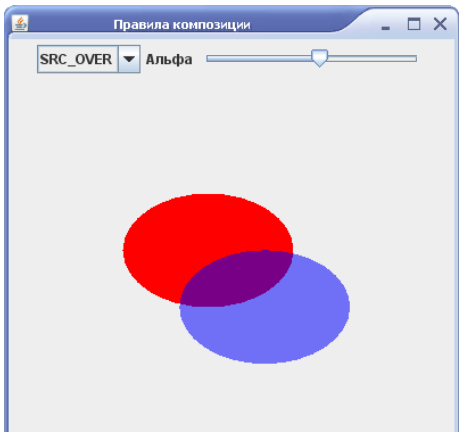
Текст программы, демонстрирующей правила композиции Портера-Даффа и пример её работы

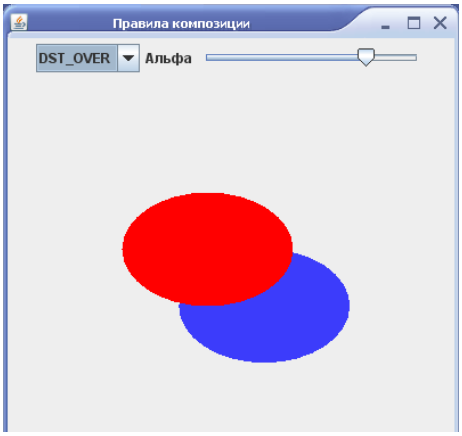
```
// Файл CompositeTest.java
// Демонстрация правил Портера-Даффа
import java.awt.AlphaComposite;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.EventQueue;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Shape;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.Ellipse2D;
import java.awt.image.BufferedImage;
import javax.swing.JComboBox;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JTextField;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class CompositeTest {
     public static void main (String[] args) {
     EventQueue.invokeLater(new Runnable(){
     public void run() {
     JFrame frame = new CompositeTestFrame();
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setVisible(true);
          }
     });
}
}
class CompositeTestFrame extends JFrame {
     private static final long serialVersionUID = 1L;
     public CompositeTestFrame() {
          this.setTitle("Правила композиции");
          this.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
          canvas=new CompositeComponent();
          add(canvas, BorderLayout.CENTER);
          ruleCombo = new JComboBox(new Object[] {
                    new Rule("CLEAR","  ","  "),
                    new Rule("SRC"," S"," S"),
                    new Rule("DST","  ","DD"),
                    new Rule("SRC_OVER"," S","DS"),
```

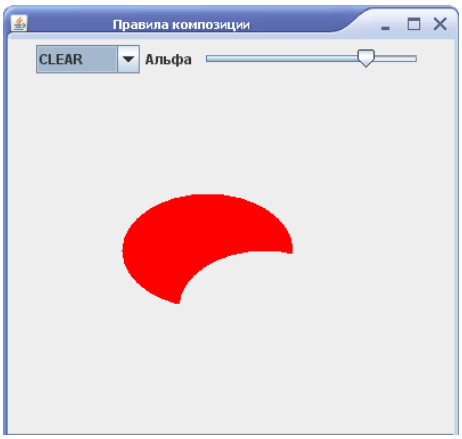
```
                    new Rule("DST_OVER"," S","DD"),
                    new Rule("SRC_IN","  "," S"),
                    new Rule("SRC_OUT"," S","  "),
                    new Rule("DST_IN","  "," D"),
                    new Rule("DST_OUT","  ","D "),
                    new Rule("SRC_ATOP","  ","DS"),
                    new Rule("DST_ATOP"," S"," D"),
                    new Rule("XOR"," S","D "),});
          ruleCombo.addActionListener(new ActionListener()
          {
          public void actionPerformed(ActionEvent event){
               Rule r=(Rule)ruleCombo.getSelectedItem();
               canvas.setRule(r.getValue());
               explanation.setText(r.getExplanation());
               }
               });
          alphaSlider=new JSlider(0,100,75);
          alphaSlider.addChangeListener(
          new ChangeListener() {
               public void stateChanged(ChangeEvent event) {
                    canvas.setAlpha(alphaSlider.getValue());
               }
          });
          JPanel panel=new JPanel();
          panel.add(ruleCombo);
          panel.add(new JLabel("Альфа"));
          panel.add(alphaSlider);
          add(panel,BorderLayout.NORTH);
          explanation = new JTextField();
          add(explanation, BorderLayout.SOUTH);
          canvas.setAlpha(alphaSlider.getValue());
          Rule r=(Rule)ruleCombo.getSelectedItem();
          canvas.setRule(r.getValue());
          explanation.setText(r.getExplanation());
     }
     private CompositeComponent canvas;
     private JComboBox ruleCombo;
     private JSlider alphaSlider;
     private JTextField explanation;
     private static final int DEFAULT_WIDTH=400;
     private static final int DEFAULT_HEIGHT=400;
}
class Rule{
public Rule(String n, String pd1, String pd2) {
     name=n;
     porterDuff1=pd1;
     porterDuff2=pd2;
}
public String getExplanation() {
StringBuilder r=new StringBuilder("Source ");
if (porterDuff2.equals("  ")) r.append("clears");
```

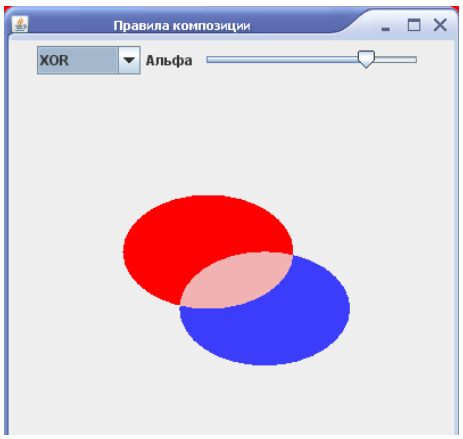
```
if (porterDuff2.equals(" S")) r.append("overwrites");
if (porterDuff2.equals("DS")) r.append("blends with");
if (porterDuff2.equals(" D")) r.append("alpha modifies");
if (porterDuff2.equals("D ")) r.append("alpha component modifies");
if (porterDuff2.equals("DD")) r.append("does not affect");
r.append(" destination");
if (porterDuff1.equals(" S")) r.append(" and overwrites empty pixels");
r.append(".");
return r.toString();
}
public String toString() {
     return name;
}
public int getValue() {
     try
     {
return(Integer)AlphaComposite.class.getField(name).get(null);
     }
     catch(Exception e) {
          return -1;
     }
}
private String name;
private String porterDuff1;
private String porterDuff2;
}
class CompositeComponent extends JComponent{
     public CompositeComponent() {
          shape1=new Ellipse2D.Double(100,100,150,100);
          shape2=new Ellipse2D.Double(150,150,150,100);
     }
     public void paintComponent(Graphics g) {
          super.paintComponent(g);
          Graphics2D g2=(Graphics2D) g;
          BufferedImage image=new BufferedImage(getWidth(),
                         getHeight(), BufferedImage.TYPE_INT_ARGB);
          Graphics2D gImage=image.createGraphics();
          gImage.setPaint(Color.RED);
          gImage.fill(shape1);
          AlphaComposite composite=
                    AlphaComposite.getInstance(rule, alpha);
          gImage.setComposite(composite);
          gImage.setPaint(Color.BLUE);
          gImage.fill(shape2);
          g2.drawImage(image,null,0,0);
     }
     public void setRule(int r) {
          rule=r;
          repaint();
     }
     public void setAlpha(int a) {
```

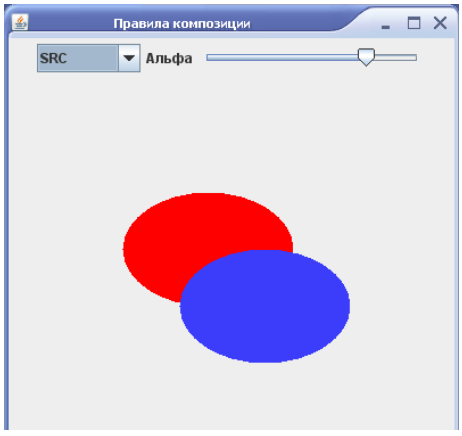
```
            alpha=(float)a/100.0F;
            repaint();
        }
        private int rule;
        private Shape shape1;
        private Shape shape2;
        private float alpha;
}
```

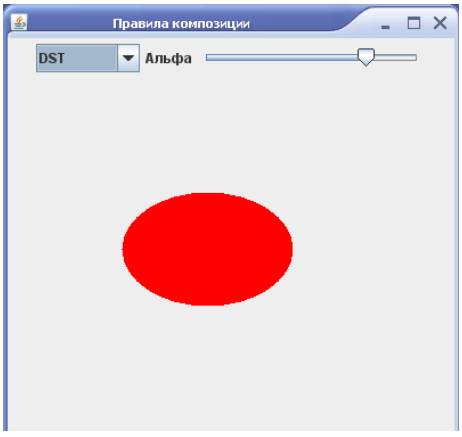












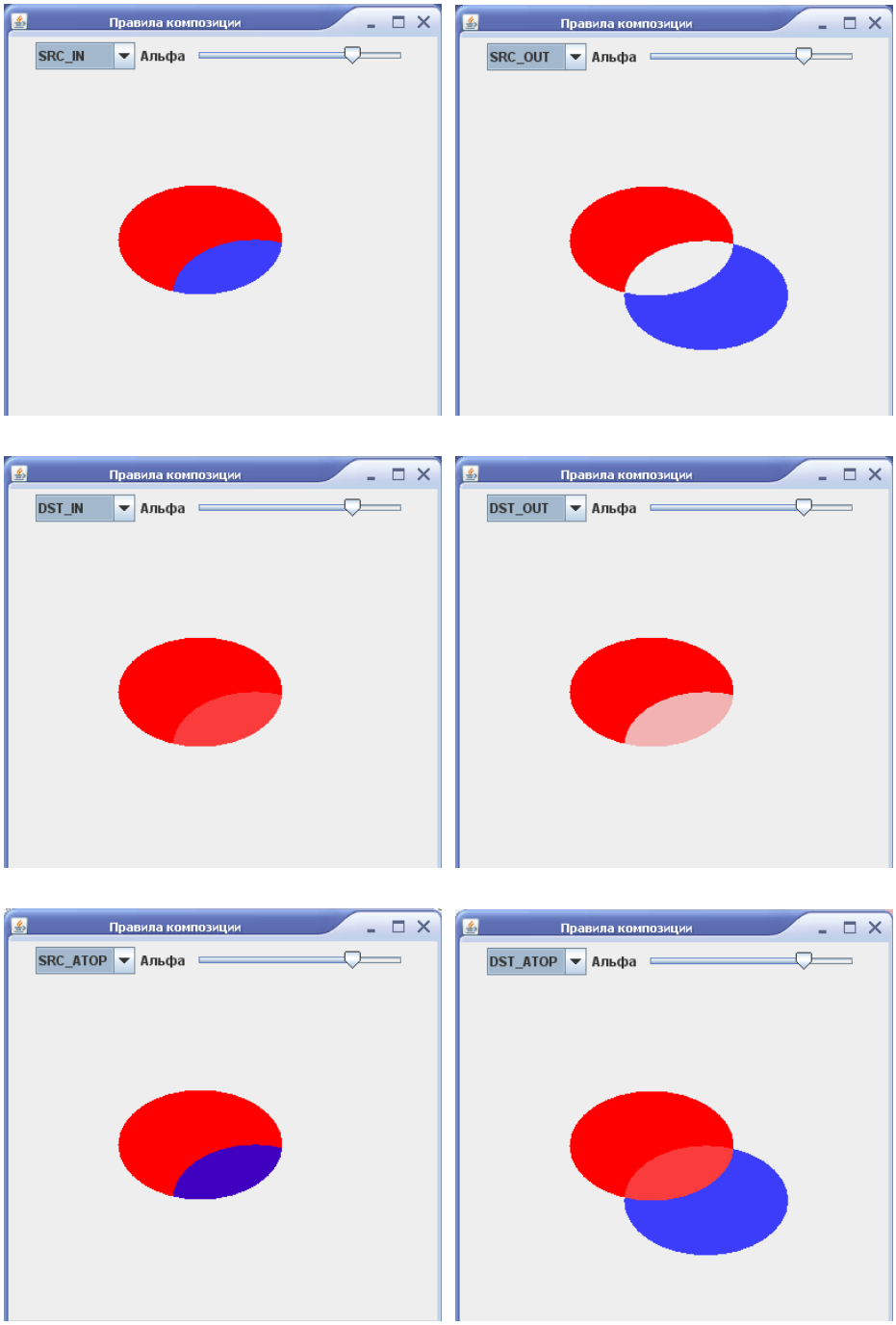


Рис. Работа программы, демонстрирующей правила композиции