

Лабораторная работа №1

Графические примитивы GDI

Цель работы: изучение графических 2D-примитивов с использованием GDI в среде Qt Creator.

Порядок выполнения работы

1. Изучить графические примитивы библиотеки Qt (<http://doc.qt.io/qt-4.8/QPainter.html>)
2. Разработать алгоритм и составить программу для построения на экране изображения в соответствии с номером варианта. В качестве исходных данных взять указанные в таблицы №1.

Требования к программе

1. В программе должна быть предусмотрена возможность ввода пользователем исходных данных (из правой колонки таблицы №1).
2. Изображение должно масштабироваться строго по центру с отступом 10 пикселей от границ и реагировать на изменение размера окна (см. пример проектов lab_1_qt_cpp, lab_1_qt_cpp_2, lab_1_vcpp).
3. Раскрасить (залить) примитивы (круги, многоугольники и др.) по собственному усмотрению.

Содержание отчёта

1. Название темы.
2. Цель работы.
3. Постановка задачи.
4. Вывод необходимых геометрических формул для построения изображения.
5. Текст программы.
6. Результат работы программы (снимки экрана).

Теоретические сведения

Qt Creator — бесплатная кросс-платформенная среда разработки (IDE), основанная на библиотеке QT и предназначена для редактирования, компиляции и отладки кода. Основной упор сделан на языки C/C++ и на разработку приложений на QT.

QPainter

QPainter содержит оптимизированные функции для выполнения большинства графических программ GUI. Он может рисовать все от простых линий до сложных фигур - секторов и дуг. QPainter может работать на любом объекте, который унаследован от класса QPaintDevice.

Методы

drawEllipse

Метод drawEllipse отрисовывает овал, вписанный в прямоугольник rectangle. Этот метод имеет 3 реализации. Первый в качестве параметра rectangle принимает прямоугольник с целочисленными координатами, второй с вещественными, третий принимает координаты верхнего левого угла прямоугольника, ширину и высоту прямоугольника.

Реализации:

```
void drawEllipse ( const QRectF & rectangle )
```

```
void drawEllipse ( const QRect & rectangle )
```

```
void drawEllipse ( int x, int y, int width, int height )
```

drawPie

Метод drawPie отрисовывает часть овала, словно его разрезали на куски. В методе должны быть заданы углы startAngle – начальный угол, и spanAngle – ширина угла. startAngle и spanAngle должны быть указаны в 1/16 степени, т. е. полный круг равен 5760 (16 * 360). Положительные значения для углов означают против часовой стрелки, тогда как отрицательные значения означают направление по часовой стрелке. Ноль градусов соответствует 3-часовой отметке.

```
void drawPie ( const QRectF & rectangle, int startAngle, int spanAngle )
```

```
void drawPie ( const QRect & rectangle, int startAngle, int spanAngle )
```

```
void drawPie ( int x, int y, int width, int height, int startAngle, int spanAngle )
```

Пример:

```
int startAngle = 30 * 16;
```

```
int spanAngle = 120 * 16;  
QPainter painter(this);  
painter.drawPie(rectangle, startAngle, spanAngle);
```

drawRect

Метод drawRect рисует прямоугольник с текущим ручкой и кистью.

Внутренний прямоугольник имеет размер rectangle.size (). Внешний прямоугольник имеет размер rectangle.size () плюс ширина пера.

```
void drawRect ( const QRectF & rectangle )  
void drawRect ( const QRect & rectangle )  
void drawRect ( int x, int y, int width, int height )
```

drawLine

Метод drawLine позволяет нарисовать прямую линию. В качестве параметра можно передать координаты двух точек, или экземпляр класса QLine (QLineF).

drawArc

```
void drawArc ( const QRectF & rectangle, int startAngle, int spanAngle )  
void drawArc ( const QRect & rectangle, int startAngle, int spanAngle )  
void drawArc ( int x, int y, int width, int height, int startAngle, int  
spanAngle )
```

Метод drawArc рисует дугу, заданную заданным прямоугольником, а так же startAngle и spanAngle (см. метод drawPie).

```
void drawLine ( const QLineF & line )  
void drawLine ( const QLine & line )  
void drawLine ( const QPoint & p1, const QPoint & p2 )  
void drawLine ( const QPointF & p1, const QPointF & p2 )  
void drawLine ( int x1, int y1, int x2, int y2 )
```

Класс QPen.

Класс QPen определяет, как [QPainter](#) должен рисовать линии и контуры фигур.

Перо имеет [style\(\)](#), [width\(\)](#), [brush\(\)](#), [capStyle\(\)](#) и [joinStyle\(\)](#).

Стиль пера определяет тип линии. Кисть используют для заполнения областей, созданных с помощью пера. Класс [QBrush](#) используется для определения стилей заполнения. Стиль окончаний (cap style) определяет,

как [QPainter](#) будет отрисовывать окончание линии, в то время как стиль соединения определяет, как линии соединяются при отрисовке. Толщина пера может быть установлена и как целое число ([width\(\)](#)), и как вещественное число ([widthF\(\)](#)). Линия с толщиной ноль определяет косметическое (cosmetic) перо. Это означает, что толщина пера всегда равна одному пикселю.

Различные параметры могут быть легко изменены с помощью функций [setStyle\(\)](#), [setWidth\(\)](#), [setBrush\(\)](#), [setCapStyle\(\)](#) и [setJoinStyle\(\)](#) (помните, что перо рисовальщика должно быть сброшено при установке других свойств пера).

Пример:

- `QPen pen();` // создаёт перо по умолчанию
- `QPen pen(Qt::green, 3, Qt::DashDotLine, Qt::RoundCap, Qt::RoundJoin);`

`painter.setPen(pen);`

По умолчанию перо является сплошным чёрным с толщиной 0, имеет прямоугольный стиль окончания ([Qt::SquareCap](#)), и тип соединений напрямую ([Qt::BevelJoin](#)).

В добавок к этому, `QPen` предоставляет вспомогательные функции [color\(\)](#) и [setColor\(\)](#) для получения и установки цвета кисти пера.

Стиль пера

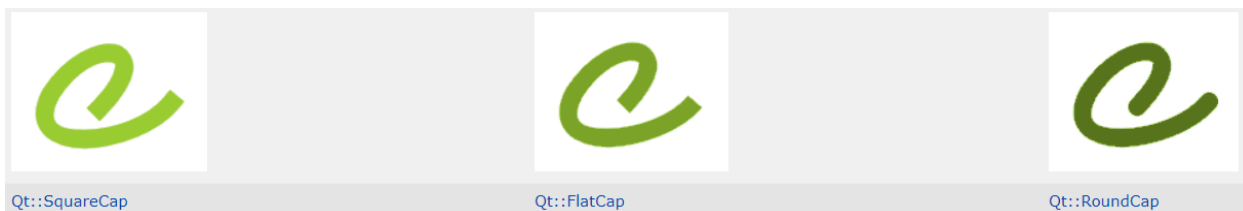
Qt обеспечивает несколько встроенных стилей, представленных в перечислении [Qt::PenStyle](#):



По умолчанию стиль установлен в [Qt::SolidLine](#).

Стиль окончаний

Стиль окончаний определяет, как будут отрисовываться окончания линий, используя [QPainter](#). Стиль окончаний применяется только при толщине 1 или более. Перечисление [Qt::PenCapStyle](#) представляет следующие стили:



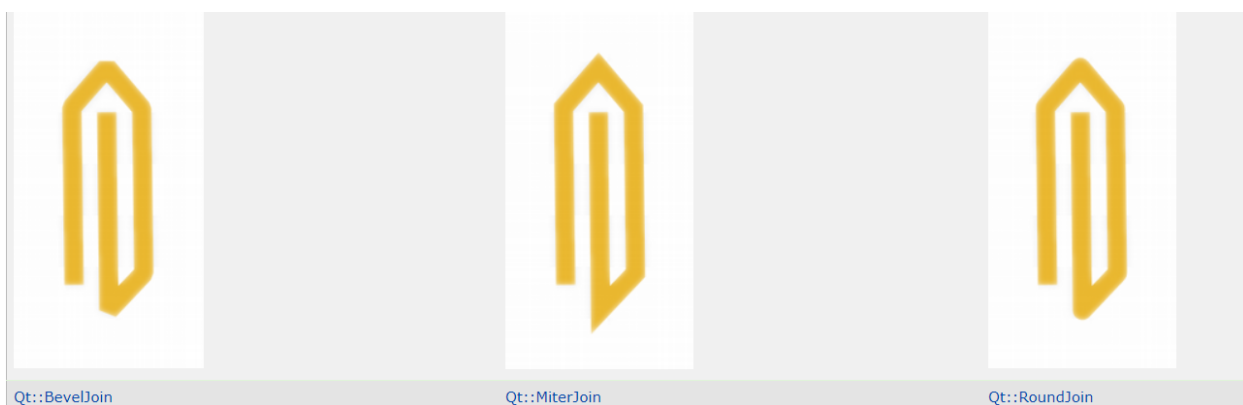
Стиль [Qt::SquareCap](#) определяет квадратное окончание и закрывает конечную точку другой линии на половину от ширины.

Стиль [Qt::FlatCap](#) определяет квадратное окончание и не закрывает конечную точку другой линии. А [Qt::RoundCap](#) определяет скруглённое окончание.

По умолчанию стоит [Qt::SquareCap](#).

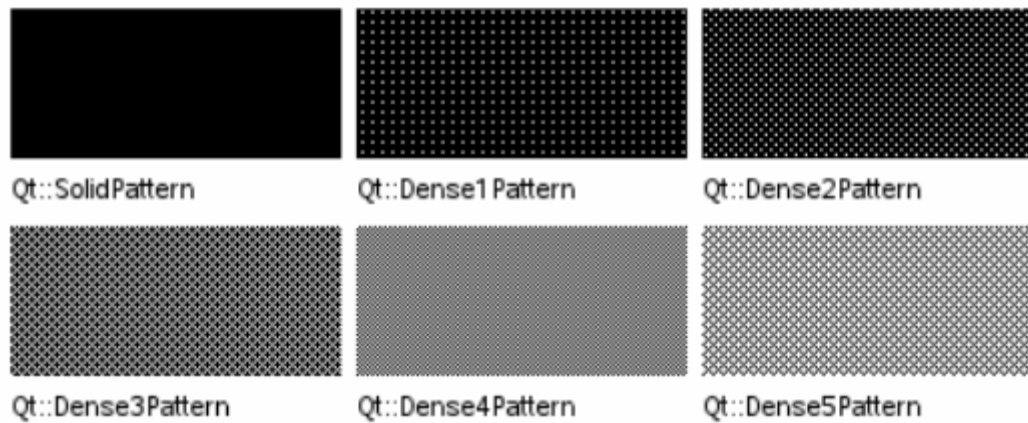
Стиль соединений

Стиль соединений определяет, как будет происходить соединение двух линий, используя [QPainter](#). Стиль соединений применяется только при толщине 1 или более. Перечисление [Qt::PenJoinStyle](#) представляет следующие стили:



Класс QBrush.

Класс QBrush задает образец заполнения фигур, рисуемых с помощью [QPainter](#). Стиль кисти по умолчанию - [Qt::NoBrush](#) (в зависимости от того, как вы создаете кисть). Данный стиль указывает, что фигуры не нужно заполнять. Стандартный стиль заполнения - [Qt::SolidPattern](#). Стиль может быть задан, когда кисть создается с помощью подходящего конструктора, дополнительно функция [setStyle\(\)](#) предоставляет средство для изменения стиля, когда кисть уже сконструирована.



Экземпляр QBrush можно получить с помощью следующих конструкторов.

`QBrush::QBrush ()`

Создает кисть по умолчанию: черная кисть со стилем `Qt::NoBrush` (такая кисть не заполняет фигуры).

`QBrush::QBrush (Qt::BrushStyle style)`

Создает черную кисть со стилем `style`.

`QBrush::QBrush (const QColor & color, Qt::BrushStyle style = Qt::SolidPattern)`

Создает кисть с полученными цветом `color` и стилем `style`.

Смотрите также `setColor()` и `setStyle()`.

`QBrush::QBrush (Qt::GlobalColor color, Qt::BrushStyle style = Qt::SolidPattern)`

Создает кисть с полученными цветом `color` и стилем `style`.

fillRect

Метод `fillRect` заполняет прямоугольник указанной кистью, или цветом и стилем, имеет несколько реализаций.

`void fillRect (const QRectF & rectangle, const QBrush & brush)`

`void fillRect (int x, int y, int width, int height, Qt::BrushStyle style)`

`void fillRect (const QRect & rectangle, Qt::BrushStyle style)`

`void fillRect (const QRectF & rectangle, Qt::BrushStyle style)`

`void fillRect (const QRect & rectangle, const QBrush & brush)`

`void fillRect (const QRect & rectangle, const QColor & color)`

Работа с цветом

QColor

Класс QColor предоставляет цвета, основанные на RGB или HSV моделях.

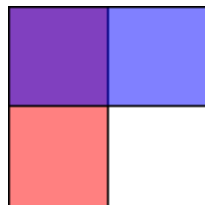
Цвет обычно задается в терминах RGB (красный, зеленый и синий) компонентов, но также может быть задан в HSV (оттенок, насыщенность и яркость) или быть установлен по имени цвета (название цвета может быть любым из имен цветов SVG 1.0).

QColor также поддерживает задание прозрачности с помощью альфа-канала. Значение альфа-компонента, равное 0, означает полностью прозрачный цвет, в то время как значение, равное 255 означает полностью непрозрачный цвет. Например:

```
// Задается полупрозрачный красный цвет  
painter.setBrush(QColor(255, 0, 0, 127));  
painter.drawRect(0, 0, width()/2, height());
```

```
// Задается полупрозрачный синий цвет  
painter.setBrush(QColor(0, 0, 255, 127));  
painter.drawRect(0, 0, width(), height()/2);
```

Вышеприведенный пример приводит к выводу следующего изображения:



Альфа-смешивание поддерживается Windows, Mac OS X и системами X11, в которых установлено расширение X Render.

Значение альфа-компоненты цвета может быть получено и установлено с помощью функций [alpha\(\)](#) и [setAlpha\(\)](#), если это значение типа integer, и с помощью функций [alphaF\(\)](#) и [setAlphaF\(\)](#), если это значение типа qreal (double).

Правильность QColor ([isValid\(\)](#)) указывает, является ли его значение корректным. Например, цвет RGB со значениями RGB, выходящими из диапазона, некорректен. По разумным соображениям QColor, главным образом, игнорирует некорректные цвета. Поэтому, результат использования некорректного цвета неопределен.

Есть 20 predefined objects
QColor: [Qt::white](#), [Qt::black](#), [Qt::red](#), [Qt::darkRed](#), [Qt::green](#), [Qt::darkGreen](#), [Qt::blue](#), [Qt::darkBlue](#), [Qt::cyan](#), [Qt::darkCyan](#), [Qt::magenta](#), [Qt::darkMagenta](#), [Qt::yellow](#), [Qt::darkYellow](#), [Qt::gray](#), [Qt::darkGray](#), [Qt::lightGray](#), [Qt::color0](#), [Qt::color1](#) и [Qt::transparent](#).



Цвета [Qt::color0](#) (нулевой пиксель) и [Qt::color1](#) (не-нулевой пиксель) - это специальные цвета для рисования на QBitmaps. Рисование цветом [Qt::color0](#) устанавливает биты битовой карты в 0 (прозрачным, т.е. цветом фона), а рисование цветом [Qt::color1](#) устанавливает биты в 1 (непрозрачным, т.е. цветом рисунка).

QColor платформенно и аппаратно независим. [QColorMap](#) - Класс-карта цветов для аппаратных средств.

Цвет можно задать передав в [setNamedColor\(\)](#) строку RGB (например "#112233") или имя цвета (например "blue"). Имена соответствуют именам цветов в SVG 1.0. Получить более светлый или более темный цвет можно с помощью [light\(\)](#) и [dark\(\)](#) соответственно. Цвета также могут быть заданы с помощью [setRgb\(\)](#) и [setHsv\(\)](#). К цветовым компонентам можно обращаться как к комплексному значению цвета, с помощью [rgb\(\)](#) и [hsv\(\)](#), так и индивидуально, с помощью [red\(\)](#), [green\(\)](#) и [blue\(\)](#).

Цвета HSV

Модель RGB является аппаратно-ориентированной. Ее представление близко к тому, что отображают большинство мониторов. HSV, напротив, представляет цвет способом, больше подходящим для человеческого восприятия цвета. Например, отношения "насыщенной чем", "темнее чем" и

"противоположный" легко выразимы в модели HSV, но их тяжело выразить в модели RGB.

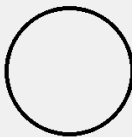


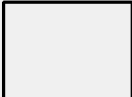



HSV, подобно RGB, имеет три компонента:

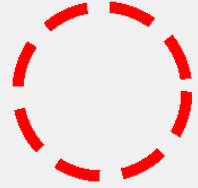
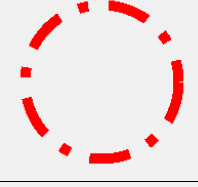
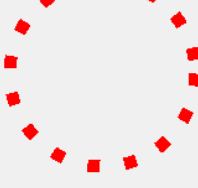
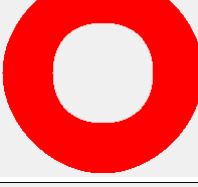

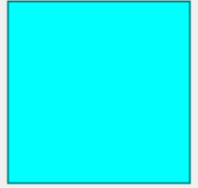
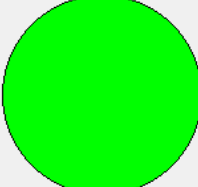
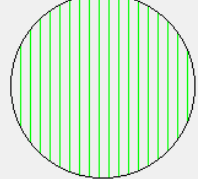
- H определяет оттенок, принимает значения из диапазона от 0 до 359, если цвет цветной (не серый), или не имеет смысла, если цвет серый. Он выражает градусы на цветовом круге, знакомом большинству людей. Красный цвет обозначается 0 (градусов), зеленый - 120, а голубой - 240.
- S определяет насыщенность, принимает значения из диапазона от 0 до 255, и чем оно больше, тем более насыщенный цвет. Сероватые цвета имеют насыщенность, близкую к 0; очень насыщенные цвета имеют насыщенность, близкую к 255.
- V определяет яркость, принимает значения из диапазона от 0 до 255. 0 соответствует черному цвету; 255 соответствует цвету, насколько это возможно, удаленному от черного.

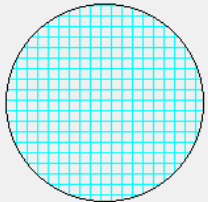

Приведем несколько примеров: чисто-красный цвет - $H=0$, $S=255$, $V=255$; темно красный, переходящий в сиреневый, может быть выражен $H=350$ (эквивалентно -10), $S=255$, $V=180$; сероватый светлокрасный цвет может иметь H, близкий к 0 (350-359 или 0-10), S приблизительно равный 50-100, а $V=255$.

Для бесцветных цветов (серых) Qt возвращает значение оттенка, равное -1. Если Вы передаете значение слишком большое значение оттенка, Qt приводит его к значению, соответствующему диапазону. Насыщенность, равная 360 или 720, толкуется как 0; насыщенность, равная 540, толкуется как 180.

Примеры вызова функций в Qt Creator

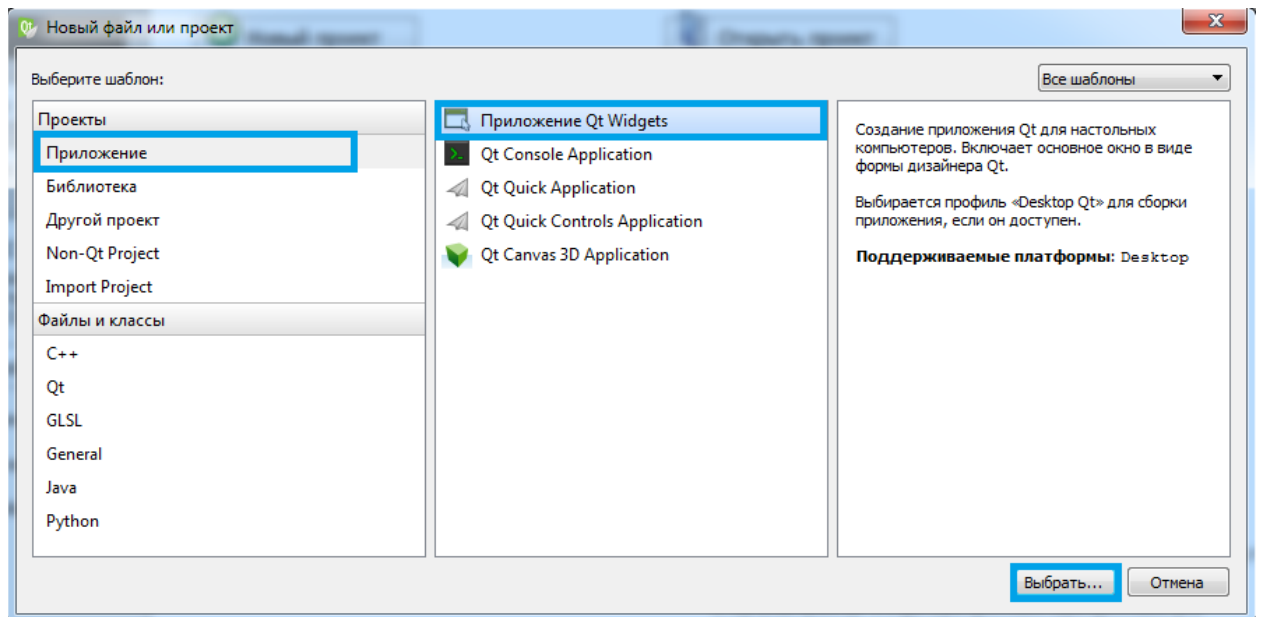
Вызов функции	Графический вид
<pre>painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawEllipse(100, 50, 150, 150);</pre>	
<pre>QRect rect = QRect(80, 80, 150, 150); painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawPie(rect, 0, (30*180)/3.1416);</pre>	
<pre>QRect rect = QRect(80, 80, 150, 150); painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawRect(rect);</pre>	
<pre>QRect rect = QRect(80, 80, 200, 150); painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawRect(rect);</pre>	
<pre>painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawEllipse(100, 50, 200, 100);</pre>	
<pre>QPoint p1 = QPoint(width() / 2, 10); QPoint p2 = QPoint(width() / 2, height() - 10); painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawLine(p1, p2);</pre>	
<pre>painter.setPen(QPen(Qt::black, 5, Qt::SolidLine, Qt::FlatCap)); painter.drawArc(100, 100, 150, 150, 0, (30*180)/3.1416);</pre>	
<p>1й вариант: QPen pen(); // создаёт перо по умолчанию pen.setStyle(Qt::DashDotLine); pen.setWidth(10); pen.setBrush(Qt::red); pen.setCapStyle(Qt::SolidLine); pen.setJoinStyle(Qt::RoundCap); painter.setPen(pen); painter.drawArc(100, 100, 150, 150, 0, (30*180)/3.1416);</p> <p>2й вариант: painter.setPen(QPen(Qt::red, 10, Qt::SolidLine, Qt::RoundCap)); painter.drawArc(100, 100, 150, 150, 0, (30*180)/3.1416);</p>	

<pre>painter.setPen(QPen(Qt::red, 10, Qt::DashLine, Qt::FlatCap)); painter.drawEllipse(100, 50, 150, 150);</pre>		
<pre>pen.setStyle(Qt::DashDotLine); painter.setPen(pen) painter.drawEllipse(100, 50, 150, 150);</pre>		
<pre>pen.setStyle(Qt::DotLine); painter.setPen(pen) painter.drawEllipse(100, 50, 150, 150);</pre>		
<pre>pen.setWidth(50); painter.setPen(pen) painter.drawEllipse(100, 50, 150, 150);</pre>		
<pre>pen.setCapStyle(Qt::SquareCap); painter.setPen(pen) painter.drawArc(100,100,150,150,0, (30*180)/3.1416);</pre>		
<pre>QPainter painter(this); painter.setBrush(Qt::cyan); painter.setPen(Qt::darkCyan); painter.drawRect(0, 0, 100,100);</pre>		
<pre>painter.setBrush(Qt::green); painter.drawEllipse(100, 50, 150, 150);</pre>		
<pre>painter.setBrush(QBrush(Qt::green,Qt::VerPattern)); painter.drawEllipse(100, 50, 150, 150);</pre>		

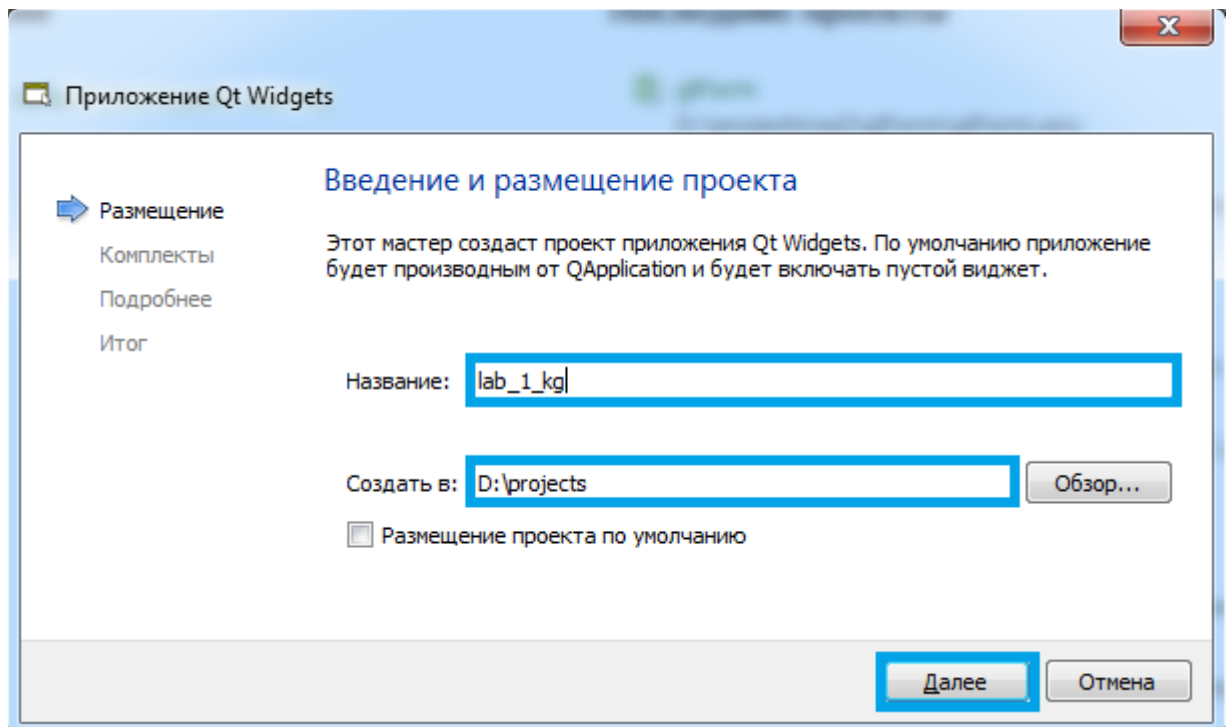
<pre>painter.setBrush(QBrush(Qt::cyan,Qt::CrossPattern)); painter.drawEllipse(100, 50, 150, 150);</pre>	
<pre>QLinearGradient gradient(0, 0, 300, 350); gradient.setColorAt(0,Qt::red); gradient.setColorAt(0.5, Qt::green); gradient.setColorAt(1, Qt::blue); painter.fillRect(10,10,100,100,gradient);</pre>	

Создание графического приложения в Qt Creator

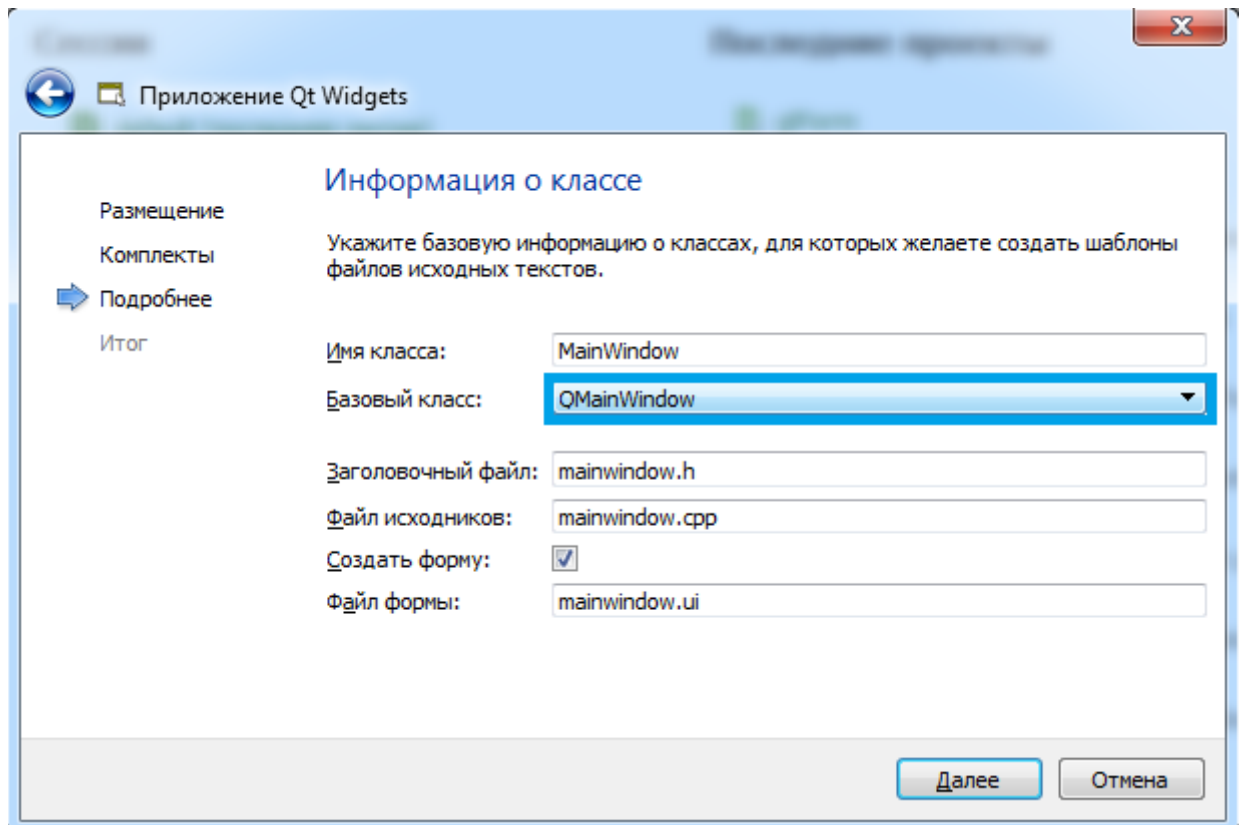
1. Запустить окно для создания нового проекта (**Ctrl+N**):



2. Выбрать имя проекта и каталог для его размещения:



3. Выбрать базовый класс для создания приложения (QMainWindow или QWidget)



4. В классе MainWindow необходимо переопределить виртуальный метод paintEvent:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

    // Обработчик события перерисовки окна
    void paintEvent(QPaintEvent* event);
};

#endif // MAINWINDOW_H
```

5. Реализовать в файле исходного кода (*.cpp) обработчик paintEvent:

```
// Функция для рисования круга
void MainWindow::paintEvent(QPaintEvent* event)
{
    QPainter painter(this); // Создаём объект-живописец

    QColor orangeRed(240, 100, 0); // Оранжево-красный цвет

    // Радиус большой окружности
    float radius;
    // Вычисляем радиус окружности
    if (width() > height()) // Если ширина окна больше высоты
        radius = (height() - 20) / 2; // Отступ от краёв - 10 пикселей
    else radius = (width() - 20) / 2;

    // Если размеры окна маленькие, ничего не выводить
    if (width() < 30 || height() < 30)
        return;

    // Координаты центра окружности
    QPointF center = QPointF(width() / 2, height() / 2);

    // Задаём область прорисовки круга
    QRectF rect = QRectF(center.x() - radius, center.y() - radius, radius*2,
radius*2);

    // Рисуем большой круг красно-оранжевым цветом
    painter.setBrush( orangeRed);
    painter.setPen( orangeRed );
    painter.drawEllipse(rect);
}
```

6. Результат запуска программы:

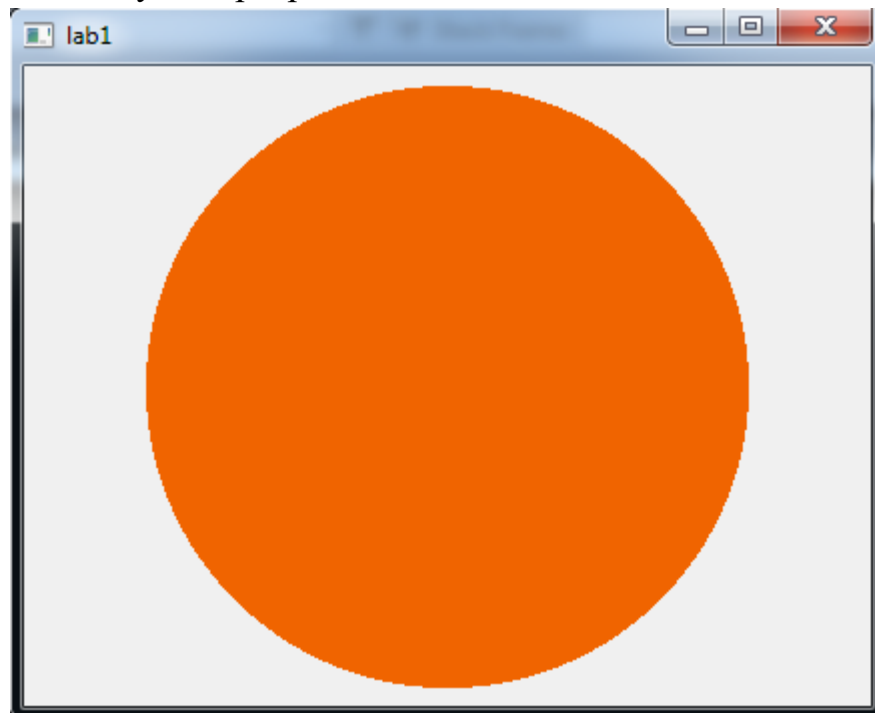
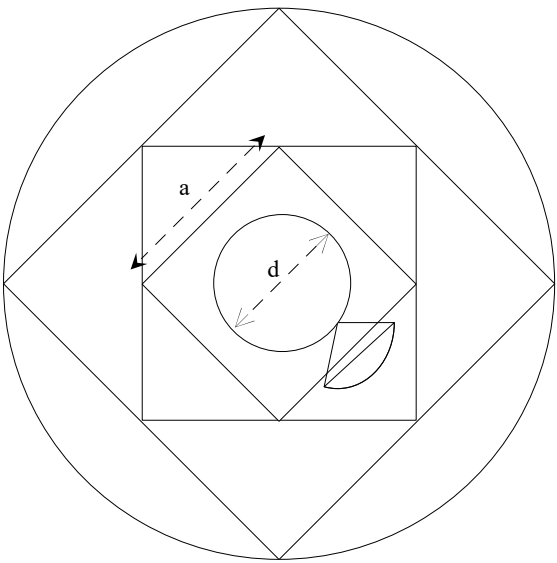
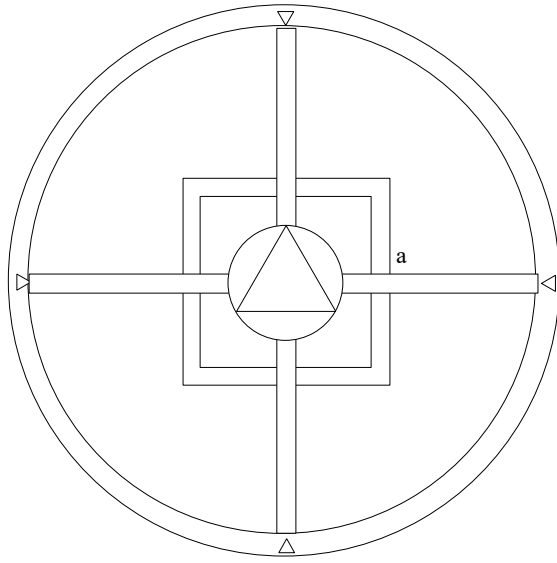
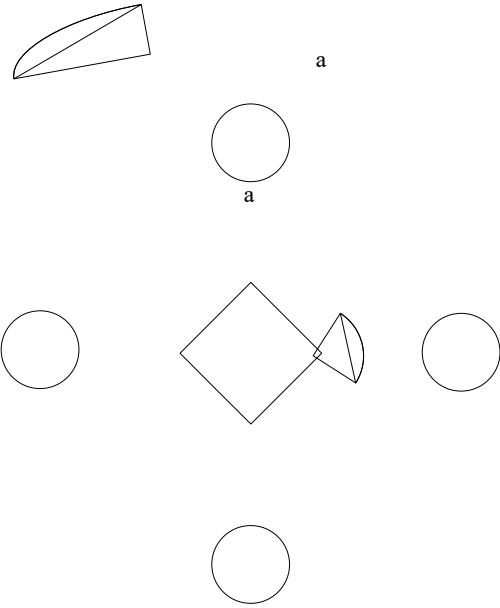
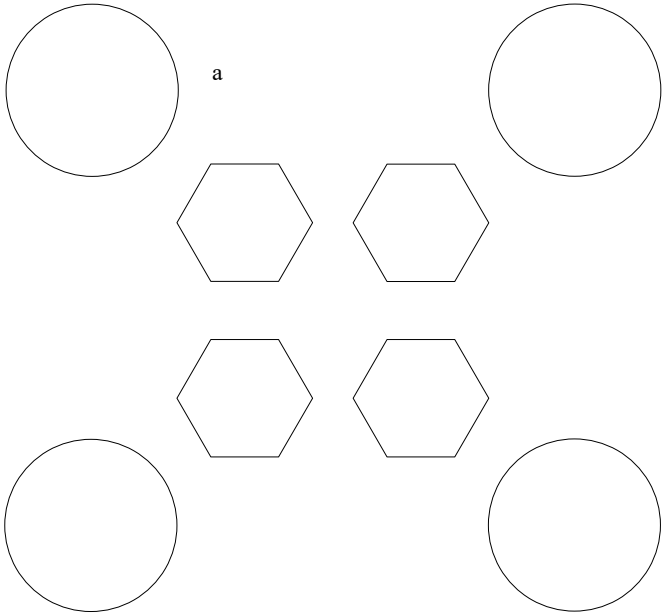
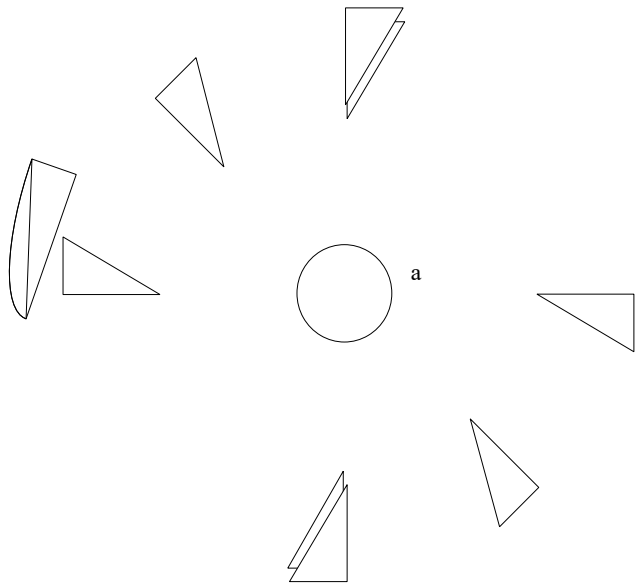
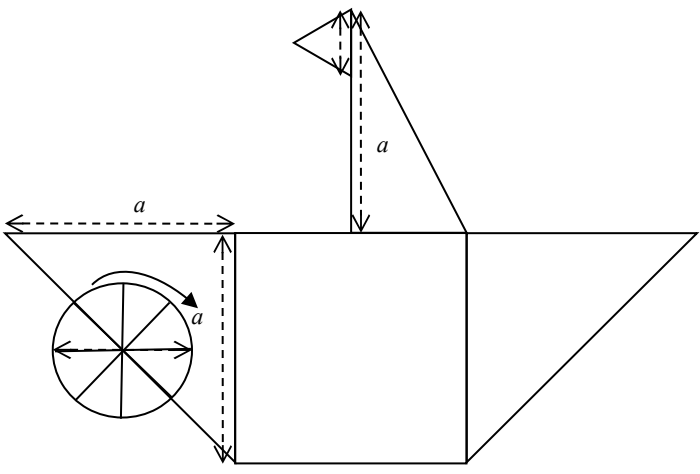


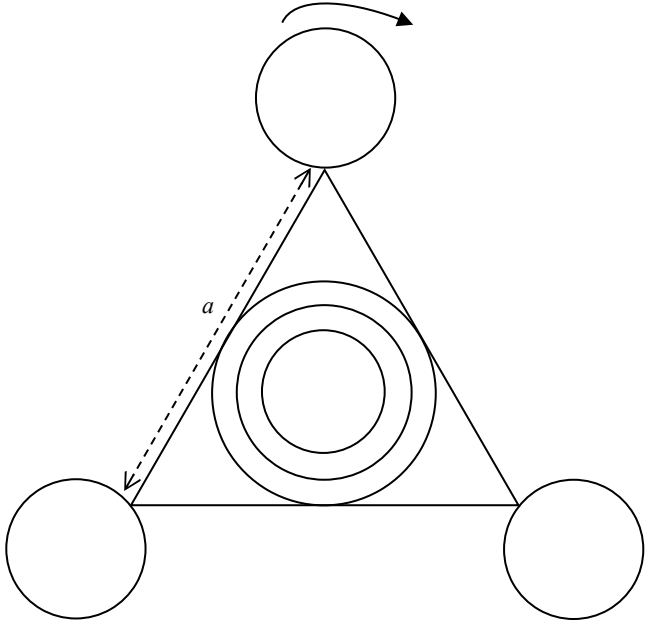
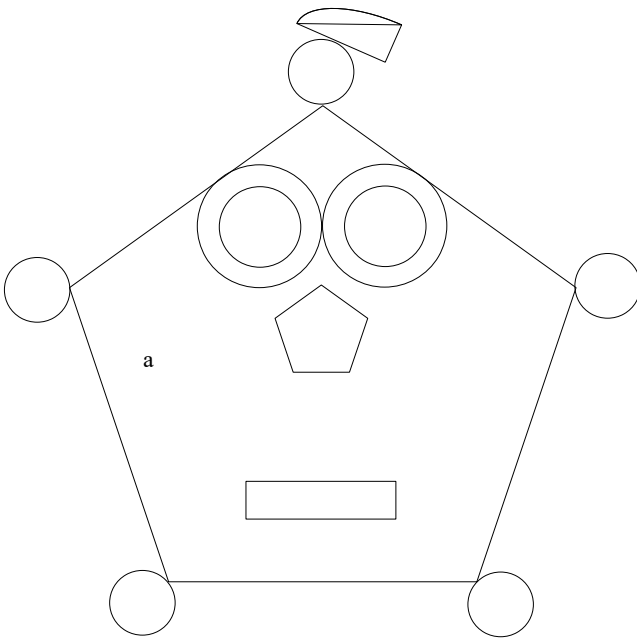
Таблица №1

Вариант	Рисунок	Исходные данные
1		$0 < \frac{d}{a} < 1$ Отношение $\frac{d}{a}$ вводится пользователем. Реализовать вращение внутреннего квадрата против часовой стрелки. Раскрасить все элементы по собственному усмотрению.
2		a – сторона большого квадрата, n – число сторон центрального многоугольника. a и n вводятся с клавиатуры. Раскрасить все элементы по своему усмотрению.

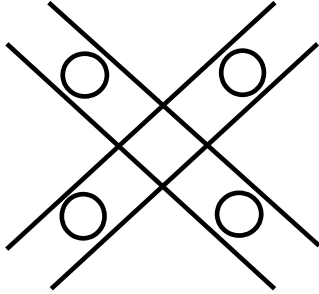
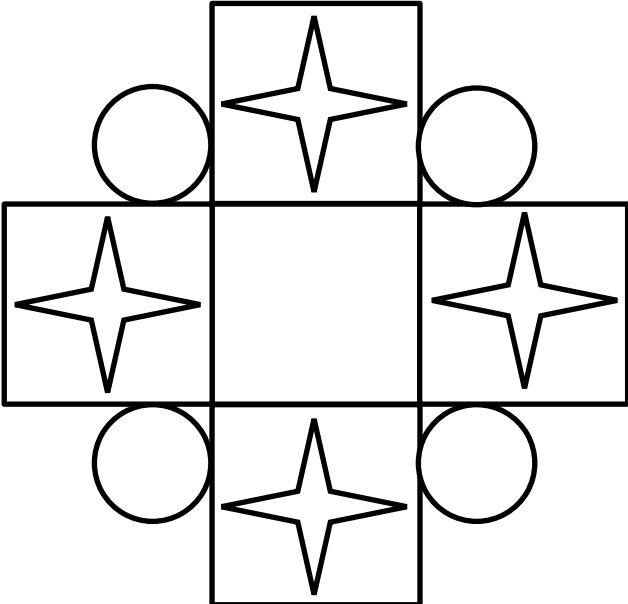
3		$0 < \frac{r_1 + r_3}{r_2} < 1$ <p>Отношение $\frac{r_1}{r_2}$ и a вводятся с клавиатуры. Реализовать вращение внутренней фигуры по часовой стрелке. Раскрасить все элементы по своему усмотрению.</p>
4		<p>n – количество сторон многоугольника, вводится с клавиатуры. Для n четного – количество треугольников и окружностей одинаково, для нечетного – окружностей больше. Раскрасить все элементы по своему усмотрению.</p>

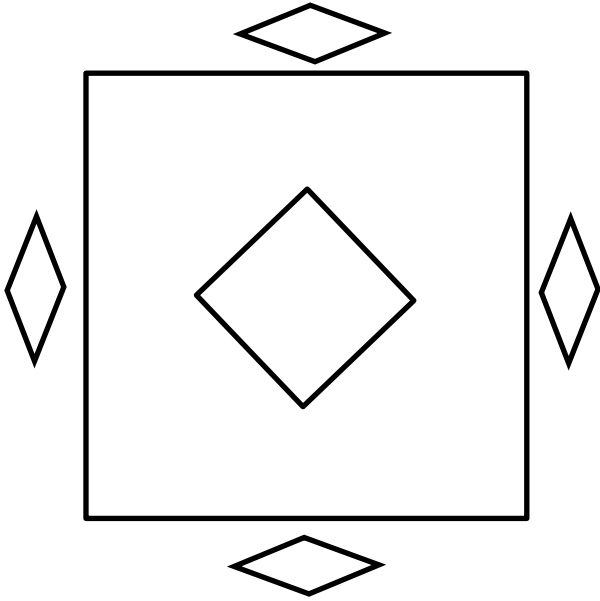
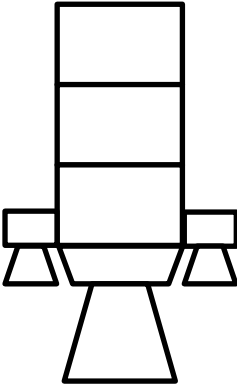
5		<p>a – сторона квадрата, вводится с клавиатуры. Вся фигура должна вращаться по часовой стрелке, центральный квадрат – против. Раскрасить все элементы по своему усмотрению.</p>
6		<p>a – сторона квадрата, n – количество сторон многоугольника. Реализовать движение окружностей параллельно сторонам квадрата по часовой стрелке. Раскрасить все элементы по своему усмотрению. *Центральные точки многоугольников соответствуют центральным точкам треугольников</p>

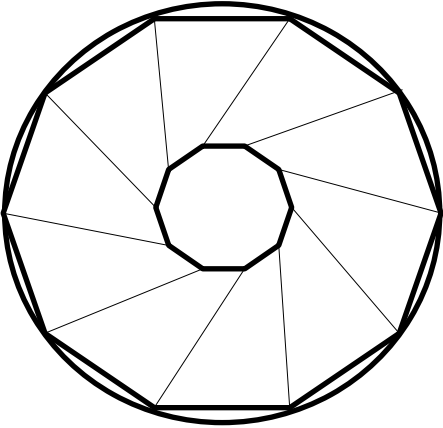
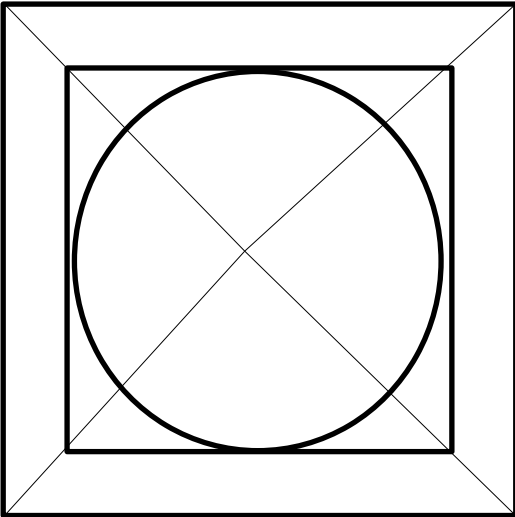
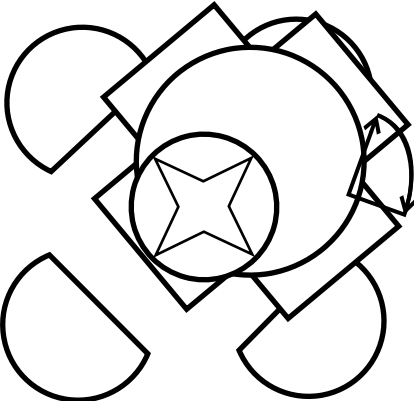
7		<p>a – диаметр внутренней окружности, n – количество сторон многоугольника. a и n вводятся с клавиатуры. Реализовать вращение крыльев мельницы по часовой стрелке. Раскрасить все элементы по своему усмотрению.</p>
8		<p>a – сторона треугольника, n – количество секторов в окружности. a и n вводятся с клавиатуры. Реализовать вращение окружности по часовой стрелке. Раскрасить каждый элемент сектора своим цветом.</p>

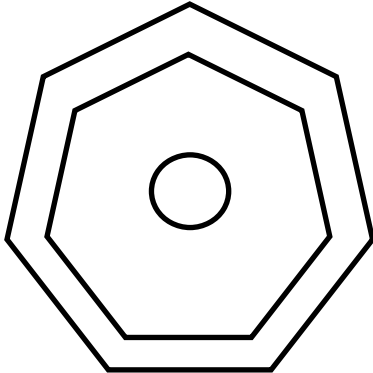
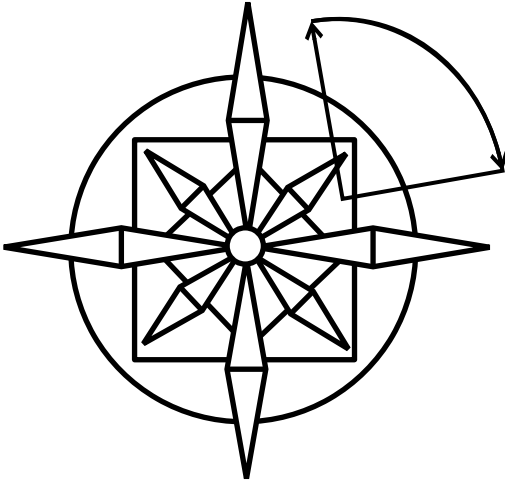
9		<p>a – сторона треугольника, n – количество сторон многоугольника, вписанных в окружности. n вводится с клавиатуры. Реализовать вращение всей фигуры по часовой стрелке. Раскрасить все элементы по своему усмотрению.</p>
10		<p>Реализовать движение внешних окружностей по часовой стрелке. Раскрасить все элементы по своему усмотрению.</p>

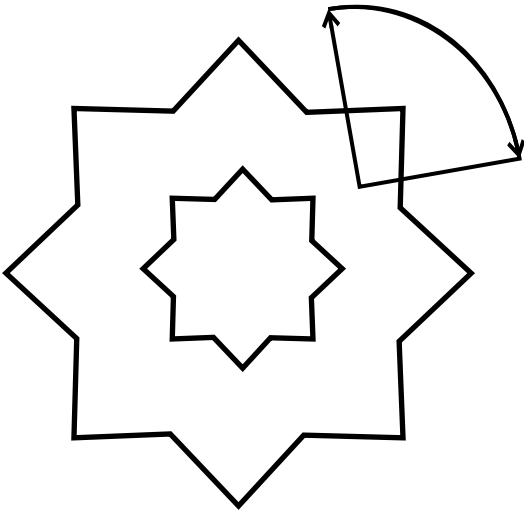
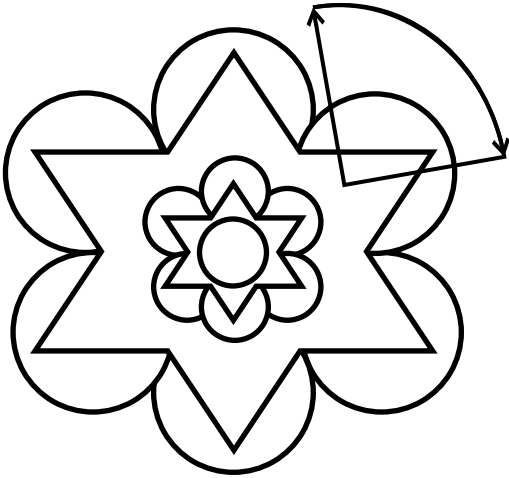
11		<p>- a – расстояние между шестиугольником и примитивами</p>
12		<p>Реализовать вращение фигуры. n-количество лучиков Ширина и высота лучиков случайная</p>
13		<p>первый пояс заполнить равномерно по всему кругу квадратами, второй - треугольниками, третий – окружностями n – количество примитивов внутри каждого пояса Реализовать вращение треугольников по кругу.</p>

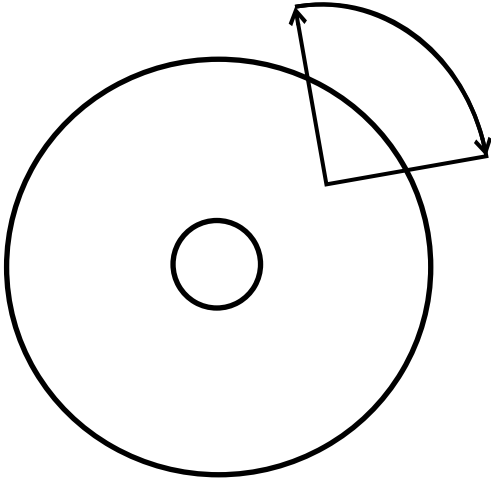
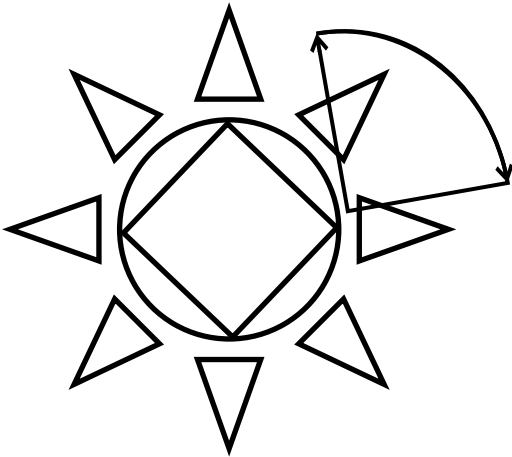
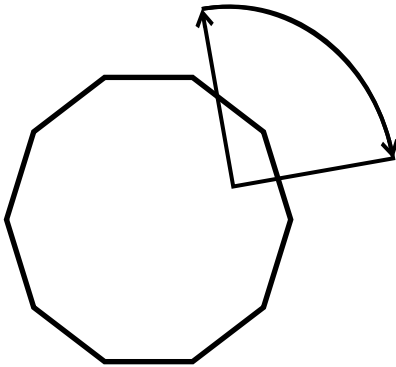
14		<p>n – количество случайных примитивов в каждом прямоугольнике Реализовать вращение закрытого квадрата.</p>
15		<p>Реализовать вращение центрального квадрата по часовой стрелке.</p>

16		<p>Реализовать вращение всей фигуры.</p>
17		<p><i>n- количество отсеков ракеты(квадрат с кругом)</i></p>

18		<i>Вращение фигуры</i>
19		<i>Вращать треугольник</i>
20		<p>Реализовать вращение фигуры по часовой стрелке, центральные круги и звезда остаются неподвижными.</p>

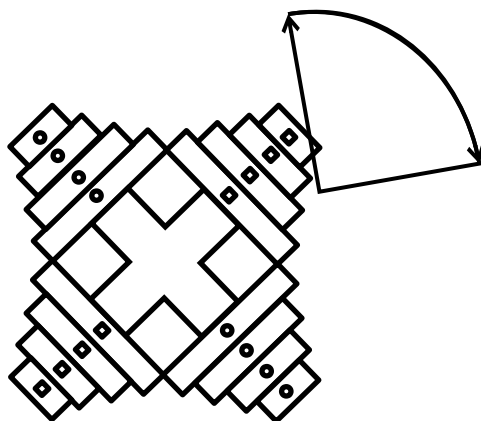
21		<p>n – количество вершин многоугольника и треугольников</p>
22		<p>Реализовать вращение 4-х больших стрелок.</p>

23		<p>Реализовать вращение маленькой восьмиконечной звезды.</p>
24		<p>Реализовать вращение большой шестиконечной звезды. Круги могут накладываться друг на друга.</p>

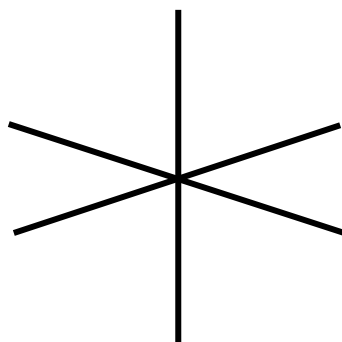
25		<p>Реализовать вращение фигуры.</p>
26		<p>a – расстояние между примитивами (треугольниками и окружностью), n – количество треугольников. Реализовать вращение.</p>
27		<p>a – расстояние между примитивами (треугольниками и многоугольником), , n – количество сторон центрального многоугольника Реализовать вращение внутреннего квадрата.</p>

28

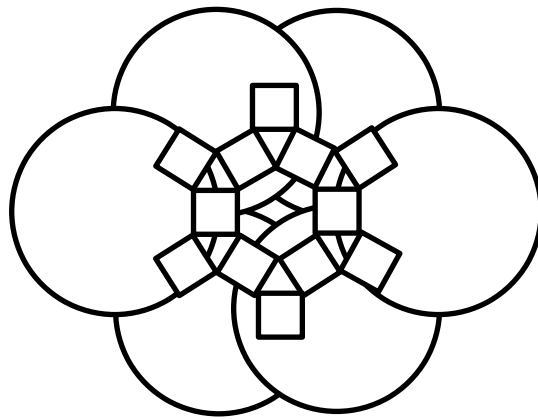
Реализовать
вращение фигуры.



29



30



31

