

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

А.Ю. Демин

КОМПЬЮТЕРНАЯ ГРАФИКА И ГЕОМЕТРИЯ

Лабораторный практикум

*Рекомендовано в качестве учебно-методического пособия
Редакционно-издательским советом
Томского политехнического университета*

Издательство
Томского политехнического университета
2024

УДК 004.92(076.5)

ББК 32.972.131.2я7

Д30

Демин А.Ю.

Д30 Компьютерная графика и геометрия. Лабораторный практикум : учебно-методическое пособие / А.Ю. Демин ; Томский политехнический университет. – Томск : Изд-во Томского политехнического университета, 2024. – 171 с.

ISBN 978-5-4387-1214-5

В пособии рассматриваются следующие темы: обработка растровых изображений в графическом редакторе GIMP, создание векторных изображений в редакторе Inkscape, разработка приложений для работы с графикой, 2D- и 3D-преобразования, получение проекций, графические библиотеки. Предусмотрены две образовательные траектории с изучением возможностей C# в Visual Studio и JavaScript в Visual Studio Code.

Предназначено для студентов, обучающихся по направлениям 090301 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 09.03.02 «Информационные системы и технологии».

УДК 004.92(076.5)

ББК 32.972.131.2я7

Рецензенты

Доктор физико-математических наук, профессор кафедры радиоэлектроники и систем связи радиотехнического факультета ТУСУРа
А.С. Задорин

Доктор физико-математических наук, доцент
заведующий кафедрой программной инженерии ИПМКН НИ ТГУ
А.Н. Моисеев

ISBN 978-5-4387-1214-5

© ФГАОУ ВО НИ ТПУ, 2024

© Демин А.Ю., 2024

© Оформление. Издательство Томского политехнического университета, 2024

Содержание

Введение	4
Лабораторная работа 1. Основы GIMP.....	6
Лабораторная работа 2. Фотомонтаж	19
Лабораторная работа 3. Обработка изображений	32
Лабораторная работа 4. Основы Inkscape.....	44
Лабораторная работа 5. Создание векторного логотипа	52
Лабораторная работа 6. Программирование графики.....	60
Лабораторная работа 7. Простейшая анимация.....	78
Лабораторная работа 8. Работа с растровыми изображениями	86
Лабораторная работа 9. Преобразования на плоскости.....	98
Лабораторная работа 10. 3D-преобразования и получение проекций	110
Лабораторная работа 11. Построение трехмерных сцен	119
Лабораторная работа 12. Трехмерные преобразования в WPF и ThreeJS	143
Заключение	169
Список литературы.....	170

ВВЕДЕНИЕ

С момента появления первых ЭВМ люди стремятся разнообразить способы общения человека и машины – приблизить к уровню общения человека с человеком. Это общение было бы гораздо более ограниченным, если бы при этом не использовался один из наиболее простых способов – язык изображений. Сегодня графические изображения на экране монитора современного персонального компьютера стали для людей нормой, совершенно неотъемлемым атрибутом интерфейса. Спектр применения компьютерной графики, помимо средства интерфейса «человек–машина», чрезвычайно широк: от создания рекламных роликов, компьютерных мультфильмов и игр, кроя одежды, малых и монументальных форм дизайна, компьютерной живописи до визуализации результатов научных изысканий [10].

Предметом данной работы является обширная область компьютерных наук, посвященная представлению данных в памяти ЭВМ в графической форме. Это самое общее определение, так как под данными можно понимать как непосредственно хранящееся в виде файла изображение в одном из графических форматов, так и протокол обмена командами между пользователем и ЭВМ (то, что называется графическим интерфейсом), а также битовую последовательность, сформированную для вывода на экран или печатающее устройство. Методы и способы представления и манипуляции этим видом данных относятся к сфере знаний компьютерной графики.

В пособии рассматриваются различные способы представления изображений в памяти ЭВМ, основы работы со распространяемым программным обеспечением с открытым исходным кодом, создание и обработка растровых и векторных изображений в редакторах GIMP и Inkscape. Подробно разбираются вопросы программирования для создания изображений, анимации, 2D- и 3D-преобразования, построения реалистичных трехмерных сцен с помощью популярных технологий и библиотек. Лабораторный практикум состоит из 12 разделов (лабораторных работ). Первые три лабораторные работы знакомят с растровым редактором GIMP, четвертая и пятая посвящены векторному редактору Inkscape. Следующие лабораторные работы рассматривают компьютерную графику и геометрию с точки зрения программирования. При этом в этих лабораторных работах реализована возможность выбора образовательной траектории. Первая траектория строится на базе использования языка C#, языка разметки XAML и технологии WPF. Вторая образовательная траектория позволяет выбрать в качестве основы выполнения тех

же самых заданий язык программирования JavaScript и библиотеку ThreeJS.

Первые четыре лабораторные работы имеют общее задание для всех обучающихся. Начиная с пятой лабораторной работы все задания индивидуальны.

Рекомендуется придерживаться следующего порядка выполнения каждой лабораторной работы:

1. **Запустите необходимое программное обеспечение:** GIMP – для лабораторных работ 1–3, Inkscape – для лабораторных работ 4–5, Visual Studio Community или VS Code – для всех остальных работ.

2. **Ознакомьтесь с разделом общие сведения.** Чаще всего в тексте теория сопровождается практическими примерами, поэтому рекомендуется опробовать эти практические примеры в соответствующем программных средствах.

3. **Проверьте себя,** отвечая на **контрольные вопросы.**

4. Начиная с шестой лабораторной работы **необходимо выбрать одну из двух образовательных траекторий** (C# или JavaScript) и выполнить лабораторные работы выбранными программными средствами. Для каждого образовательного трека в разделе **Общие сведения** содержатся своя теория и свои примеры. **Контрольные вопросы** в основном приводятся отдельно для обоих образовательных треков.

5. Не забывайте **составлять отчет о проделанной работе.** Вставляйте код программы и скриншоты из запущенной программы в отчёт, пишите к ним пояснения: в будущем это может помочь вспомнить необходимые детали работы.

6. После изучения **Общих сведений** и самопроверки **выберите индивидуальное задание** или попросите преподавателя выбрать его, если задание выдаётся им.

7. **Продемонстрируйте преподавателю выполненное индивидуальное задание,** ответьте на его вопросы и, при необходимости, внесите исправления.

8. **Подготовьте отчет** в электронном виде в соответствии со стандартом образовательной организации и отправьте его на проверку.

Лабораторная работа 1

ОСНОВЫ GIMP

Цель: изучить основы растрового редактора GIMP по созданию и импорту растровых изображений.

Общие сведения

GIMP – многоплатформенное программное обеспечение с открытым исходным кодом, предназначенное для редактирования растровых изображений (GIMP – GNU Image Manipulation Program). Редактор GIMP пригоден для решения множества задач по изменению изображений, включая ретушь фотографий, объединение и создание изображений.

GIMP входит в состав большинства дистрибутивов GNU/Linux. Также доступен и для других операционных систем, таких как Windows или Mac OS. GIMP является свободным программным обеспечением, выпускаемым под лицензией GPL (General Public License). GPL предоставляет пользователям право доступа к исходному коду программ и право изменять его.

Будучи весьма мощным продуктом, GIMP способен стать незаменимым помощником в подготовке графики для web-страниц и полиграфической продукции, оформлении программ (рисование пиктограмм, заставок и т. п.), создании анимационных роликов, обработке кадров для видеофрагментов и построении текстур для трехмерной анимации.

Основные термины GIMP

Изображения. Изображение – основной объект, с которым работает GIMP. Под словом «изображение» подразумевается один файл с расширением TIFF или JPEG. Можно отождествлять изображение и окно, которое его содержит, но это будет не совсем правильно: можно открыть несколько окон с одним и тем же изображением. В то же время нельзя открыть в одном окне более одного изображения и нельзя работать с изображением без отображающего его окна.

Изображение в GIMP может быть достаточно сложным. Наиболее правильной аналогией будет не лист бумаги, а, скорее, книга, страницы которой называются слоями.

Слой. Если изображение подобно книге, то слой можно сравнить со страницей внутри книги. Простейшее изображение содержит только один слой. Слои могут быть прозрачными и могут покрывать не все пространство изображения.

Каналы. В GIMP каналы – наименьшая единица подразделения стека слоев, из которых создается изображение. Каждый канал имеет тот же размер, что и слой, и состоит из тех же пикселей. Смысл этого значения зависит от типа канала. Например, в цветовой модели RGB значение канала R означает количество красного цвета, добавляемого к другим цветам пикселей.

Выделения. Часто при работе возникает необходимость изменить только часть изображения. Для этого существует механизм выделения областей. В каждом изображении можно создать выделенную область, которая, как правило, отображается в виде движущейся пунктирной линии.

История правки. Ошибки при редактировании изображений неизбежны, однако почти всегда возможно отменить свои действия: GIMP записывает историю действий пользователя, позволяя при необходимости вернуться на несколько шагов назад. Однако история занимает память, поэтому возможности отмены не безграничны.

Основные приемы использования GIMP

Для того чтобы открыть программу GIMP в Windows, нужно нажать на кнопку пуск и выбрать в меню GIMP пункт GIMP.

На рис. 1.1 показано стандартное расположение окон GIMP. Элементами GIMP являются:

1. Панель инструментов, которая содержит кнопки для выбора инструментов выделения, рисования, трансформации изображения и т. д. Также по умолчанию здесь доступны цвета фона и переднего плана.

2. Параметры инструментов: под панелью инструментов прикреплен диалог **Параметры инструментов**, который отображает параметры выбранного инструмента (в данном случае это инструмент «Прямоугольное выделение»).

3. Окно изображения. Каждое изображение в GIMP отображается в отдельном окне. Можно открыть одновременно достаточно большое количество изображений: столько, сколько позволяют системные ресурсы.

4. Диалоги **Слои/Каналы/Контур/Отменить**, которые отображают структуру слоев активного изображения, позволяют управлять ими, работать с каналами и контурами, а также отменять операции.

5. Диалоги **Кисти/Текстуры/Градиенты**: панель, расположенная ниже диалога слоев, показывает диалоги управления кистями, текстурами и градиентами.

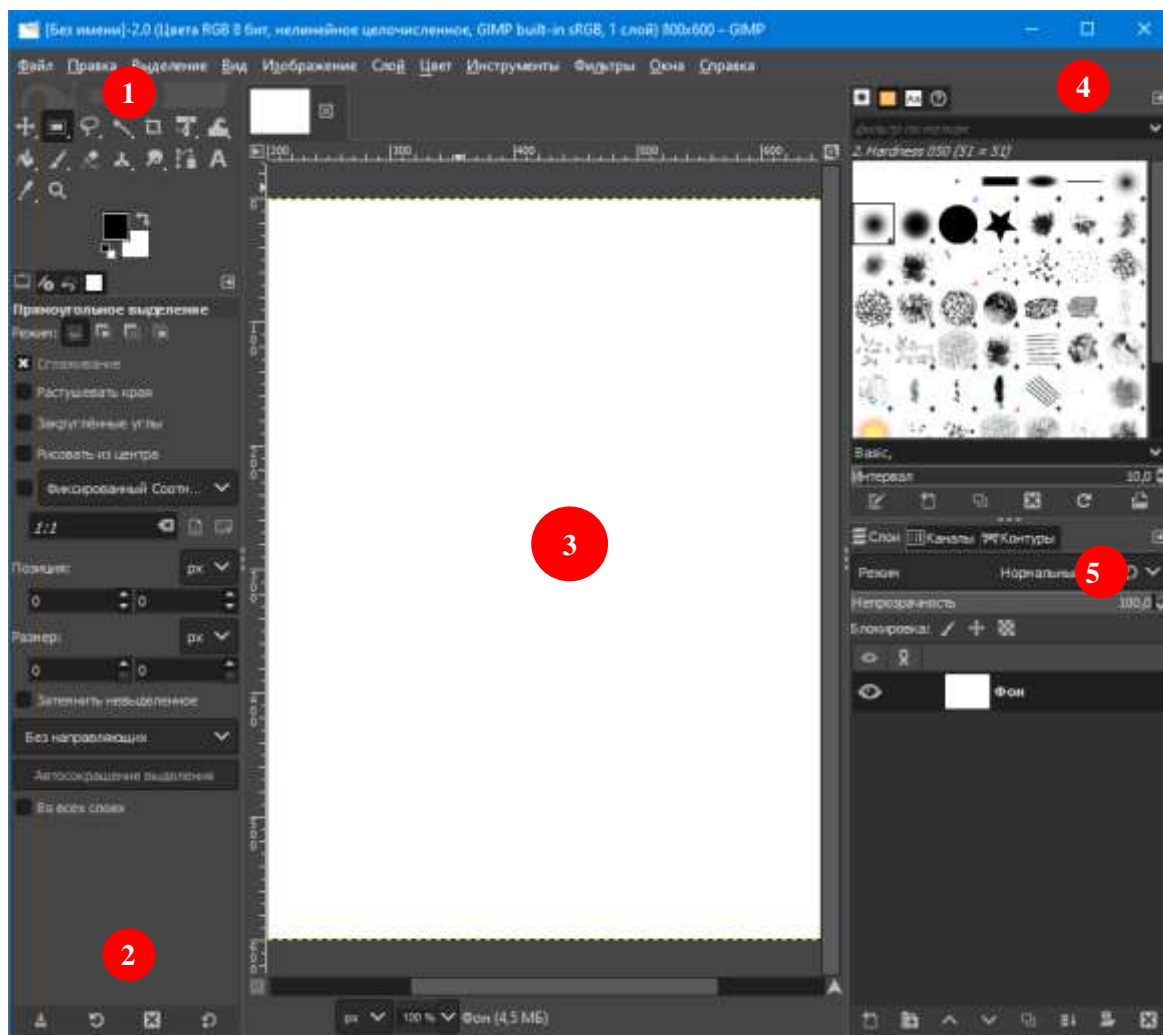


Рис. 1.1. Общий вид редактора GIMP

Приведенный набор – это минимальный набор окон. В GIMP используется более дюжины различных диалогов, которые можно при необходимости открыть. Опытные пользователи обычно держат открытыми панель инструментов (с параметрами инструментов) и диалог **Слой**.

Панель инструментов

Панель инструментов (рис. 1.2) – единственная часть интерфейса программы, которую нельзя продублировать или закрыть.

Панель инструментов включает:

1. Кнопки, которые активируют инструменты для разнообразных действий: выделение частей изображений, рисования, преобразования и т. п.
2. Цвета фона/переднего плана: область выбора цвета показывает текущий выбранный цвет переднего плана и фона, который применяется во многих операциях. Щелчок по одному из них вызовет выборщик цветов, который позволит установить другой цвет.




Рис. 1.2. Панель инструментов

Окно изображения

Каждое открытое изображение в GIMP отображается в своем собственном окне. Существует два основных режима отображения окон, панелей и диалогов, которые выбираются командой **Окна** → **Одноконный режим**.

Элементы окна (рис. 1.3) с отключённой опцией одноконного режима:

1. Заголовок изображения содержит ряд полезных сведений: имя файла изображения, наименование цветовой модели, номер текущего слоя, размер изображения в пикселях.

2. Прямо под заголовком находится меню изображения. С помощью этого меню можно получить доступ ко всем операциям, применимым к изображению. Также можно вызвать меню изображения щелчком правой кнопкой мыши на изображении или щелчком левой кнопкой мыши по небольшому значку – стрелке  в левом верхнем углу 3.

3. Щелчок по этой небольшой кнопке вызывает меню изображения, оформленное в виде столбца. Такие кнопки широко используются в GIMP для вызова меню в различных окнах.

4. Линейки, которые используются для измерений. По желанию можно выбрать, в каких единицах измерения отображаются координаты. Единицей измерения по умолчанию являются пиксели. Одно из основных действий для работы с линейками – это создание направляющих. Если щелкнуть по линейке и перетащить на окно изображения, будет создана направляющая линия, которая поможет аккуратно расположить объекты на изображении.

5. В левом нижнем углу окна изображения расположена небольшая кнопка, которая включает или выключает быструю маску, являющуюся альтернативным и часто полезным методом просмотра выделенной области внутри изображения.

6. В левом нижнем углу окна расположена прямоугольная область, используемая для отображения текущих координат указателя (положение мыши).

7. Используемыми по умолчанию единицами измерения для линеек и некоторых других целей являются пиксели. Можно заменить их на дюймы, сантиметры или другие доступные единицы.

8. Меню изменения масштаба.

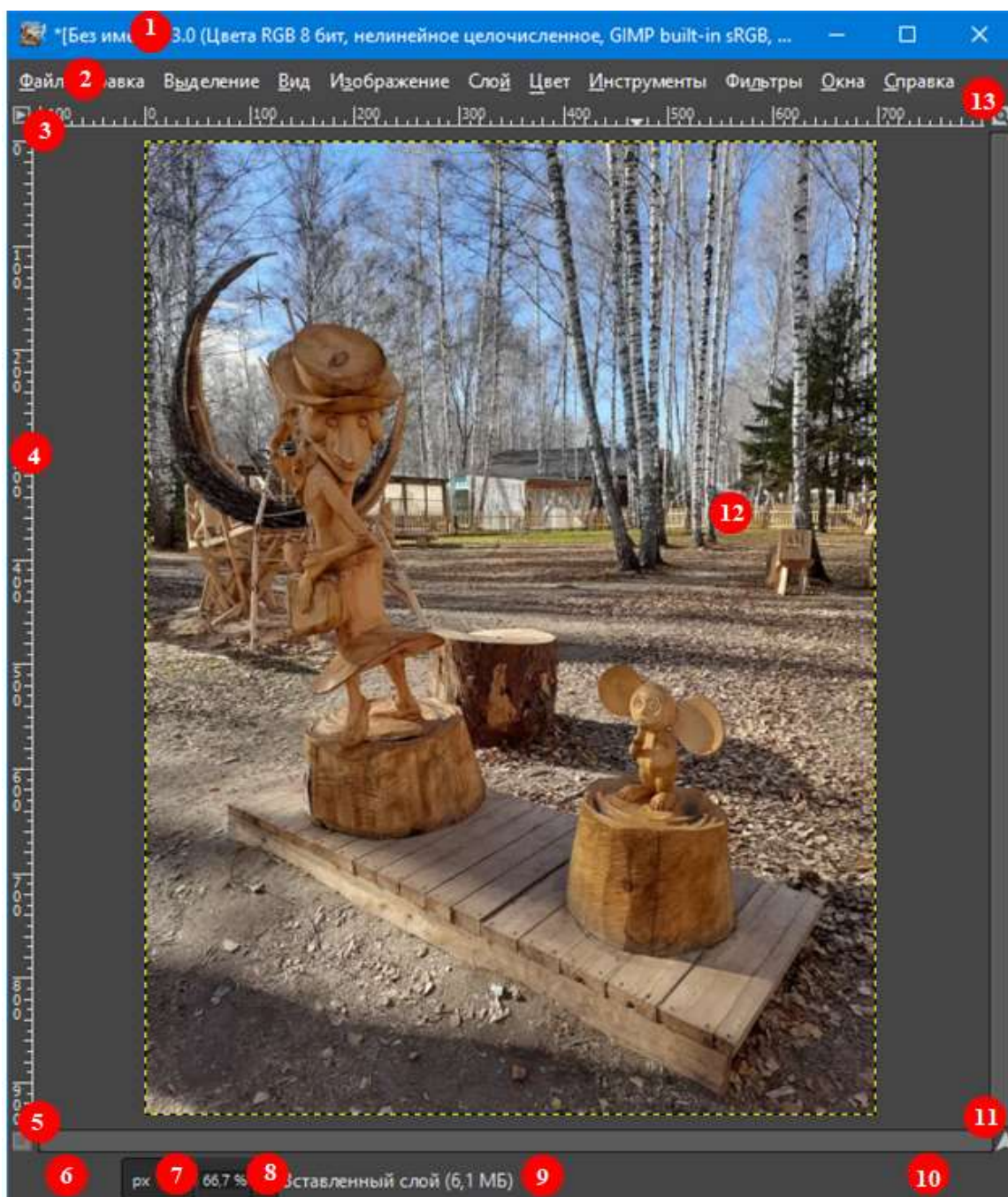



Рис. 1.3. Окно изображения

9. Область статуса расположена под изображением. Она отображает активный слой изображения и количество занятой изображением системной памяти.

10. Кнопка отмены. Если запущена сложная, занимающая время операция (обычно дополнение), кнопка **Отмена** временно появится в правом нижнем углу окна. Нажатие на эту кнопку немедленно отменит операцию.

11. Панель навигации – небольшая кнопка , расположенная справа, внизу под изображением. Можно перемещаться к другим частям изображения, двигая мышь при нажатой кнопке.

12. Наиболее важная часть окна изображения – это, конечно, само изображение. Оно занимает центральную область окна и окружено желтой пунктирной линией, в отличие от нейтрального серого цвета фона.

13. Кнопка **Изменение размера изображения**. Если эта кнопка нажата, при изменении размера окна будет меняться масштаб изображения.

Диалоги и панели

Начиная с версии 2.4 (GIMP) пользователь получил больше удобства в плане размещения диалоговых окон на экране. Вместо размещения каждого диалога в своем собственном окне появилась возможность группировать их вместе с помощью панелей. Панель – это окно-контейнер, которое может содержать собрание постоянных вкладок-диалогов, таких как **Параметры инструментов, Кисти, Палитры** и др. Вкладки можно перетаскивать с помощью мыши по панели.

В каждой панели имеется специальная кнопка для настройки вкладок (рис. 1.4).

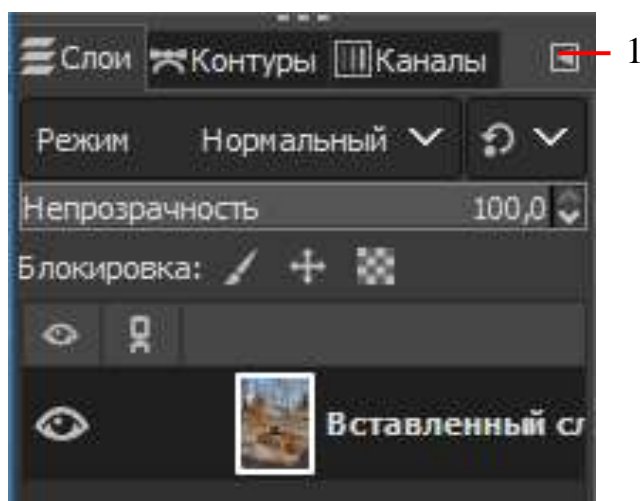


Рис. 1.4. Панель с кнопкой настройки вкладок

С помощью кнопки 1 (рис. 1.4) можно выполнить ряд действий с диалогами: добавление, закрытие, прикрепление, отсоединение вкладки.

Работа с файлами

Создание нового изображения. В GIMP существует несколько путей для создания нового изображения. Классический путь – это выбор пункта меню: **Файл** → **Создать проект**. При этом откроется диалог «**Создать новое изображение**» (рис. 1.5), где можно установить начальные ширину и высоту файла.

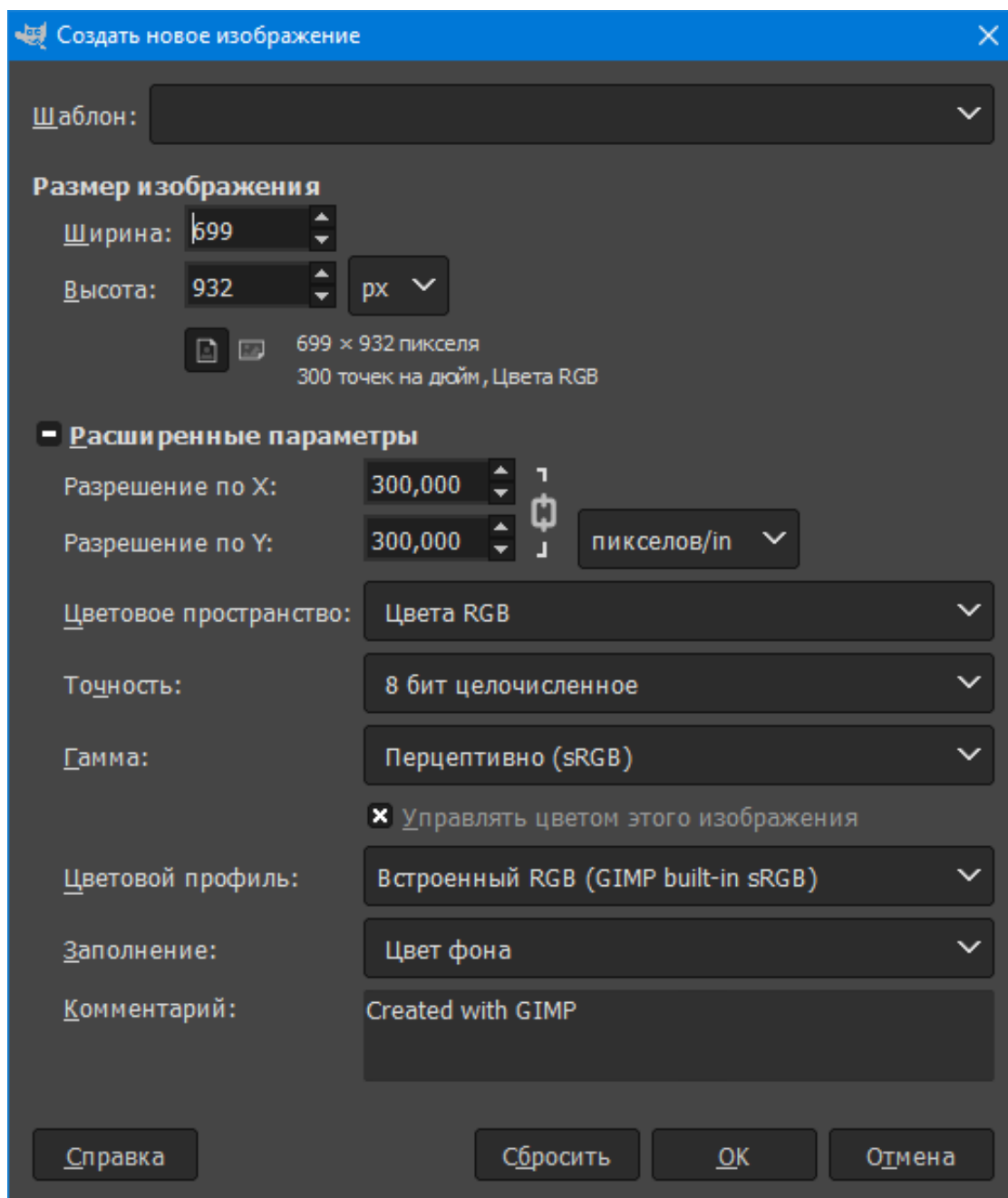


Рис. 1.5. Диалог «Создать новое изображение»

При выборе расширенных параметров устанавливается разрешение, цветовая модель и цвет фона.

Вторая команда главного меню **Файл** → **Создать** позволяет создать изображение и вставить рисунок из буфера обмена. При этом будут установлены размеры изображения, которое находится в буфере обмена. Также при выборе этой команды возможен захват изображения с экрана, сканера или фотокамеры.

Открытие изображения. Доступно несколько способов открыть существующее изображение в GIMP. Наиболее очевидный – это открыть его с помощью меню **Файл** → **Открыть** в главном меню. При этом появится диалог **Выбор файла**. Другой способ заключается в использовании технологии drag&drop. Если значок файла перетащить на существующее изображение в GIMP, то файл добавится как новый слой или слои этого изображения.

Сохранение изображения. GIMP поддерживает собственный формат для хранения изображений XCF, в котором можно хранить информацию о слоях в изображении. Для сохранения изображения в формате XCF необходимо выбрать команду **Файл** → **Сохранить**. Для сохранения изображения в других популярных растровых форматах, например в формате jpg, используется команда **Файл** → **Экспортировать как....** После этого в появившемся окне (рис. 1.6) необходимо задать папку, куда будет сохраняться файл, имя и тип файла.

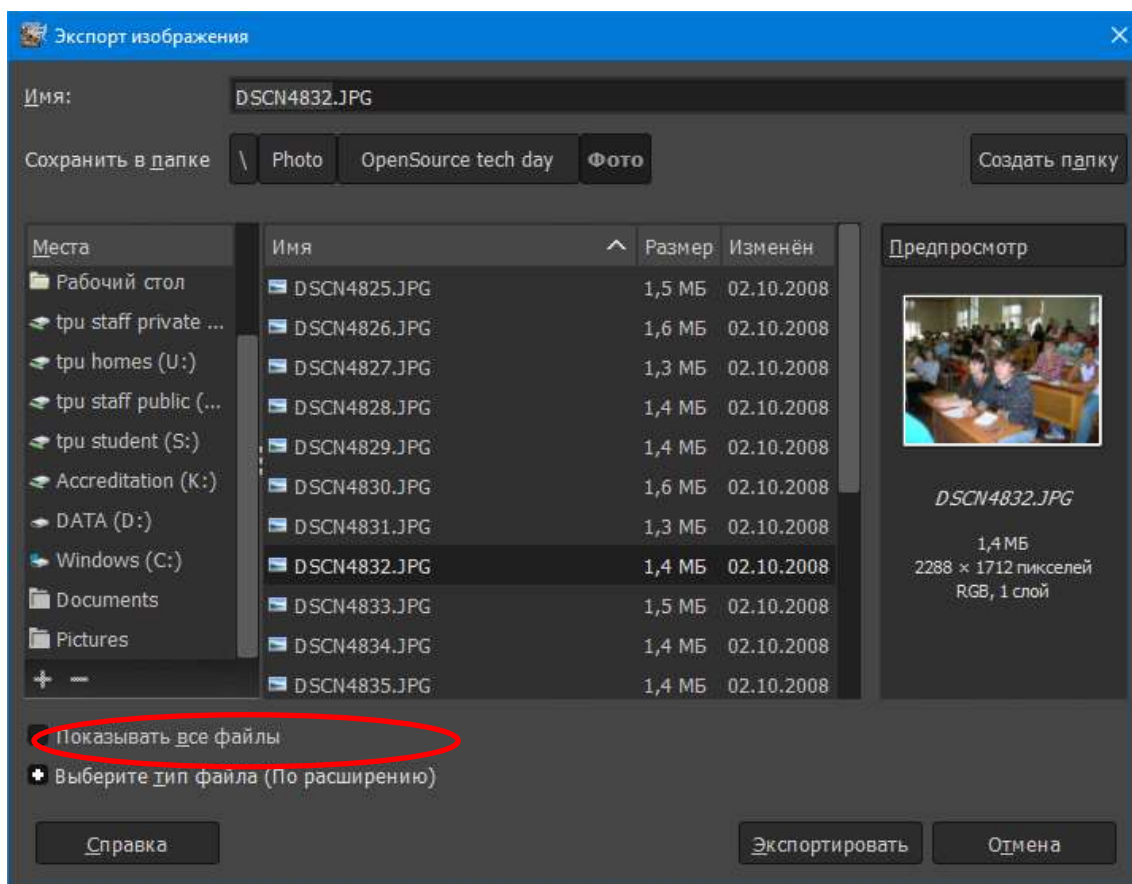


Рис. 1.6. Диалог сохранения файла

Для задания формата графического файла достаточно ввести соответствующее расширение (gif, bmp, tif и т. п.) после имени файла при выбранном параметре **По расширению** либо выбрать тип файла, расширив диалог сохранения файла (рис 1.7).

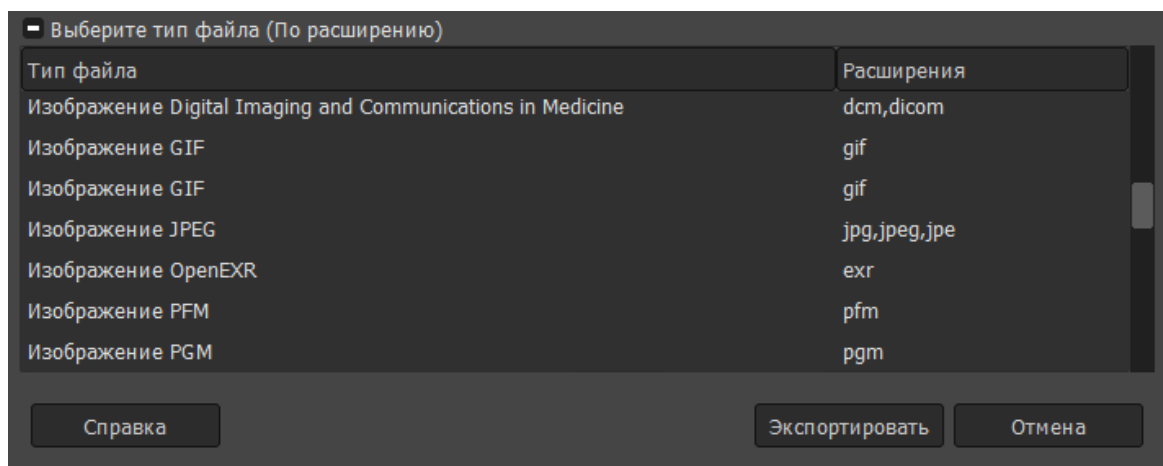


Рис. 1.7. Задание типа изображения

При сохранении изображения в некоторые форматы могут появляться дополнительные окна для задания параметров изображения. Отметим формат jpg, при сохранении в котором можно задавать качество изображения. Чем выше будет задано качество, тем больший размер будет у файла, хранящего изображение.

Изменение масштаба и навигация по изображению

В ряде случаев, например при обработке некоторых относительно маленьких областей, возникает необходимость изменения масштаба отображения изображения на экране. Это можно осуществить несколькими способами через интерфейсную часть программы, через клавиатуру и мышь. Текущий масштаб можно увидеть внизу окна изображения (рис. 1.3, область 8).

В меню изображения **Вид** найдите опцию **Масштаб**. Открывается подменю, в котором находится множество возможностей изменить масштаб изображения на экране.

Способ изменения масштаба через клавиатуру заключается в использовании кнопок + (плюс) и – (минус).

Рассмотрим перемещение по увеличенному изображению через кнопку навигации. Для этого:

- 1) увеличим изображение до 400 %;
- 2) нажмем на кнопку навигации (рис. 1.3, кнопка 11) и, не отпуская левую кнопку мышки, переместимся в любую часть масштабированного изображения.

Рисование. Кисти

Инструменты рисования представлены на рис. 1.8.



Рис. 1.8. Инструменты рисования

Работа с инструментами Заливка, Карандаш, Кисть, Ластик, Аэрограф, Перо, Размывание/резкость, Палец, Осветление/Затемнение отражена в их названии. Для простых действий применение данных инструментов не представляет сложности.

При выборе любого инструмента внизу панели инструментов отображаются его параметры (см. рис. 1.1, область 2).

Основным инструментом рисования является кисть. При выборе кисти устанавливается **Режим**, который по умолчанию стоит в значении **Нормальный** (рис. 1.9). Это позволяет рисовать линии определённым цветом. Все остальные режимы при нанесении цвета учитывают также цвет фона – тем самым получается смешение цветов.

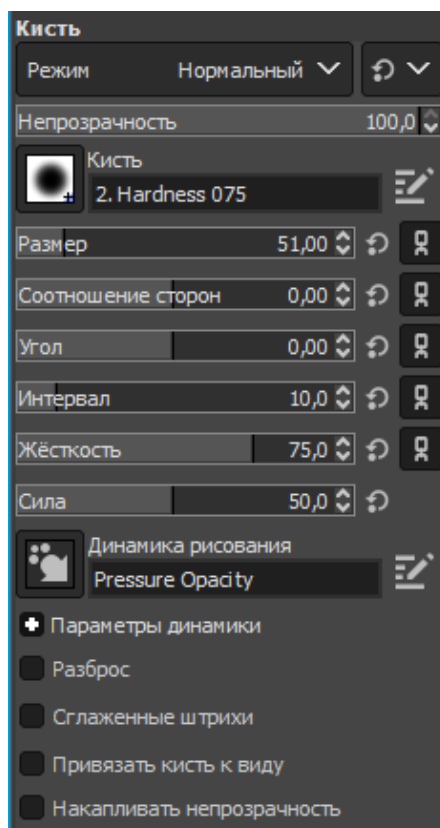


Рис. 1.9. Параметры кисти

Далее определяется, на сколько будет непрозрачен цвет, наносимый кистью, форма и размер кисти. Интересен тот факт, что любой выделен-

ный объект, помещенный в буфер обмена командой **Правка** → **Копировать**, отображается в списке доступных форм кистей и может быть использован как кисть.

Ниже можно задать ряд параметров, позволяющих добиться ряда специальных эффектов для кисти, главное – не забыть выбрать цвет, которым будем рисовать. Для выбора цвета на панели инструментов существуют специальные элементы (рис. 1.10).

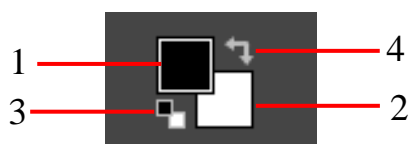


Рис. 1.10. Элемент выбора цветов

Для выбора цвета кисти, карандаша, заливки используется область 1. Для выбора цвета фона, цвета ластика – область 2. Обе области можно применить для задания градиента. Градиент – это плавный переход от одного цвета к другому. Элемент 3 задает цвета по умолчанию: черного – основного и белого – цвета фона. Элемент 4 позволяет поменять цвет фона с основным цветом.

При нажатии на область 1 или 2 (рис. 1.10) открывается дополнительная панель для выбора цвета (рис 1.11).

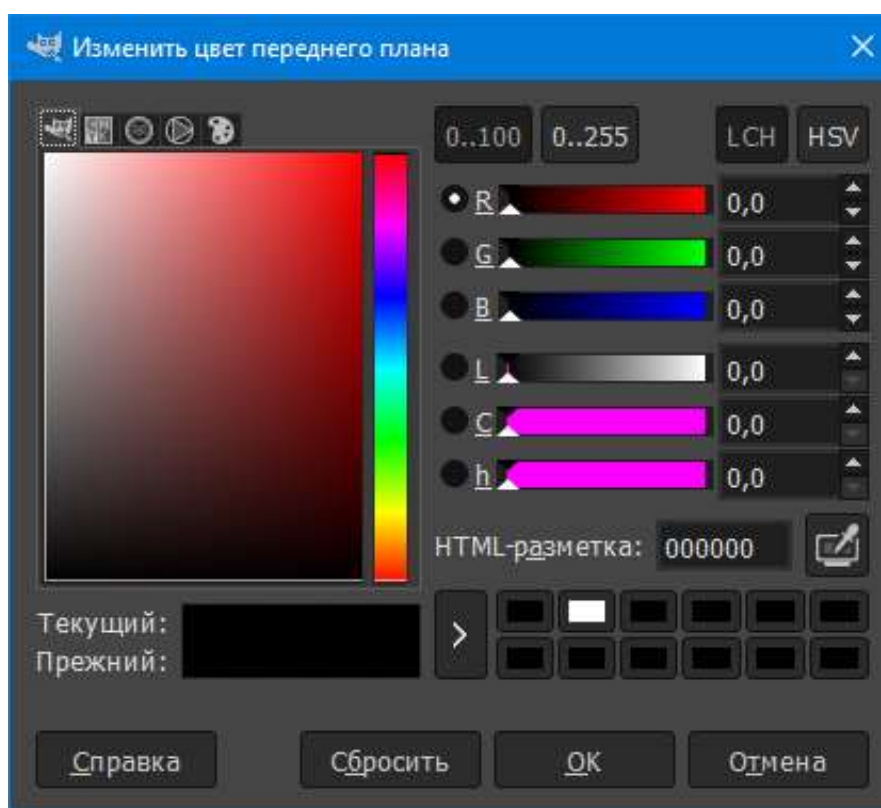


Рис. 1.11. Панель для выбора цвета

Отмена действий

Почти все, что делается с изображением, может быть отменено. Можно отменить последнее действие, выбрав в меню изображения **Правка** → **Отменить**, но эта операция применяется так часто, что лучше запомнить сочетание клавиш **Ctrl+Z**.

Сама отмена также может быть отменена. После отмены действия можно вернуть его, выбрав в меню изображения пункт **Правка** → **Повторить** или с использованием клавиши быстрого доступа **Ctrl+Y**. Часто это полезно при оценке эффекта какого-либо действия, с помощью его неоднократной отмены и повтора.

Если часто используется отмена и возврат на множество шагов за раз, возможно, будет более удобно работать с диалогом **Истории действий** – прикрепляемой панелью, которая показывает небольшие эскизы каждой точки в истории отмены, позволяя перемещаться назад или вперед к точке, по которой произведен щелчок мыши (рис. 1.12).

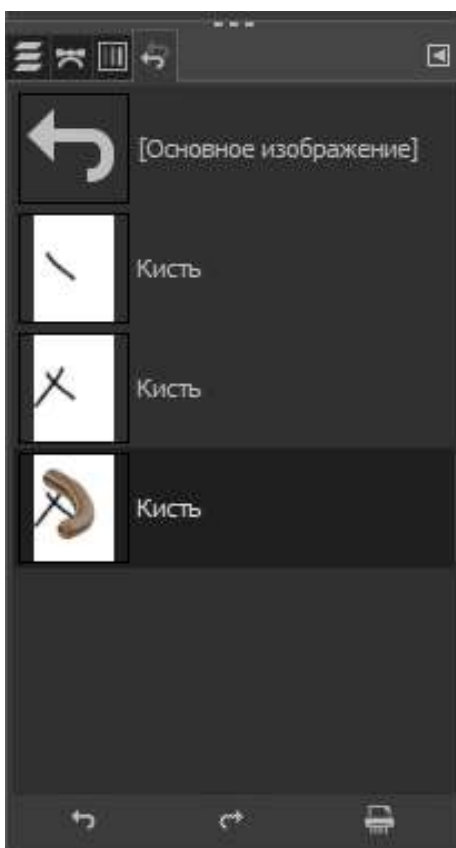


Рис. 1.12. Панель «История действий»

Задание по лабораторной работе

1. Запустите GIMP.
2. Создайте изображение размером 640×480 пикселей с разрешением 72 dpi. При этом используйте цветовую модель RGB.

3. Используя различные инструменты рисования, создайте изображение. Обязательно попробуйте применить кисти различных форм и размеров, различные режимы наложения цветов, специальные эффекты. Также используйте ластик, заливку и градиентную заливку.

4. Полученное изображение сохраните в различных форматах: xcf, bmp, tif (используя LZW-компрессию), png, gif, gif с градациями серого цвета, jpg (с различной степенью сжатия – 90, 60, 40).

5. Создайте отчет в тестовом редакторе MS Word, вставьте в него изображения из полученных файлов и запишите после каждого размер полученного файла.

6. Создайте изображение 20×20 пикселей.

7. Увеличьте масштаб отображения и создайте рисунок с эмблемой какой-либо компании, используя ограниченное число цветов.

8. Сохраните полученное изображение в различных форматах: bmp, tif (используя LZW компрессию), gif, jpg (с различной степенью сжатия – 90, 60, 40).

9. Добавьте в отчет полученный изображения из сохранённых файлов. Запишите после каждого изображения размер полученного файла.

10. Проанализируйте результаты и сделайте выводы. При этом оцените качество полученных изображений и размеры файлов.

Контрольные вопросы

1. В каких операционных системах может быть установлен и использован растровый графический редактор GIMP?

2. Что такое слои, каналы и выделение в GIMP?

3. В чем заключается разница в сохранении изображения и его экспорте?

4. Зачем устанавливается режим при рисовании кистью?

Лабораторная работа 2 ФОТОМОНТАЖ

Цель: познакомиться с основными способами получения фотоколлажей в растровом редакторе GIMP.



Общие сведения

Выделение областей

Выделение области является одним из важнейших этапов работы с изображением, с помощью которого выделяют объекты на изображении, чтобы использовать их в дальнейшем для фотомонтажа. Выделенные области можно заливать цветом, текстурой, градиентом. С выделенными областями можно проводить отдельную цветокоррекцию и к ним можно применять фильтры. Выделенная область обычно отображается в виде пунктирной рамки.

Неудивительно, что для выделения областей существует ряд приемов и инструментов.

Инструменты выделения предназначены для выделения областей активного слоя, чтобы можно было работать только с ними, не трогая всё остальное. Слои и работа с ними будут рассмотрены далее.

Прямоугольное и эллиптическое выделение. Инструменты прямоугольного  и эллиптического выделения  позволяют выделять прямоугольные и эллиптические области соответственно. Это самые простые, но часто используемые типы выделения. Обычно кнопки, выполняющие однотипные действия, такие как выделения областей, могут объединяться в один элемент с возможностью выбора текущего инструмента. Поэтому для активации эллиптического выделения требуется нажать на правый нижний угол кнопки **прямоугольное выделение** и выбрать нужный инструмент.

Для того чтобы воспользоваться инструментом, выделите его на панели инструментов и далее выделите с помощью мыши область на изображении. После этого выделенная область будет отображаться в виде пунктирной рамки (рис. 2.1).

После этого размер выделенной области можно менять. Для этого используются области в углах выделения. Также выделенную область можно переносить, не боясь испортить изображение.

Более подробно параметры выделения можно задать в свойствах инструмента (рис. 2.2).



Рис. 2.1. Выделенная область

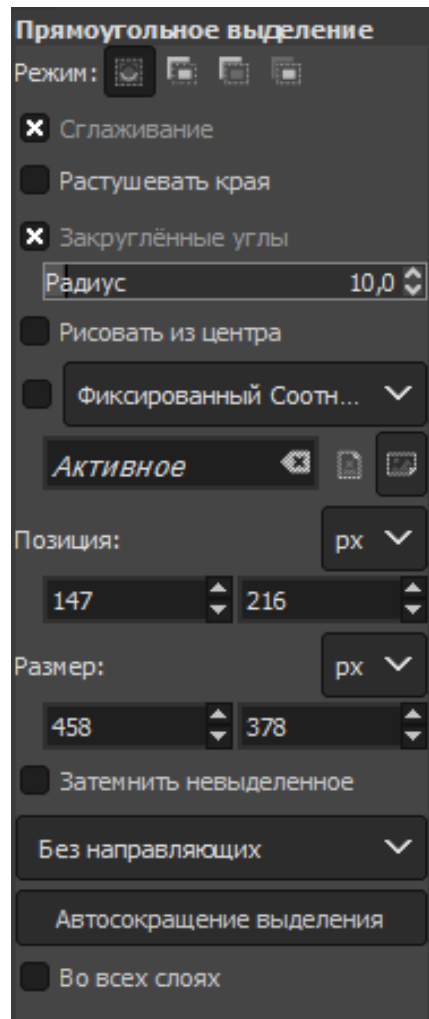


Рис. 2.2. Параметры инструмента «Прямоугольное выделение»

Как видим, что через параметры можно задать точную позицию размещения выделения и точные размеры. После задания размеров можно установить параметр **Фиксированный Соотношения**, означающий фиксировать, например, соотношение сторон. Это позволяет делать выделение, например, по размеру печати фотографий 10:15 и т. п.

Наиболее важным параметром при сложных выделениях является **Режим** (рис. 2.3).

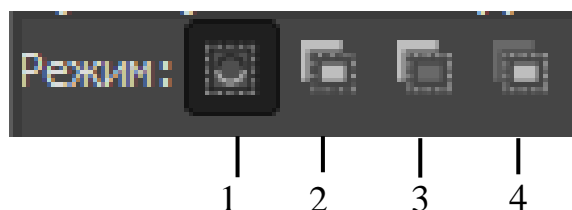


Рис. 2.3. Режимы выделения

Первый режим называется «Заменить текущее выделение» (рис. 2.4, кнопка 1). В таком режиме каждое выделение происходит заново, а предыдущее выделение снимается. Остальные режимы учитывают, какое выделение было сделано до этого. Второй режим «Добавить в текущее выделение» к уже сделанному выделению добавляет новую область. Причем можно не выбирать данный режим в свойствах инструмента выделения, а при выделении удерживать кнопку **Shift**. Третий режим позволяет вырезать из сделанной области какую-либо часть. В этом режиме можно работать, удерживая кнопку **Ctrl**. Последний режим позволяет найти пересечение.

Комбинируя режимы и инструменты, можно создать выделенную область достаточно сложной формы, как, например, на рис. 2.4.

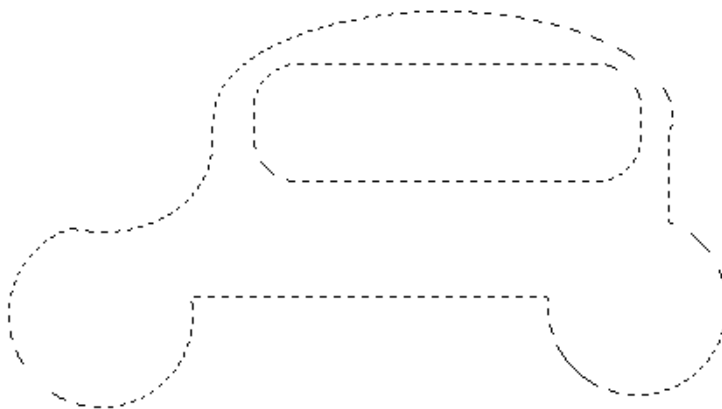


Рис. 2.4. Пример использования режимов выделения

На основе использования этих приемов создаются области выделения для разнообразных графических кнопок, размещаемых на сайтах. Более подробно эту информацию можно найти в Интернете, набрав поисковый запрос «уроки создания кнопки в GIMP».


Свободное выделение и работа с быстрой маской. Инструмент **Свободное выделение**  позволяет выделять свободные области (рис. 2.5) при удержании левой кнопки мыши. При однократных нажатиях выделение происходит с помощью многоугольника.



Рис. 2.5. Пример выделения с помощью инструмента «Свободное выделение»

После такого выделения часто возникает необходимость коррекции области выделения, что удобно сделать, перейдя в режим быстрой маски нажав соответствующую кнопку (рис. 1.3, кнопка 5) или сочетания **Shift+Q**. В этом режиме выделенная область остается цветной, а невыделенная отображается красным (рис. 2.6).



Рис. 2.6. Режим быстрой маски

В этом режиме для изменения границ выделенной области используются инструменты рисования: кисть для добавления к красной области и ластик для расширения области выделения.

Выход из режима быстрой маски осуществляется таким же образом, как и вход.


Умные ножницы. Инструмент **Ножницы**  используется для выделения объектов по краю. Для этого необходимо расставить ряд опорных точек по краю объекта (рис. 2.7), замкнуть линию и щелкнуть внутри области. При этом GIMP пытается самостоятельно определить цветовые границы. Этот метод хорошо работает, когда выделяемый объект не сливается с другими по цвету. Однако завершающим этапом выделения рекомендуется использовать доработку области выделения в режиме быстрой маски.



Рис. 2.7. Выделение с помощью умных ножниц

Выделение по цвету. Часто возникает необходимость выделения какой-либо области пикселей, похожих по цвету (например, для выделения объектов на однородном фоне). В этом случае выделяют фон, а потом инвертируют выделение (командой из главного меню **Выделение** → **Инвертировать**). Таким образом, оказывается выделенным сам объект.


Для выделения пикселей, близких по цвету, используется инструмент **Выделение смежных областей**, который часто называется волшебной палочкой . Так, один щелчок на синей области может привести к выделению, показанному на рис. 2.8.



Рис. 2.8. Результат использования «Волшебной палочки»

При выборе данного инструмента самым главным параметром является **Порог**, который определяет чувствительность выделения к цветам. Так, снижение порога до значения 4,0 в показанном ранее примере может существенно сократить выделяемую область (рис. 2.9).



Рис. 2.9. Результат снижения значения порога

И последнее: снятие выделения происходит при выборе **Выделение** → **Снять** или нажатии **Shift+Ctrl+A**.

Работа со слоями

Удобно представлять изображение в GIMP как пачку прозрачных листов: в терминологии GIMP каждый прозрачный лист носит название «слой». В принципе нет ограничений на количество изображений в слое: единственное ограничение – это количество доступной памяти в системе.

В GIMP границы слоя необязательно равны границам его содержащего изображения. Когда пользователь использует инструмент **Текст**, каждый текстовый элемент располагается в своём отдельном слое и слой равен размеру текста, не больше. Когда пользователь создаёт новый слой с помощью вырезания и вставки, новый слой имеет достаточный размер для размещения вставленного содержимого. В окне изображения границы текущего активного слоя показаны чёрно-жёлтой пунктирной линией.

Структура слоёв в изображении показана в диалоге «Слой» (рис. 2.10).

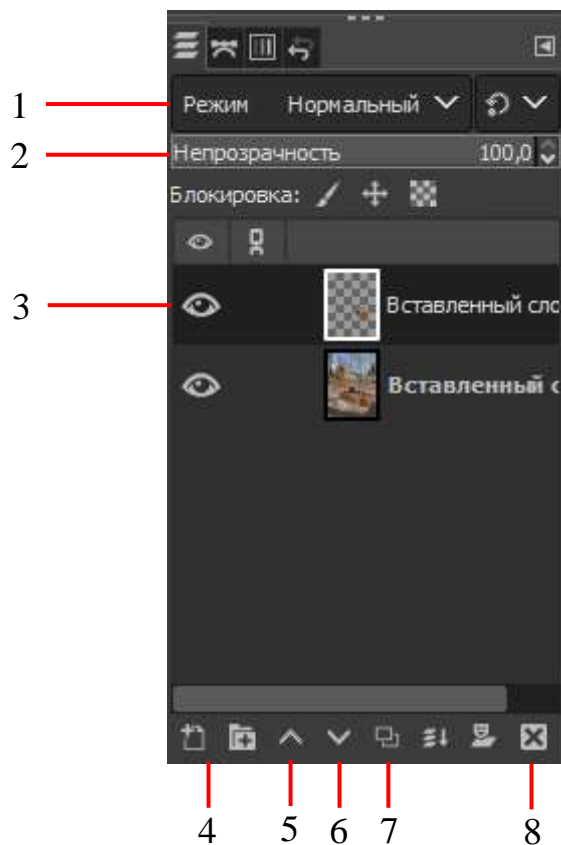


Рис. 2.10. Диалог **Слои**

В диалоге **Слои** можно изменять приведенные далее свойства выделенного слоя.

Непрозрачность. Прозрачность слоя определяется степенью доступных цветов из нижних отображаемых слоёв списка. Непрозрачность задается в диапазоне от 0 до 100, где 0 означает полную прозрачность и 100 – полную непрозрачность (рис. 2.10, элемент 2).

Видимость. Существует возможность временно не отображать слой без его уничтожения, с помощью щелчка по пиктограмме «глаза» (рис. 2.10, кнопка 3) в диалоге слоёв. Это называется «переключением видимости» слоя. Для большинства операций над изображением отключение видимости равносильно отсутствию слоя. При работе с изображением, содержащим множество слоёв с разной прозрачностью, рекомендуется время от времени отключать видимость других слоев, чтобы рассмотреть текущий слой без помех.

Режим. Режим слоя (рис. 2.10, элемент 1) определяется способом комбинации цветов из текущего и расположенного ниже слоя для представления видимого результата.


Режимы слоя иногда называются «режимами смешивания». Выбор режима слоя изменяет внешний вид слоя или изображения в зависимости

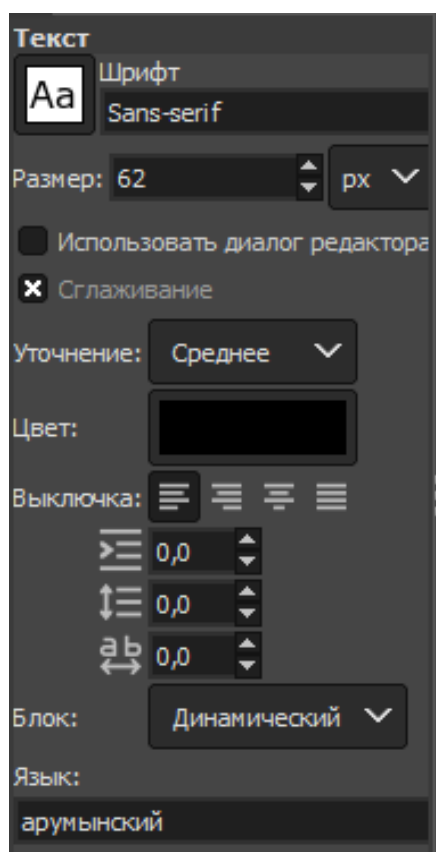
от нижележащих слоёв. Если есть только один слой, то режим слоя ни на что не влияет. Поэтому должно быть по крайней мере два слоя, чтобы использовать режимы слоя.

Кнопки внизу диалога **Слой** позволяют создавать новые слои (рис. 2.10, кнопка 4), изменять порядок следования слоев (рис. 2.10, кнопки 5, 6), копировать слой (рис. 2.10, кнопка 7), удалять выделенный слой (рис. 2.10, кнопка 8).

Одним из этапов фотомонтажа обычно является создание нового слоя (рис. 2.10, кнопка 4) на изображении, а затем вставка выделенного объекта из другого изображения. Для вставки может использоваться буфер обмена и стандартные команды **Правка** → **Копировать**, **Правка** → **Вставить**. Более подробно фотомонтаж рассмотрим в следующих разделах.

Текст в GIMP

На изображение может быть добавлен любой текст с помощью инструмента **Текст** . При использовании этого инструмента происходит



добавление текста в *текстовый слой* – особый вид слоя, содержащий только текст. Сам текст может быть отредактирован в дальнейшем с помощью того же инструмента **Текст**. При выборе инструмента можно задать параметры шрифта (рис. 2.11), такие как: название шрифта, размер, цвет.

После выбора инструмента **Текст** на изображении надо выделить прямоугольную область, в которую вводится текст. Для редактирования ранее введённого текста необходимо выбрать инструмент **Текст** и нажать на текст.


Для размещения текста по контуру создается сам контур с помощью кнопки **Контур** . Далее нужно убедиться, что данный контур создан на вкладке диалога контуров и является активным. Далее во вкладке **Слой** необходимо найти слой, соответствующий тексту, нажать правой кнопкой на него и выбрать команду **Текст по контуру**.

Рис. 2.11. Параметры инструмента «Текст»

Преобразование изображения в слое

Для преобразования слоя существует ряд инструментов (рис. 2.12) в группе инструментов **Трансформации**.

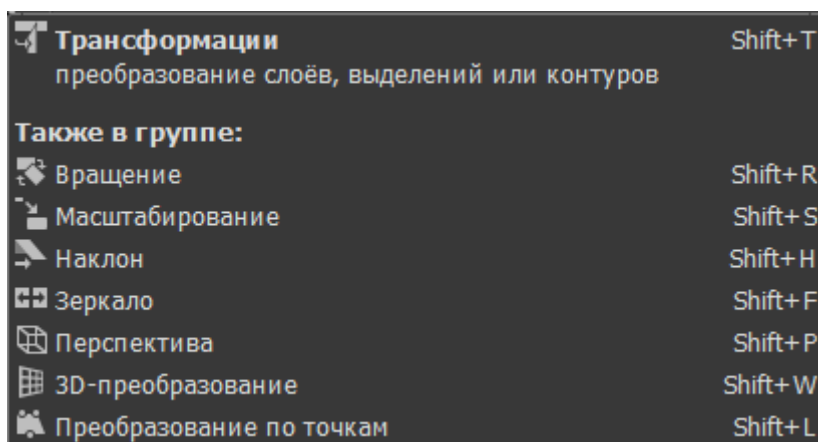


Рис. 2.12. Инструменты преобразования

Некоторые из приведенных инструментов можно применить к отдельному выделенному слою, выделению или в целом к изображению.

Общие свойства инструментов преобразования. Перед изучением инструментов преобразования заметим, что некоторые параметры для этих инструментов являются общими. Во-первых, это группа кнопок **Преобразование** (рис. 2.13), которая доступна в свойствах инструмента при выборе любого инструмента преобразования.

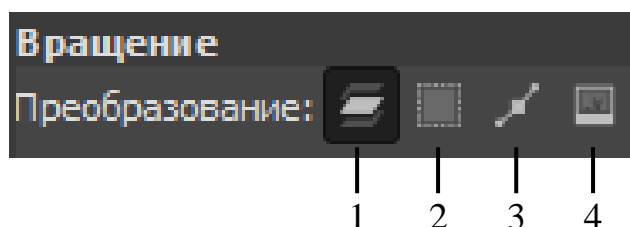




Рис. 2.13. Группа кнопок **Преобразование**

При выборе первой кнопки инструмент работает над активным слоем. Если в слое есть выделение, то выделенная часть изображения будет трансформирована. При выборе второй кнопки инструмент работает только над выделением, при выборе третьей – только над контуром. И, наконец, четвертая кнопка позволяет делать преобразования в целом над всем изображением.

Инструменты преобразования. Кратко рассмотрим назначение инструментов преобразования.

Инструмент **Перемещение**  служит для переноса активного слоя, выделения или контура.

Инструмент **Кадрирование**  предназначен для удаления областей с края изображения или слоя. Чаще всего этот инструмент применяется не к слою, а ко всему изображению, перед выводом на печать для задания нужных соотношений сторон, например 10×15. Для этого в параметрах инструмента устанавливается галочка в положении **Фиксированный Соотношения**, а в соответствующем списке должно быть установлено **Соотношение сторон**. Для обрезки только текущего слоя в параметрах инструмента устанавливается галочка в опции **Только текущий слой**.

Поскольку при фотомонтаже размеры изображений сильно отличаются, необходимо провести коррекцию масштаба отдельных объектов, находящихся в разных слоях. Для этого применяют инструмент **Масштабирование** . При выборе этого инструмента и попытке масштабирования изображения появляется дополнительный диалог **Масштабирование** (рис. 2.14).

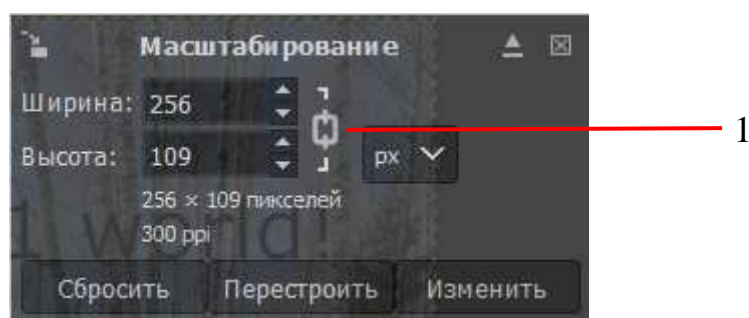






Рис. 2.14. Диалог «Масштаб»

Если нужно сохранить пропорции при масштабировании, необходимо нажать на элемент в виде цепочки (рис. 2.14, область 1), чтобы она обрела вид единого целого. Далее, применяя перетаскивание за углы выделенного объекта на слое, производится масштабирование. Окончательно преобразование подтверждается нажатием кнопки **Изменить** в диалоге **Масштаб**. Подобные диалоги появляются и при других преобразованиях.

Инструмент **Вращение**  служит для поворота активного слоя, выделения или контура. При активном инструменте поворот может осуществляться как с помощью мыши, так и через диалог **Вращение**.

Следующие инструменты **Наклон** , служащий для скоса изображения, **Перспектива** , **Зеркало**  выполняют соответствующие преобразования аналогичным образом. Они могут быть использованы для построения перспективной тени выделенного объекта следующим

образом. Сначала слой дублируют с помощью кнопки на панели **Слой** (рис. 2.10, кнопка 7) и получают две копии объекта (рис. 2.15).



Рис. 2.15. Изображения с двумя копиями объекта

Далее используем инструменты **Перспектива**, **Наклон**, при необходимости – **Зеркало**, один слой трансформируем (рис. 2.16).



Рис. 2.16. Пример использования инструментов преобразования

Выделим объект в преобразованном слое. Для этого можно использовать **Волшебную палочку**, выделив сначала фон, а затем инвертировав выделение, выбрав **Выделение** → **Инвертировать**. После этого зальем выделение черным цветом, используя инструмент **Плоская заливка** с установленным параметром **Все выделение** либо с нажатой кнопкой **Shift**. Далее остается переместить слой с помощью инструмента «Перемещение» и установить его прозрачность (рис. 2.17).



Рис. 2.17. Результат работы с копией слоя

Фотомонтаж

Все сведения для выполнения фотомонтажа уже изложены выше, приведем лишь общую схему действий. На первом этапе осуществляется подбор исходных изображений и открытие их в GIMP. Далее с помощью различных приемов и инструментов происходит выделение объектов и копирование их в буфер обмена при выборе команды **Правка** → **Копировать** либо **Ctrl+C**. После этого переходим к фоновому изображению и выполняем вставку объекта (команда **Правка** → **Вставить** либо **Ctrl+V**). Объект в таком случае вставляется как плавающее выделение и лучше всего сразу создать для него новый слой, нажав на панели **Слои** кнопку **Создать новый слой** (рис. 2.10, кнопка 4). После этого используются инструменты преобразования: **Масштабирование**, **Перемещение** и т. д., устанавливаются режимы слоя и прозрачность.

Описанные действия повторяются и для других объектов с других изображений. При необходимости меняется порядок следования слоев с панели **Слои** с помощью соответствующих кнопок (рис. 2.10, кнопки 5, 6).

Для того чтобы информация о слоях не была утеряна, полученное изображение рекомендуется сохранять в формате GIMP – xcf.

Задание по лабораторной работе

1. Подберите 3–4 исходных изображений. Добавьте найденные изображения в отчет по лабораторной работе.
2. Выполните фотомонтаж. Результат сохраните в отчет.
3. Добавьте тени от объектов, как описано выше. Результат этого этапа добавьте в отчет.
4. Добавьте текстовые надписи. Полученное изображение вставьте в отчет.
5. Сохраните полученный файл в формате xcf.

Контрольные вопросы

1. Какие способы выделения областей используются в GIMP?
2. Где в GIMP можно найти структуру изображения по слоям?
3. Какие инструменты используются для преобразования изображения в слое?
4. Какие основные действия необходимо выполнить для фотомонтажа?

Лабораторная работа 3 ОБРАБОТКА ИЗОБРАЖЕНИЙ

Цель: познакомиться с основными способами обработки растровых изображений в графическом редакторе GIMP.

Общие сведения

Коррекция цвета

Для коррекции цвета в GIMP существует ряд инструментов в меню **Цвет** (рис. 3.1).

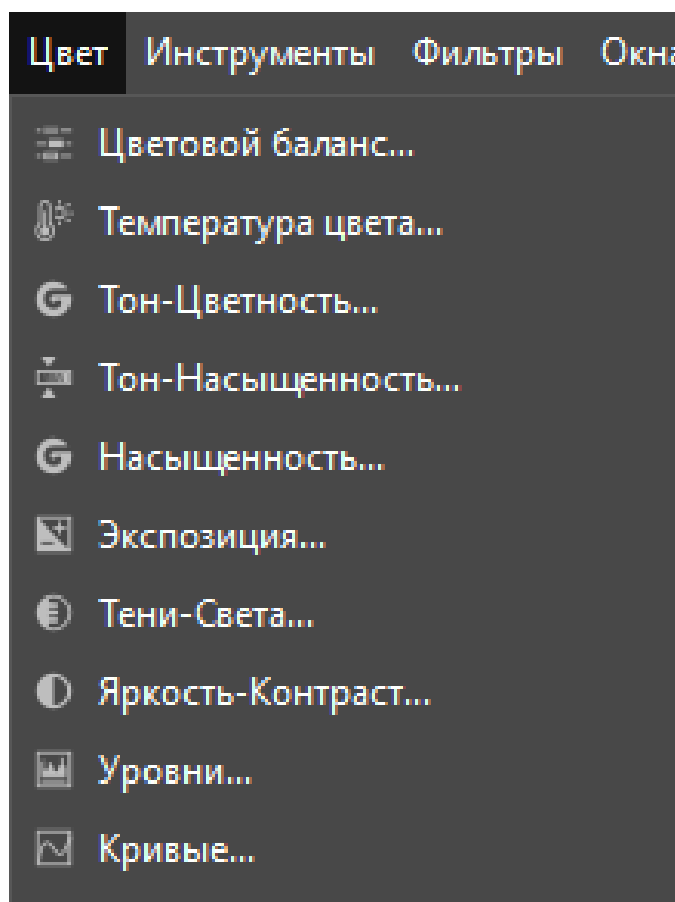


Рис. 3.1. Инструменты для коррекции цвета

Инструменты можно использовать для активного слоя или для участка выделения.

Цветовой баланс. Позволяет регулировать соотношение основных цветов из RGB- и CMY-моделей. Инструмент наиболее полезен для исправления цветов, преобладающих на цифровых фотографиях. Регулировка происходит либо для светлых участков изображения, либо для полутонов, либо для теней с помощью специального диалога (рис. 3.2).

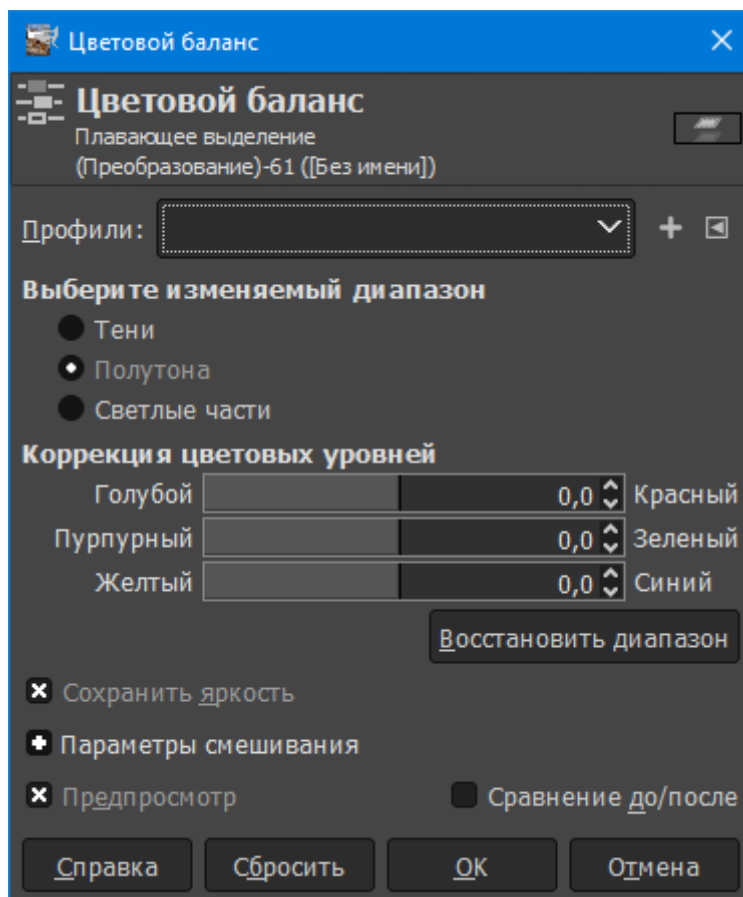


Рис. 3.2. Диалог для корректировки цветового баланса

Коррекция тона, освещенности, насыщенности. Следующий инструмент позволяет менять значения тона, насыщенности и яркости выбранного цветового диапазона в активном слое или выделении с помощью специального диалога (рис. 3.3).

Тон позволяет отличать между собой основные цвета в модели HSV: красный, зеленый, синий, голубой, малиновый (пурпурный), желтый.

В теории цвета **насыщенность** – это интенсивность определённого тона. Можно сказать, что насыщенность – это характеристика цвета, показывающая его чистоту. Насыщенный цвет можно назвать сочным, глубоким, менее насыщенный – приглушённым, приближённым к серому. Также насыщенность позволяет отличать красный цвет от розового, зеленый – от светло-зелёного и т. д.

Полностью ненасыщенный цвет будет оттенком серого. Насыщенность (saturation) – одна из трёх координат в цветовых пространствах моделей HSL и HSV.

Термин «освещенность» (от англ. – *lightness*) в GIMP переведен на русский язык не совсем корректно. В русском языке в теории цвета ис-

пользуют обычно термин **светлота**. **Светлота** – одна из основных характеристик цвета наряду с насыщенностью и тоном. Светлота – это субъективная яркость участка изображения, позволяющая отличать, например, серый цвет от черного, а белый – от серого. Для полного понимания этих характеристик цвета рекомендуется обратиться к лекции по основам компьютерной графики и цветовым моделям HSV и HSL.

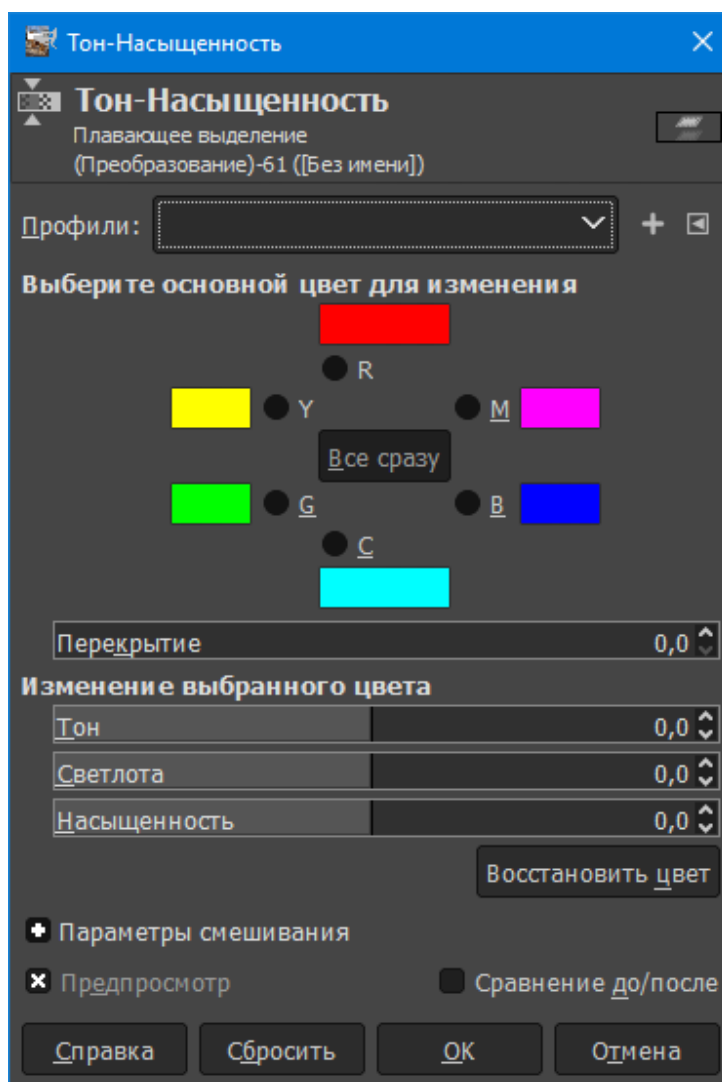
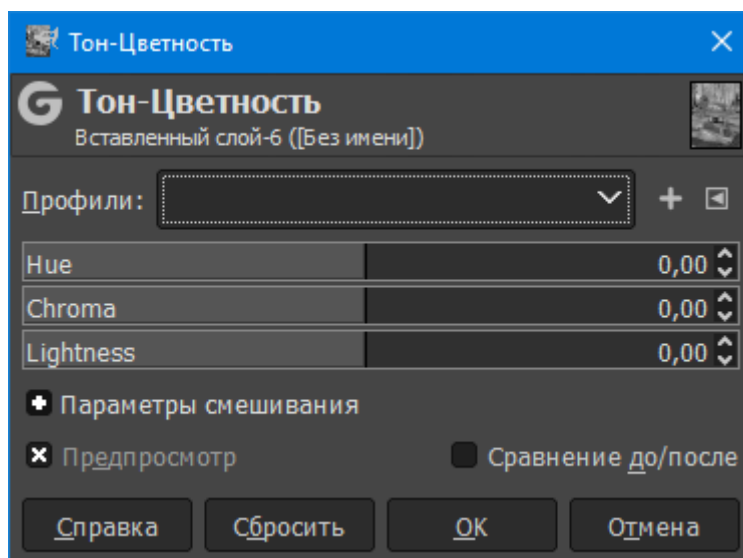


Рис. 3.3. Диалог коррекции тона, освещенности, насыщенности

Тонирование. Пункт меню **Тон-Цветность** также можно использовать для изменения тона, освещенности и насыщенности через специальный диалог (рис. 3.4). Но в отличие от предыдущего инструмента тонирование применяется для изображений в градациях серого цвета для получения цветности и может быть использовано для перевода изображения в сепию (дуотон). Инструменты выделения отдельных областей и тонирования позволяют черно-белую фотографию (изображение в градациях серого цвета) перевести в цвет. Для превращения изображения из

градаций серого в цветное необходимо перейти в режим RGB изображения путем выбора в меню команды **Изображение** → **Режим** → **RGB**.



*Рис. 3.4. Диалог «Тонирование»:
Hue – значение тона; **Chrome** – значение насыщенности;
Lightness – значение освещенности*

Яркость и контраст

Достаточно просто изменяется яркость и контрастность изображения или выделения при использовании инструмента **Яркость-контраст**.

Яркость и контраст являются субъективными характеристиками изображения, воспринимаемыми человеком.

Яркость изображения представляет собой характеристику, определяющую то, насколько сильно цвета пикселей отличаются от чёрного цвета. Например, если оцифрованная фотография сделана в солнечную погоду, то ее яркость будет значительной. Однако если фотография сделана вечером или ночью, то её яркость будет невелика.

Контраст представляет собой характеристику отношения яркостей самой светлой и самой тёмной частей изображения. Иными словами, контраст показывает, насколько большой разброс имеют цвета пикселей изображения. Чем больший разброс имеют значения цветов пикселей, тем больший контраст имеет изображение.

Гистограмма изображения

Несколько следующих команд в меню **Цвет** используют **гистограмму** изображения (уровни) как основной или дополнительный инструмент. **Гистограмма** (в фотографии) – это график распределения полутонов изображения, в котором по горизонтальной оси представлена

яркость, а по вертикали – относительное число пикселей с данным значением яркости (рис. 3.5). Гистограмма изображения позволяет оценить количество и разнообразие оттенков изображения, а также общий уровень яркости изображения.

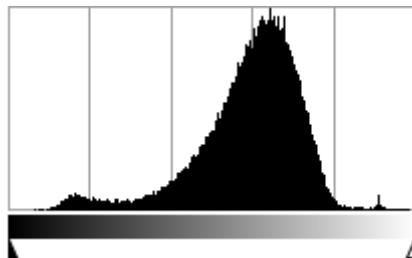


Рис. 3.5. Пример гистограммы

Лучше всего построение гистограммы можно пояснить при помощи изображения, представленного в градациях серого цвета. В этом случае гистограмма представляет собой диаграмму, где по горизонтальной шкале откладываются градации серого от 0 (черный) до 255 (белый), а по вертикальной – количество точек соответствующей градации в этом изображении. Чем выше столбец, тем больше точек этого оттенка серого содержится в фотографии.

Так, гистограмма, приведенная на рис. 3.5, отображает количество пикселей определённого тона для рис. 3.6.

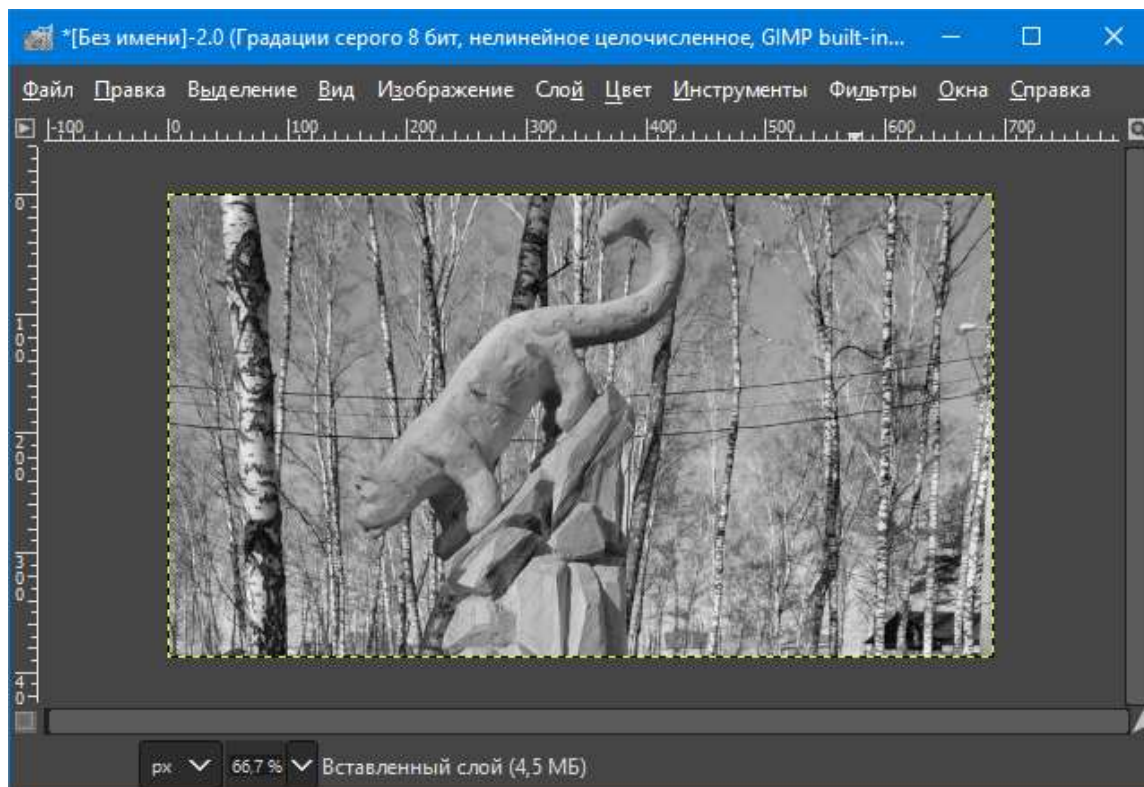


Рис. 3.6. Пример изображения для построения гистограммы

При выборе пункта меню **Уровни** и задании параметров, как показано на рис. 3.7 (бегунки справа и слева сдвинуты к середине), значение интенсивностей равномерно распределяется по всей области градаций серого цвета. Этим самым повышаем контрастность изображения (рис. 3.8).

В некоторых случаях описанный метод позволяет улучшить изображение, а в некоторых наоборот – уменьшить художественную ценность изображения.

Гистограмма для цветного изображения строится по яркости либо по каждому отдельному каналу для основных цветов цветовой модели. Например, для RGB-модели гистограмма может быть построена для трех каналов – R, G и B.

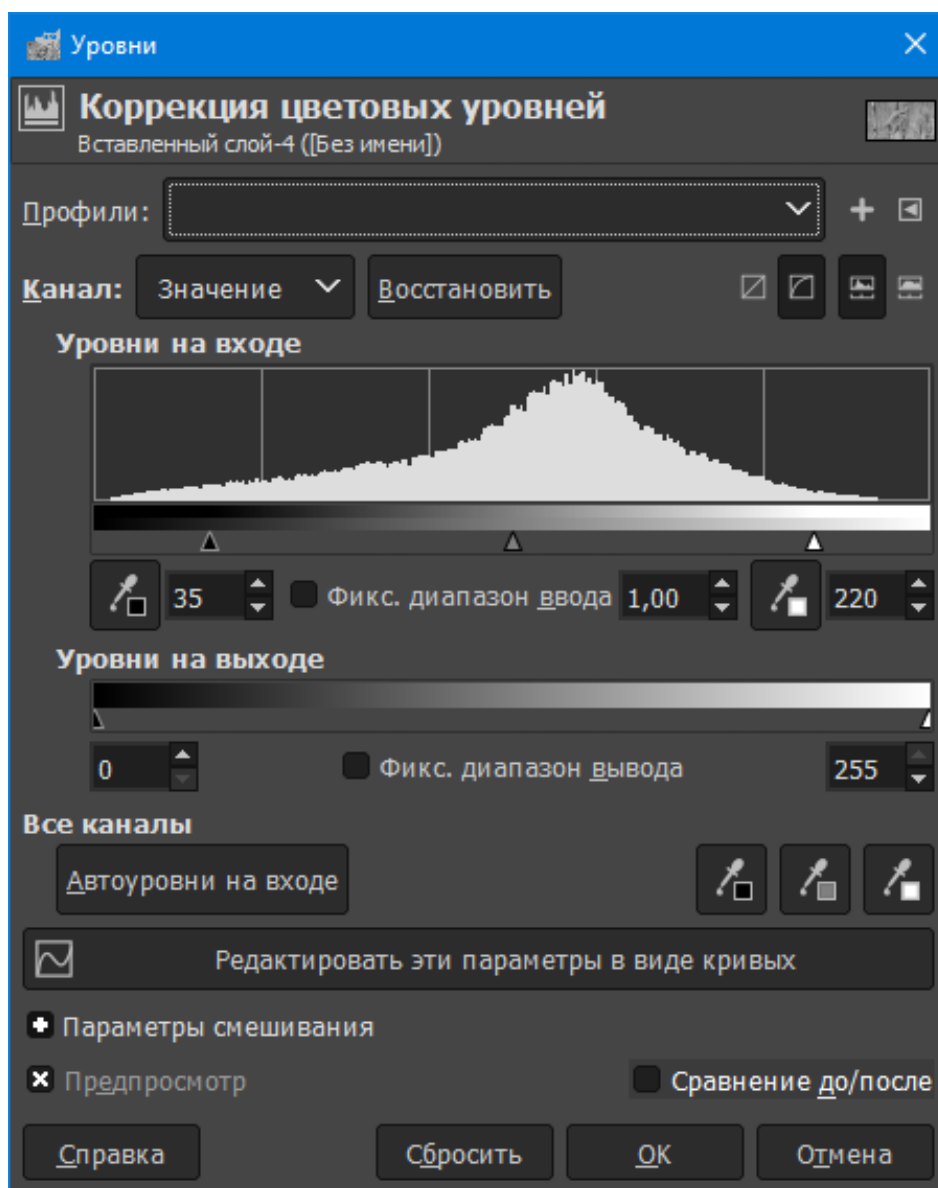


Рис. 3.7. Изменение гистограммы

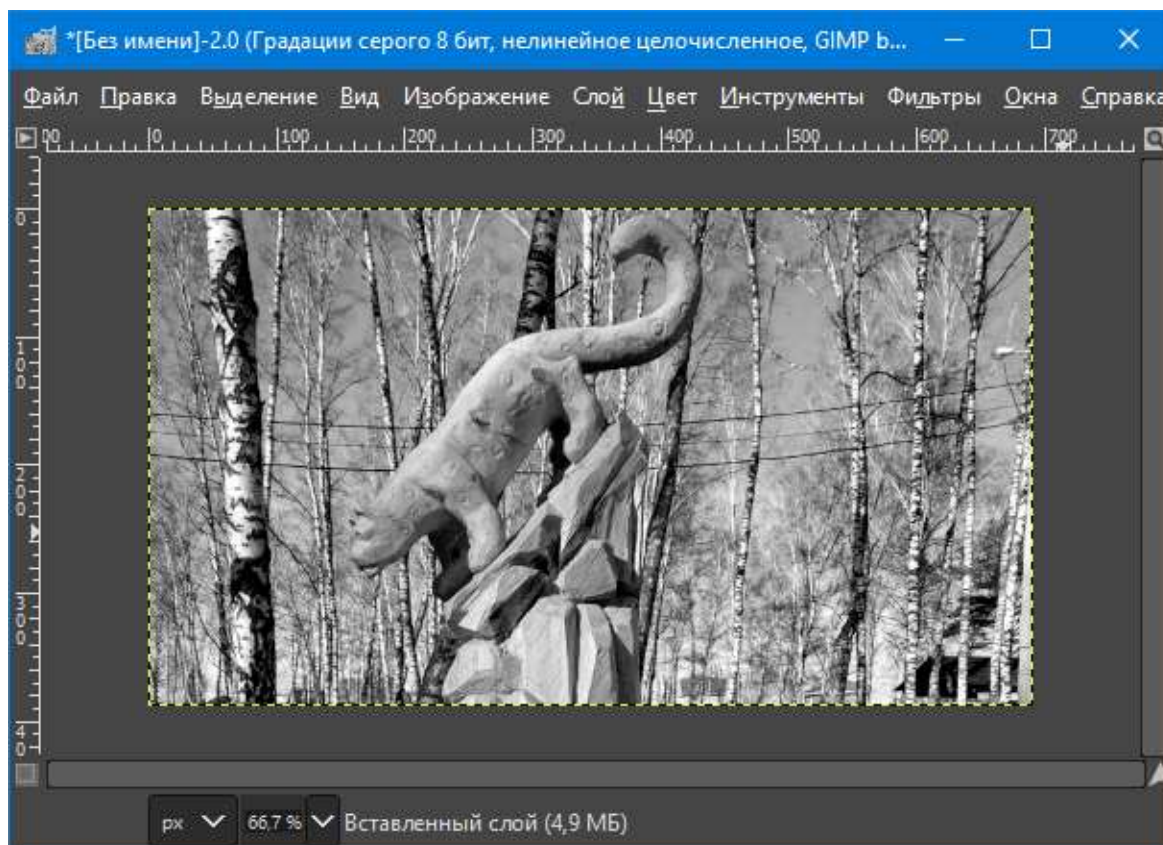


Рис. 3.8. Изображение с повышенной контрастностью

Коррекция цветковых кривых

Второй значимый элемент после гистограммы – **Кривые**, которые позволяют управлять функцией яркости и контрастности.

Рассмотрим функцию для управления яркостью и контрастностью, областью определения и значений которой являются значения цветковых компонент в модели RGB. Аргумент функции – цвет пикселя исходного изображения. Значение функции представляет собой цвет пикселя обработанного изображения. Для изменения яркости/контраста функция применяется для каждого пикселя изображения.

Если яркость и контраст изображения никак не меняются в процессе преобразования, то функция имеет график, представленный на рис. 3.9, *а*. Из рисунка видно, что функция в этом случае просто передаёт на выход значение своего аргумента.

Яркость для рассматриваемой функции представляет собой сдвиг прямой линии в вертикальном направлении. Яркость изображения увеличивается пропорционально сдвигу прямой. Если прямая сдвигается вверх (рис. 3.9, *б*), яркость изображения увеличивается, а если прямая сдвигается вниз (рис. 3.9, *в*) – уменьшается.

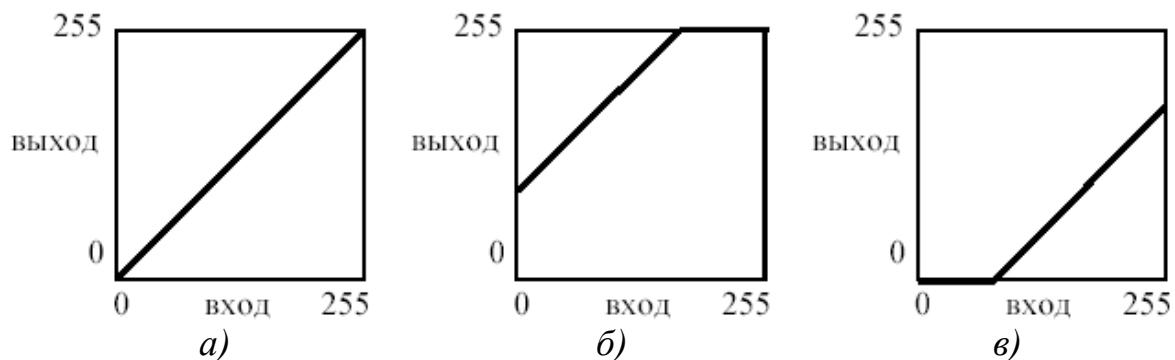


Рис. 3.9. Графики яркости

При использовании преобразования контраста прямая линия меняет свой наклон. При увеличении контраста изображения (рис. 3.10, а) наклон прямой увеличивается, при уменьшении контраста – уменьшается (рис. 3.10, б). При этом сдвиг прямой в горизонтальном направлении означает, что помимо контраста изменяется и яркость изображения.

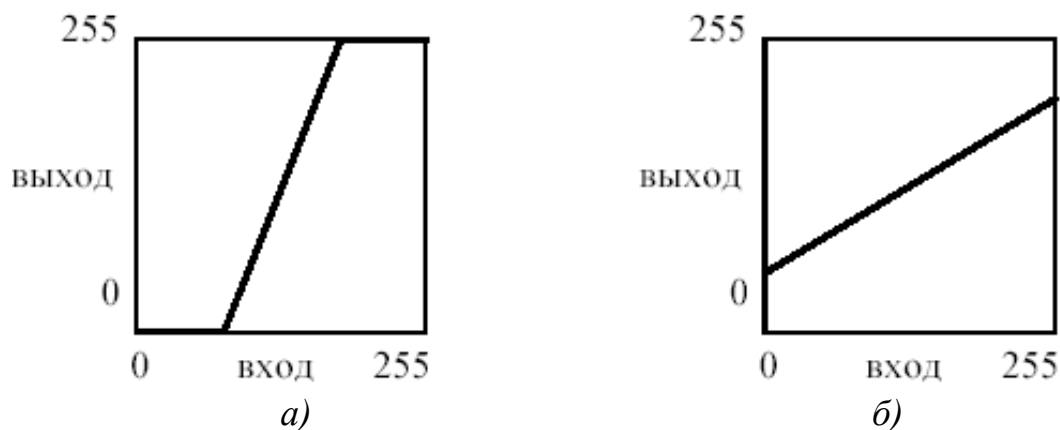


Рис. 3.10. Графики контрастности

Комбинации наклона и сдвига прямой позволяют одновременно изменять и яркость, и контраст изображения. Например, на рис. 3.11 представлен график функции, усиливающей контраст и увеличивающей яркость изображения.

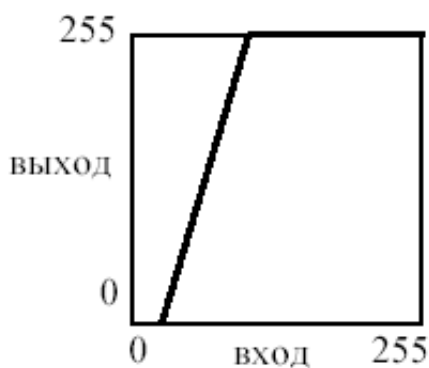


Рис. 3.11. Увеличение яркости и контрастности

Преобразование яркости/контраста может быть применено и к отдельным компонентам модели RGB, например к компоненту красного цвета. Тогда яркость/контраст будут изменяться только для красного компонента, а для других компонент они останутся неизменными. Более того, можно задавать различные преобразования яркости/контраста одновременно для каждого компонента модели RGB.

Фильтры

Фильтр – специальный вид инструмента, который берёт входной слой или изображение, применяет к нему математический алгоритм и возвращает измененный слой или изображение в новом формате. Фильтры позволяют накладывать на изображение различные эффекты, например размытие, резкость, деформацию, шум и т. д.

Для работы с фильтрами в GIMP выделены специальное меню **Фильтры** и диалоговые окна для задания их параметров.

Фильтры размытия. Это набор фильтров, которые тем или иным способом размывают изображение или его часть. Тем не менее цвета необработанной области могут попасть в размытую область. Ниже приведены иллюстрации к действию каждого из фильтров размывания, которые помогут выбрать фильтр, оптимально подходящий для той или иной задачи. Разумеется, это всего лишь примеры, поскольку почти для каждого фильтра можно изменить тип размывания и силу действия эффекта.

На рис. 3.12 приведены примеры использования различных фильтров размытия.



Рис. 3.12. Примеры использования фильтров размытия:
а) исходное изображение; б) размывание по Гауссу; в) пикселизация;
г) размывание в движении

Фильтры улучшения. Среди фильтров улучшения можно выделить фильтр повышения резкости, удаления пятен и штрихов и самое главное – удаления эффекта красных глаз на цифровых фотографиях. Для использования последнего рекомендуется сначала выделить область с эффектом красных глаз и далее применить фильтр, меняя пороговое значение в диалоговом окне.

Фильтры искажения. Фильтры искажения преобразуют изображение разными способами, такими как имитация ветра, ряби или волн на воде, загнутая страница, искажения оптики и т. д.

Фильтры Свет и тень. Делятся на три группы:

- фильтры световых эффектов рисуют разные эффекты освещения изображения;
- фильтры для создания разного рода теней. Необходимо отметить, что описанный в лабораторной работе № 2 способ получения тени через работу со слоями более гибок и позволяет получать более сложные тени, например с изгибом на полу и стене;
- фильтры эффекта стекла искажают изображение так, как будто на него смотрят сквозь линзу или стеклянные блоки.

Фильтры выделения края ищут границы между разными цветами, находя контуры объектов. Фильтры этой группы могут использоваться для различных художественных целей.

Фильтры имитации создают эффекты, присущие различным стилям живописи: кубизму, живописи маслом, картине на холсте или плетённой поверхности и т. д.

Фильтры визуализации. Большинство фильтров в GIMP работает над слоем, изменяя его, но фильтры в группе «Визуализация» отличаются тем, что они создают текстуры с нуля. Обычный результат такого фильтра – полная замена содержимого слоя. Некоторые фильтры формируют случайные или шумовые текстуры, другие – фракталы, а один (Gfig) больше напоминает общий (но ограниченный) инструмент векторной графики.

В этой же группе фильтров находятся фильтры для построения и изучения фракталов. При выборе **Фильтры → Визуализация → Природа → IFS-фрактал** вызывается подсистема построения геометрических фракталов с помощью системы итерируемых функций. При выборе фильтра **Исследователь фракталов** вызывается подсистема построения разнообразных алгебраических фракталов. Для этих фильтров GIMP содержит достаточно подробную справку с пошаговыми инструкциями.

Задание по лабораторной работе

1. Подберите исходное изображение в градациях серого цвета. Поместите изображения в отчет.
2. Переведите изображения в режим RGB-модели, выбрав **Изображение → Режим → RGB**.
3. Используя инструменты выделения, тонирования и окрашивания, сделайте изображение цветным.
4. Отрегулируйте цветовой баланс, яркость и контрастность.
5. Сохраните полученное изображение. Опишите ход работы и полученные результаты.

6. Подберите несколько разных цветных изображений и исследуйте изображения с помощью гистограммы.

7. При необходимости увеличьте контрастность изображений. Отразите результаты исследования в отчете, в котором приведите исходные, полученные изображения и их гистограммы.

8. Создайте новое изображение и залейте его градиентом от черного к белому цвету, слева направо.

9. Используя коррекцию цветových кривых, (рис. 3.13), преобразуйте изображение. Объясните результат в отчете.

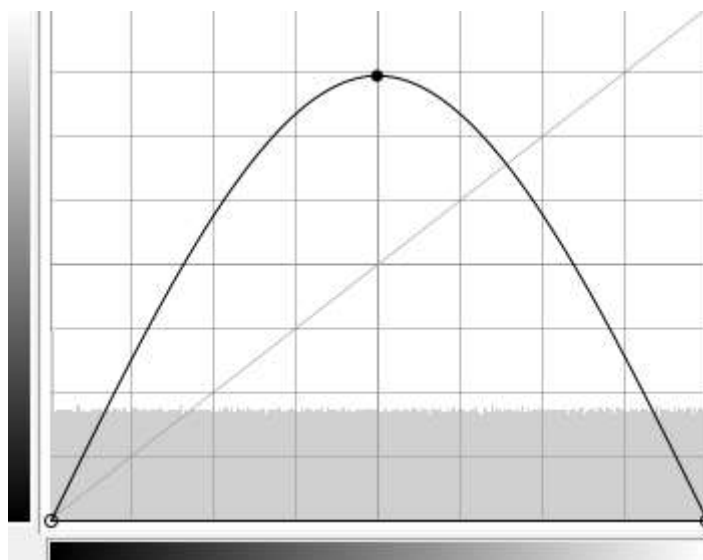


Рис. 3.13. Кривая цветокоррекции

10. Подберите фотографию с эффектом красных глаз.

11. Устраните эффект красных глаз с помощью соответствующего фильтра.

12. Кадрируйте изображение, подготовив его к печати на фотобумаге размером 10×15 см.

13. Улучшите изображение, используя гистограмму (уровни).

14. Создайте выделение по краям изображения. Для этого используйте прямоугольное выделение с закругленными краями. Для симметричности выделения установите в параметрах инструмента **Прямоугольное выделение** позицию и размер. Затем инвертируйте выделение командой **Выделение** → **Инвертировать (Ctrl+I)**.

15. Примените к выделению несколько различных фильтров для получения оригинальных рамок. Можно попробовать выбрать **Фильтры** → **Карта** → **Фрактальный след**. Наиболее удачные результаты сохраните в отчет. Используйте историю для отмены неудачных действий и новых попыток.

16. Подберите цветные изображения для дальнейших экспериментов.

17. Исследуйте группы фильтров **Искажение**, **Выделение края**, **Имитация**, применяя их в целом ко всему изображению. Наиболее интересные результаты занесите в отчет.

18. Создайте новое изображение 640×480.

19. Исследуйте на этом изображении возможности группы фильтров **Визуализация**.

20. Исследуйте возможности построения геометрических и алгебраических фракталов.

21. Исследуйте возможности построения векторных примитивов на изображении с помощью фильтра Gfig.

22. Сделайте выводы, в которых необходимо проанализировать основные возможности GIMP в сравнении с другими графическими редакторами.

Контрольные вопросы

1. Что позволяет регулировать цветовой баланс?
2. Что такое яркость и контрастность изображения?
3. Какие характеристики изображения можно изучить с помощью гистограммы?
4. Для чего нужны фильтры?

Лабораторная работа 4 ОСНОВЫ INKSCAPE

Цель: изучить основы редактора Inkscape по созданию векторных изображений.

Общие сведения

Inkscape (Инкскейп) – свободно распространяемый векторный графический редактор, поддерживающий открытый формат svg (от англ. *Scalable Vector Graphics* – масштабируемая векторная графика).

Формат svg является открытым стандартом и широко применяется в графических пакетах. Формат svg использует язык разметки XML, поэтому файлы в этом формате могут редактироваться любым текстовым или XML-редактором (отдельно от Inkscape). Помимо svg в Inkscape можно работать и с другими форматами (например, eps и png).

Основы интерфейса Inkscape

Окно Inkscape. На рис. 4.1 представлено основное окно редактора Inkscape.

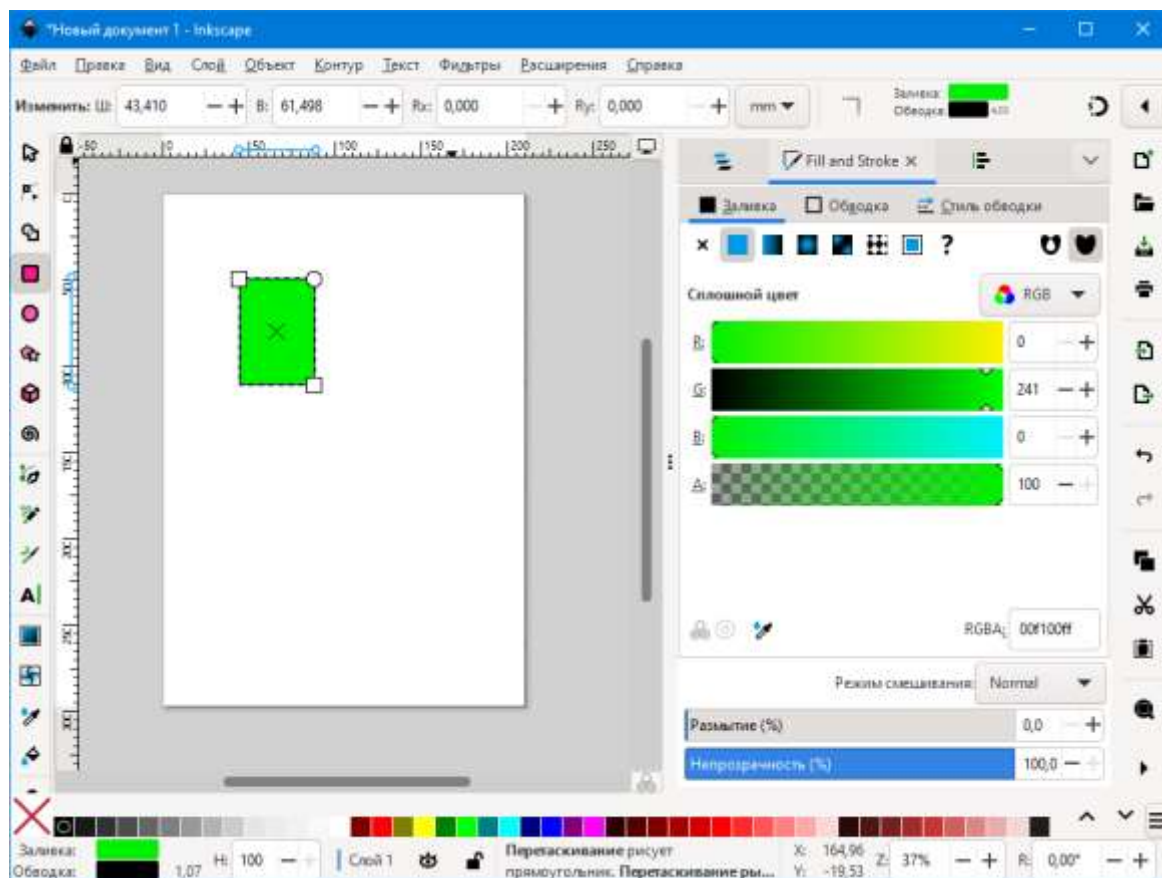



Рис. 4.1. Окно Inkscape

Сверху расположено главное меню, содержащее основные команды. Слева находится панель инструментов с командами добавления и редактирования объектов. Под главным меню располагается панель свойств для выбранного инструмента. Справа располагаются панели, содержащие дополнительные команды, в том числе команды для работы с заливкой, обводкой и документом. Внизу окна расположены палитра цветов и строка состояния.

Многие действия доступны с клавиатуры. Полный справочник по клавишам находится в меню **Справка** → **Использование клавиатуры и мыши**.

Перемещение по холсту и изменение масштаба. Основной документ, с которым работает Inkscape, называется холст. Есть множество способов перемещаться по холсту. Для перемещения при помощи клавиатуры используйте **Ctrl+стрелки**. Также можно передвигаться, зажав его поверхность средней клавишей мыши или при помощи полос прокрутки.

Изменять масштаб можно также с клавиатуры кнопками – или +. Кроме этого, существует специальный инструмент для масштабирования  , которым можно увеличивать только необходимую выделенную область.

Инструменты Inkscape. Панель со значками в левой части окна представляет инструменты Inkscape для рисования и редактирования. В верхней части окна (под меню) находится **Панель управления** с основными командными кнопками и панель **Параметры инструментов** (чуть ниже панели управления), содержащая параметры, специфичные для каждого инструмента. Строка состояния (внизу окна) будет показывать полезные подсказки в процессе работы.






Работа с документами

Для создания нового документа выберите **Файл** → **Создать** или нажмите **Ctrl+N**. Чтобы открыть существующий документ SVG, используйте **Файл** → **Открыть** (**Ctrl+O**). Для сохранения нажмите **Файл** → **Сохранить** (**Ctrl+S**) либо **Сохранить как...** (**Shift+Ctrl+S**) для сохранения файла под другим именем.

Фигуры

У Inkscape есть четыре удобных инструмента для работы с **фигурами**, каждый из которых может создавать или редактировать только собственный тип фигур. **Фигура** – это объект, изменять который можно разными уникальными для него способами, используя *узлы управления* и числовые *параметры*, которые определяют внешний вид фигуры.

Если взять, к примеру, звезду, то можно менять количество лучей, её длину, угол, округлость и т. п. При этом звезда остаётся звездой. Фигура менее свободна, чем просто контур, но часто более интересна и полезна. Всегда есть возможность преобразовать фигуру в контур (**Ctrl+Shift+C**), но обратное преобразование невозможно.

К инструментам фигур относятся инструменты для рисования прямоугольников , эллипсов , звёзд и многоугольников , для создания 3D-объектов  и спиралей . Для начала рассмотрим общие принципы работы с ними.


Основные приемы. Новая фигура создается нажатием и **перетаскиванием** по холсту курсора соответствующего инструмента. Когда фигура создана (и выбрана), она отображает свои узлы управления в виде белых меток в форме кристалла. Если фигура выбрана, то ее можно редактировать, перемещая эти узлы.

Все четыре разновидности фигур показывают свои узлы управления с любым из включенных инструментов редактирования фигур, так же как и с включенным инструментом редактирования узлов (**F2**). В момент наведения курсора мыши на один из узлов он (курсор) сообщает в строке состояния, что сделает этот узел при его смещении или нажатии по нему с разными модификаторами.

Кроме того, каждая фигура отображает свои параметры в панели, которая находится над холстом. Обычно панель настроек инструмента содержит несколько числовых полей для ввода и кнопку сброса значений в изначальное состояние.

Любые изменения в параметрах инструмента запоминаются и используются для следующей новой фигуры. Например, если изменить количество лучей звезды, у всех последующих новых звезд будет такое же количество лучей. Более того, новые параметры фигур запоминаются глобально для каждой новой сессии работы с Inkscape.

С включенным инструментом редактирования фигуры объект можно выбрать при помощи **щелчка (мыши)**. Комбинации **Ctrl+щелчок** (выбрать одну фигуру из группы) и **Alt+щелчок** (выбрать под фигурой) работают так же, как и в инструменте выделения. **Esc** сбрасывает все выделения.

Перемещение, изменение размера и вращение. Наиболее популярный инструмент в Inkscape – **Селектор** . Выбрать его можно щелчком по чёрной стрелке (либо нажав **F1** или пробел). Этим инструментом можно выбрать любой объект на холсте. Щёлкните мышью по любому

объекту (например, квадрату). Вокруг объекта увидите восемь стрелок. Теперь можно:

- передвигать объект (с нажатым **Ctrl** перемещения ограничиваются двумя осями: горизонтальной и вертикальной);
- менять размер объекта, потянув за любую из стрелок (с нажатым **Ctrl** сохраняются пропорции оригинала);

Щёлкните мышью по прямоугольнику ещё раз – направление стрелок изменится. Теперь можно:

- поворачивать объект, потянув за угловые стрелки (с нажатым **Ctrl** объект будет поворачиваться шагами по 15°; сместив крестик, сместите центр вращения);
- перекашивать (наклонять) объект, двигая неугловые стрелки (с нажатым **Ctrl** перекашивание будет производиться с шагом в 15°).

В этом режиме (режиме выделения объектов) также можно менять размеры и расположение выделения на холсте, используя поля сверху.

Изменение формы при помощи клавиш. Одна из особенностей Inkscape, отличающая его от большинства других редакторов векторной графики, – удобное управление с клавиатуры. Трудно найти команду или действие, которые было бы невозможно выполнить с клавиатуры, и изменение формы объектов – не исключение.

Можете использовать клавиатуру для перемещения объектов (клавиши-стрелки), изменения размера (клавиши < и >) и вращения (клавиши [и]). По умолчанию шаг перемещения и смены размера равен двум пикселям. С нажатой клавишей **Shift** это значение увеличивается в 10 раз (и становится равным 20 пикселям). Клавиши **Ctrl+>** и **Ctrl+<** увеличивают или уменьшают объект на 200 или 50 % от оригинала соответственно. С нажатой клавишей **Ctrl** вращение будет выполняться с шагом в 90° вместо 15°.

Наиболее удобны пиксельные изменения формы, производимые с нажатой клавишей **Alt** и клавишами изменения форм. Например, **Alt+стрелки** будут двигать выбранное на 1 пиксель *данного масштаба* (то есть на один пиксель экрана, не путайте с пикселем, который является SVG-единицей длины и отличается от пикселя масштаба). Это означает, что если пользователь увеличил масштаб, то **Alt+стрелка** даст *меньшее* смещение от абсолютного измерения, что по-прежнему будет выглядеть как смещение на пиксель на экране. Это даёт возможность точно разместить объект, увеличивая или уменьшая масштаб.

Схожим образом **Alt+>** и **Alt+<** изменяют размер на один пиксель, а **Alt+[** и **Alt+]** вращают объект на один пиксель.

Выделение нескольких объектов. Можно выбрать любое количество объектов одновременно, нажав **Shift+щелчок** на желаемые объекты. Также можно выбрать объекты рамкой выделения – так называемым резиновым выделением (рамка выделения появляется тогда, когда выделение начинается с пустого места, а с нажатой клавишей **Shift** рамка выделения появится и над объектом).

Каждый выделенный объект отображается с пунктирной рамкой вокруг него. Благодаря этой рамке просто определить, какой объект выделен, а какой нет. Например, если выбрать оба эллипса и прямоугольник под ними, то без пунктирной рамки будет сложно понять, выделены эллипсы или нет.

Shift+щелчок на выделенном объекте исключает его из общего выделения. Попробуйте для практики выбрать три объекта сверху, а после этого, используя **Shift+щелчок**, исключите эллипсы, оставив выделенным только прямоугольник.

Нажатие **Esc** сбросит все выделения. **Ctrl+A** выделяет все объекты в пределах активного слоя (если ранее не созданы слои, то это равносильно выделению всех объектов документа).

Группировка. Несколько объектов могут быть объединены в группу. При перемещении и трансформации группа ведёт себя так же, как и обычный объект.

Для создания группы нужно выбрать один или более объектов и выбрать **Объект → Сгруппировать (Ctrl+G)**. Разгруппировать их можно, нажав **Ctrl+U** или нажав в меню **Объект → Разгруппировать** и предварительно выбрав группу. Сами по себе группы могут быть сгруппированы и как одиночные объекты. Подобная поэтапная группировка может быть сколько угодно сложной. При этом следует помнить, что **Ctrl+U** разгруппирует только последнюю группировку. Группировка может выполняться последовательно: для этого пользователь несколько раз группирует объекты в группы и далее объединяет группы вместе. Для полной разгруппировки таких сложных групп надо последовательно несколько раз нажать **Ctrl+U**.

Очень удобно то, что не нужно разбивать группу для редактирования отдельных объектов. Нажав **Ctrl+щелчок** на объект, в группе можно получить доступ к функциям редактирования отдельного объекта в группе. Таким же образом работает комбинация **Shift+Ctrl+щелчок**, позволяющая редактировать несколько объектов независимо от группы.

Заливка и обводка. Множество функций Inkscape доступны через диалоги (субменю). Вероятно, самый простой способ заполнить объект каким-либо цветом – это выбрать палитру внизу окна.

Но более грамотным способом будет выбор диалога **Заливка и обводка...**, который отображается справа от основного пространства для рисования (рис. 4.2). Если этот диалог закрыт, его можно активировать командой **Объект → Заливка и обводка (Shift+Ctrl+F)**.

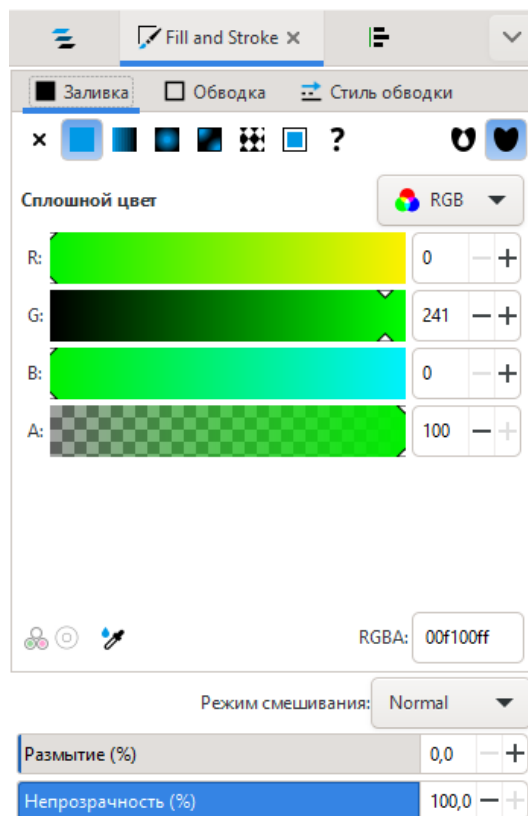


Рис. 4.2. Диалог **Заливка и обводка**


Диалог содержит три вкладки: **Заливка**, **Обводка** и **Стиль обводки**. Вкладка **Заливка** позволит изменить заполнение выбранного объекта (или объектов). Используя кнопки под вкладкой, можно выбрать тип заливки, включая режим **Нет заливки** (кнопка со знаком X), режим **Сплошной цвет**, режимы **Линейный градиент** или **Радиальный градиент**.

Чуть ниже расположены кнопки-варианты выбора цвета. Каждый вариант имеет свою вкладку: **RGB**, **CMYK**, **HSL** и **ColorWheel**. Вероятно, самым удобным вариантом является режим **HSL** с развернутым **ColorWheel**. В нём можно выбрать тон цвета, вращая треугольник, а затем подобрать насыщенность и яркость в самом треугольнике. Все варианты выбора цвета имеют возможность менять альфа-канал (прозрачность) выбранного объекта (или объектов).

Каждый раз при выборе объекта вкладка **Заливка и обводка...** показывает текущее значение для данного объекта (для нескольких одновременно выбранных объектов, вкладка цвета показывает их *усреднённый* цвет).

При использовании вкладки **Обводка** можно задать границу объекта, установить ее цвет или прозрачность.

Последняя вкладка **Стиль обводки** позволяет изменить толщину и другие параметры границы.

Ещё один способ изменить цвет объекта – использовать инструмент **Пипетка**  (F7). Щёлкните мышью в любой части рисунка, и полученный цвет будет присвоен выбранному до этого объекту (**Shift+щелчок** присвоит цвет границе объекта).

Дублирование, выравнивание, распределение. Одним из наиболее распространённых действий является дублирование объекта (**Ctrl+D**). Дублирование размещает дубликат над оригиналом и делает его выделенным так, что можно переместить его в сторону при помощи мыши или клавиш со стрелками.

Выделенные несколько объектов можно выравнивать с помощью специального диалога **Align and Distribute (Выровнять и распределить)** (рис. 4.3), вызываемого сочетанием клавиш **Ctrl+Shift+A**.

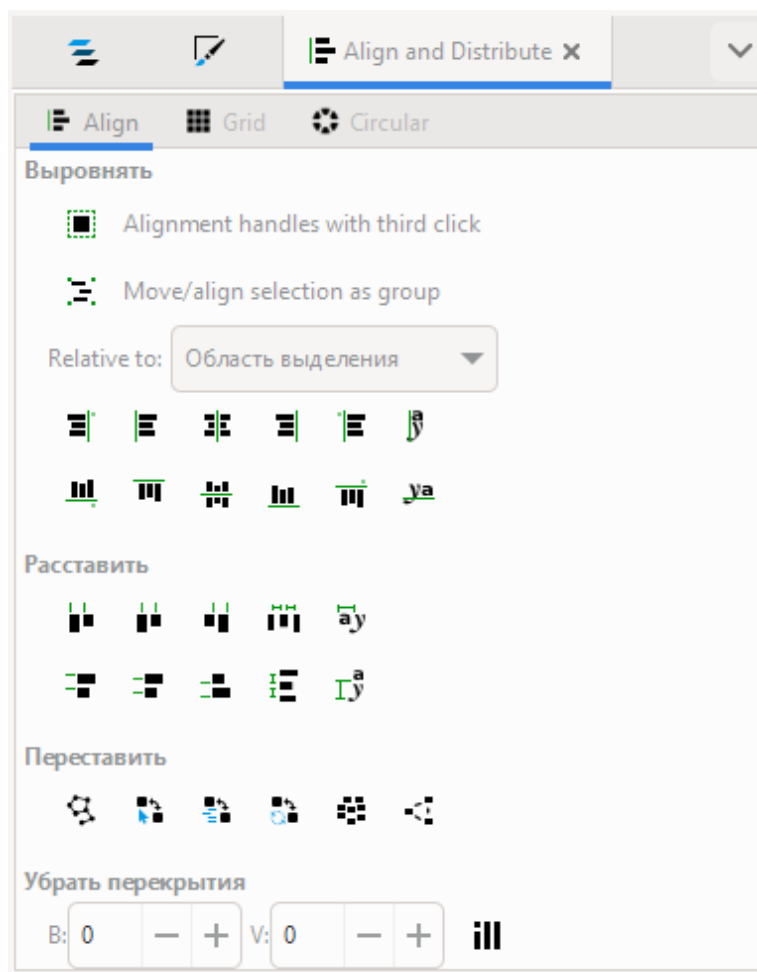


Рис. 4.3. Диалог «Align and Distribute»

Z-порядок. Термин «Z-порядок» (порядок по оси Z) относится к перекрыванию объектами друг друга на рисунке. Иначе говоря, Z-порядок определяет, какой объект находится выше и закрывает собой другие. Две команды в меню **Объект** → **Поднять на передний план** (кнопка **Home**) и **Объект** → **Опустить на задний план** (кнопка **End**), переместят выбранный объект в самую верхнюю или самую нижнюю позицию по оси Z данного слоя. Две другие команды – **Поднять (PgUp)** и **Опустить (PgDn)** – приподнимут или опустят выбранный объект (или объекты), *но только на один уровень* относительно других невыделенных объектов по оси Z (считаются только объекты, перекрывающие выделенные; если выделение ничем не перекрывается, действие «Поднять» и «Опустить» будет ставить его в самую верхнюю или самую нижнюю позицию соответственно).

Задание по лабораторной работе

1. Запустите Inkscape.
2. Создайте новый лист размером «По умолчанию».
3. Создайте прямоугольник, эллипс, звезду, спираль, кривую Безье и текст.
4. Задайте различную заливку (в том числе и градиенты) и обводку (в том числе и пунктиры) созданным фигурам.
5. Экспортируйте изображение в растр и добавьте его в отчет.
6. Сохраните документ в формате SVG.
7. Откройте сохраненный *.SVG в текстовом редакторе (notepad).
8. Перенесите в отчет XML теги каждой фигуры. Опишите ключевые параметры каждой из фигур, влияющие на их положение, форму и цвет.
9. Проанализируйте результаты и сделайте выводы. При этом необходимо оценить качество полученных изображений и размеры файлов.

Контрольные вопросы

1. В каком собственном формате хранит изображения Inkscape?
2. Что такое фигура в терминах редактора Inkscape?
3. Для чего нужны заливка и обводка при работе с фигурами?
4. Что такое Z-порядок?

Лабораторная работа 5 СОЗДАНИЕ ВЕКТОРНОГО ЛОГОТИПА


Цель: создать векторный логотип в редактора Inkscape на основе образца.

Общие сведения

Размещение текста вдоль контура

В Inkscape существует возможность размещения текста вдоль каких-либо линий, в том числе и кривых, и вдоль любой сложной фигуры, созданной путем преобразования фигуры в кривую. Сложная фигура, сформированная путем слияния, автоматически становится кривой. Располагать текст вдоль фигур (прямоугольников, эллипсов и т. д.) нельзя, поэтому предварительно такие фигуры нужно обязательно преобразовывать в кривые. Например, это можно выполнить командой **Контур** → **Оконтурить объект**.

Алгоритм данной операции следующий:

- 1) используйте инструмент **A** и создайте текстовый объект;
- 2) нажмите на кнопку для создания нужной фигуры. Нарисуйте фигуру;
- 3) выберите оба объекта с помощью кнопки . Для этого необходимо последовательно щелкнуть по ним с нажатой клавишей **Shift**;
- 4) в меню выберите **Текст** → **Разместить по контуру**.

Выполнение логических операций над фигурами

Над несколькими выделенными объектами возможно выполнение логических операций: сложение, вычитание, пересечение, исключающее ИЛИ и т. п. Все эти команды собраны в меню **Контур** (рис. 5.1).








	Сумма	Ctrl++
	Разность	Ctrl+-
	Пересечение	Ctrl+*
	Исключающее ИЛИ	Ctrl+^
	Разделить	Ctrl+/ /
	Разрезать контур	Ctrl+Alt+/ /

Рис. 5.1. Команды для выполнения логических операций

Сумма. Эта команда сливает два объекта и делает из них один. Может применяться к любому количеству объектов. Полученный объект всегда использует настройки стиля (заливки и штриха) нижнего объекта.


Алгоритм выполнения операции:

- 1) создайте две фигуры с помощью инструментов;
- 2) выберите оба объекта с помощью кнопки . Чтобы выбрать оба объекта, необходимо, последовательно щелкнуть по ним с нажатой клавишей **Shift**;

- 3) в меню выберите **Контур → Сумма**.

Разность. Эта команда удаляет у объекта области, перекрываемые вышележащим выделенным объектом (или объектами). Может применяться только к двум объектам.


Алгоритм выполнения операции:

- 1) создайте две фигуры с помощью инструментов;
- 2) выберите оба объекта с помощью кнопки . Для этого необходимо последовательно щелкнуть по ним с нажатой клавишей **Shift**;

- 3) в меню выберите **Контур → Разность**.

Пересечение. Эта команда позволяет создать новый объект, который включает в себя область пересечения двух или более объектов, выделенных перед слиянием. Если выделено более двух объектов, то необходимо, чтобы во всех выделенных объектах был пересекающийся сегмент. Если такого сегмента нет, то команда игнорируется. Пересекающийся фрагмент может быть только один. Полученный объект всегда использует настройки стиля (заливки и штриха) нижнего объекта.


Алгоритм выполнения операции:

- 1) создайте две фигуры с помощью инструментов;
- 2) выберите оба объекта с помощью кнопки . Для этого необходимо последовательно щелкнуть по ним с нажатой клавишей **Shift**;

- 3) в меню выберите **Контур → Пересечение**.

Исключающее ИЛИ. Эта команда делает пересекающиеся области прозрачными. Может применяться только к двум объектам.


Алгоритм выполнения операции:

- 1) создайте две фигуры с помощью инструментов;
- 2) выберите оба объекта с помощью кнопки . Для этого необходимо последовательно щелкнуть по ним с нажатой клавишей **Shift**;

- 3) в меню выберите **Контур → Исключающее ИЛИ**.


Разделить. Данная команда соединяет в себя команды **Разность** и **Пересечение**. Число выделенных объектов не может быть больше двух.

Алгоритм выполнения операции:

- 1) создайте две фигуры с помощью инструментов;
- 2) выберите оба объекта с помощью кнопки . Для этого необходимо последовательно щелкнуть по ним с нажатой клавишей **Shift**;

- 3) в меню выберите **Контур → Разделить**.

Работа с узлами

Инструменты для управления узлами. Инструмент для управления узлами  предназначен для редактирования и выбора узлов. Чтобы активизировать инструмент, можно использовать боковое окно панели инструментов (этот инструмент расположен в нем вторым сверху) или нажать клавишу **F2**. При этом изменится состав кнопок контекстной панели инструментов (рис. 5.2).

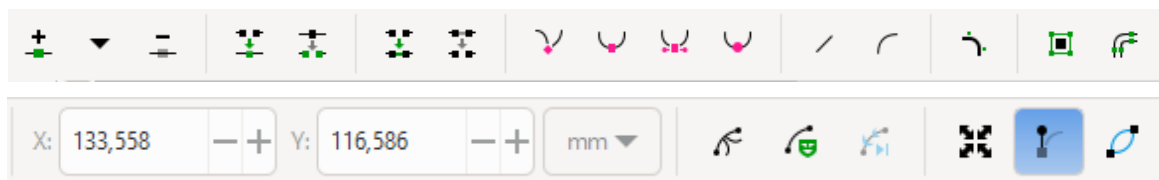



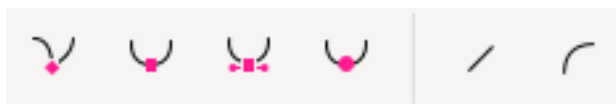
Рис. 5.2. Панель инструментов управления узлами


Для того чтобы на активной фигуре выделить узлы, необходимо выбрать кнопку **Оконтурить объект**  или нажать комбинацию клавиш **Shift+Ctrl+C**.

Если инструмент управления узлами активен, то по контуру активной фигуры могут отображаться узлы в виде квадратов. Для того чтобы выбрать узел, просто щелкните по нему. Щелчок по контуру между узлами выбирает оба этих узла. Если нужно добавить или удалить узел, удерживайте при щелчке мыши клавишу **Shift**. Также добавить узел можно с помощью двойного щелчка левой кнопкой мыши по контуру активной фигуры.

Если необходимо выбрать все узлы фигуры, то можно воспользоваться комбинацией клавиш **Ctrl+A**. В этом случае будут выбраны все узлы, кроме вложенных. Для того чтобы выбрать все узлы, включая вложенные, следует использовать комбинацию клавиш **Ctrl+Alt+A**.

Следующие кнопки позволят изменить форму:



Кнопка  показывает или скрывает обрисовку контура.

Перемещение узлов. Перемещать узлы можно с помощью мыши обычным образом. Если удерживать при перемещении узла клавишу **Ctrl**, то узел сможет перемещаться только по вертикали или по горизонтали. Удерживая комбинацию клавиш **Ctrl+Alt**, можно перемещать узел строго вдоль его направляющей.

Перемещать узлы можно также с помощью стрелок на клавиатуре. В этом случае объект будет перемещаться с шагом 2 пикселя (по умолчанию, но эту настройку можно изменить).

После того как узел выбран, если это возможно для данного вида узла, то будет отображаться его направляющая. Расположение направляющей также влияет на вид кривой этого узла. С помощью маркеров на концах направляющей можно изменять ее длину и вращать ее. Удерживая при вращении направляющей клавишу **Ctrl**, можно вращать ее с интервалом 15°. Удержание клавиши **Alt** блокирует изменение длины направляющей. Клавиша **Shift** позволяет перемещать обе направляющих.

Горячие клавиши:

- **Shift** – удерживайте клавишу для выбора нескольких узлов;
- двойной щелчок или **Shift+Alt** на пути или кривой – создает новый узел, не изменяя форму контура;
- **Tab** – выбирает следующий узел;
- **Shift+Tab** – выбирает предыдущий узел;
- **Ctrl+Alt** – удаляет узел;
- **Ctrl+щелчок** указателем мыши на маркер направляющей – обновляет ее длину. Чтобы вытянуть направляющую назад из узла, используйте клавишу **Shift**.



Задание по лабораторной работе

Используя описанные в данном пособии приемы, создайте векторное изображение по образцу из списка ниже, соответствующему номеру варианта (взять у преподавателя).

При выполнении действий сохраняйте промежуточные результаты в отчет с комментариями.

Проанализируйте проделанную работу и сделайте выводы.

Варианты заданий

	
1	2



Red Hat

3



4



5



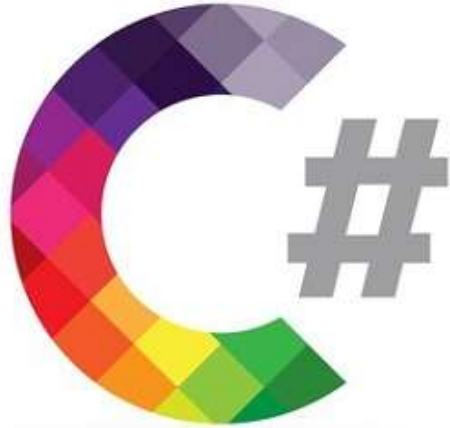




6



7



8

	
<p>9</p>	<p>10</p>
	
<p>11</p>	<p>12</p>
	
<p>13</p>	<p>14</p>



15



16



17



18



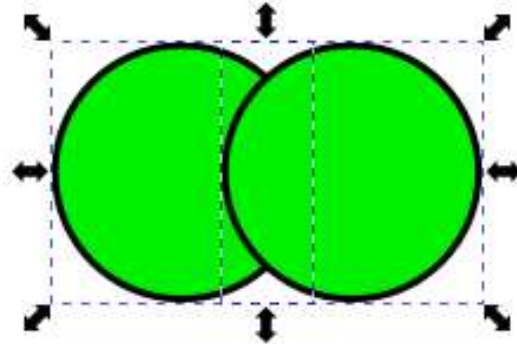
19



20

Контрольные вопросы

1. Какую последовательность действий необходимо выполнить для размещения текста по контуру?
2. Какие логические операции можно выполнять над фигурами?
3. Какой результат будет при выполнении операции **Пересечение** над двумя кругами?



4. Какое действие необходимо выполнить, чтобы начать работу с узлами фигуры?

Лабораторная работа 6 ПРОГРАММИРОВАНИЕ ГРАФИКИ

Цель: в соответствии с первой образовательной траекторией изучить возможности C# по созданию простейших графических изображений либо в соответствии с второй образовательной траекторией изучить возможности JavaScript; написать и отладить программу построения на экране различных графических примитивов.

Общие сведения

Создание векторного изображения в Visual Studio

Сообщение WM_PAINT. Прежде чем приступить к описанию способов рисования в окнах, применяемых приложениями .NET Frameworks, расскажем о том, как это делают «классические» приложения Microsoft Windows.

ОС Microsoft Windows следит за перемещением и изменением размера окон и при необходимости извещает приложения, о том, что им следует перерисовать содержимое окна. Для извещения в очередь приложения записывается сообщение с идентификатором **WM_PAINT**. Получив такое сообщение, функция окна должна выполнить перерисовку всего окна или его части, в зависимости от дополнительных данных, полученных вместе с сообщением **WM_PAINT**.

Все функции, выводящие какие-либо графические примитивы или изображения в окно, должны выполняться при обработке события **WM_PAINT**. Приложение должно быть сделано таким образом, чтобы в любой момент времени при поступлении сообщения **WM_PAINT** функция вывода окна могла перерисовать все окно или любую его часть, заданную своими координатами.

Последнее нетрудно сделать, если приложение будет хранить где-нибудь в памяти свое текущее состояние, пользуясь которым функция окна сможет перерисовать окно в любой момент времени.

Здесь не имеется в виду, что приложение должно хранить образ окна в виде графического изображения и восстанавливать его при необходимости, хотя это и можно сделать. Приложение должно хранить информацию, на основании которой оно может в любой момент времени перерисовать окно.

Сообщение **WM_PAINT** передается функции окна, если стала видна область окна, скрытая раньше другими окнами, если пользователь изменил размер окна или выполнил операцию прокрутки изображения в окне.

Приложение может передать функции окна сообщение **WM_PAINT** явным образом, вызывая функции программного интерфейса Win32 API, такие как **UpdateWindow**, **InvalidateRect** или **InvalidateRgn**.

Иногда ОС Microsoft Windows может сама восстановить содержимое окна, не посылая сообщение **WM_PAINT**. Например, при перемещении курсора мыши или значка свернутого приложения ОС восстанавливает содержимое окна. Если же ОС не может восстановить окно, функция окна получает от ОС сообщение **WM_PAINT** и перерисовывает окно самостоятельно.

Событие Paint. Для форм класса **System.Windows.Forms** предусмотрен удобный объектно-ориентированный способ, позволяющий приложению при необходимости перерисовывать окно формы в любой момент времени. Когда вся клиентская область окна формы или часть этой области требует перерисовки, форме передается событие **Paint** (рис. 6.1). Все, что требуется от программиста, это создать обработчик данного события, наполнив его необходимой функциональностью.

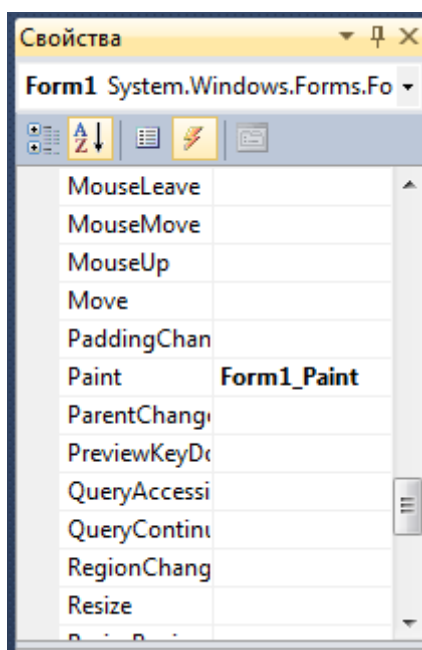


Рис. 6.1. Создание обработчика события *Paint*

Объект Graphics для рисования. Перед тем как рисовать линии, фигуры, отображать текст и выводить изображения, необходимо создать объект **Graphics**. Объект **Graphics** представляет собой поверхность рисования GDI¹ и используется для создания графических изображений.

¹ GDI (Graphics Device Interface) – один из основных компонентов операционной системы, предоставляющий средства для обработки двумерной графики, рисунков и для решения типографских задач.

Исходя из вышесказанного, перечислим два этапа работы с графикой:

- 1) создание объекта `Graphics`;
- 2) использование объекта `Graphics` для рисования линий и фигур, для отображения текста или изображения и управления ими.

Существует несколько способов создания объектов **Graphics**. Одним из самых используемых является получение ссылки на объект **Graphics** через объект **PaintEventArgs** при обработке события **Paint** формы или элемента управления:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics; // Объявляется объект Graphics
    // Далее следует код рисования
}
```

Методы и свойства класса Graphics. Имена большого количества методов, определенных в классе `Graphics`, начинается с префиксов `Draw*` и `Fill*`. Первые предназначены для рисования текста, линий и незакрашенных фигур (таких, например, как прямоугольные рамки), а вторые – для рисования закрашенных геометрических фигур. Рассмотрим применение только самых важных из этих методов, а полную информацию можно в документации на сайте learn.microsoft.com.

Метод **DrawLine** рисует линию, соединяющую две точки с заданными координатами². Ниже приведены прототипы различных перегруженных версий этого метода:

```
public void DrawLine(Pen, Point, Point);
public void DrawLine(Pen, PointF, PointF);
public void DrawLine(Pen, int, int, int, int);
public void DrawLine(Pen, float, float, float, float);
```

Первый параметр задает инструмент для рисования линии – перо. Перья создаются как объекты класса **Pen**, например:

```
Pen p = new Pen(Brushes.Black, 2);
```

Здесь создано черное перо толщиной 2 пикселя. При этом пользователь может выбрать его цвет, толщину и тип линии, а также другие атрибуты.

Остальные параметры перегруженных методов `DrawLine` задают координаты соединяемых точек. Эти координаты могут быть определены

² Начало координат находится в левом верхнем углу элемента, на котором происходит прорисовка. В нашем случае – в левом верхнем углу окна. Ось y направлена вниз, ось x – направо.

как объекты класса **Point** и **PointF**, а также в виде целых чисел и чисел с плавающей десятичной точкой.

В классах **Point** и **PointF** определены свойства **X** и **Y**, соответствующие координатам точки по горизонтальной и вертикальной осям. При этом в классе **Point** эти свойства имеют целочисленные значения, а в классе **PointF** – значения с плавающей десятичной точкой.

Третий и четвертый варианты метода **DrawLine** позволяют задавать координаты соединяемых точек в виде двух пар чисел. Первая пара определяет координаты первой точки по горизонтальной и вертикальной осям, а вторая – координаты второй точки по этим же осям. Разница между третьим методом и четвертым заключается в использовании координат различных типов (целочисленных **int** и с плавающей десятичной точкой **float**).

Чтобы испытать метод **DrawLine** в работе, разработайте приложение DrawLineApp (аналогично предыдущему примеру). В этом приложении создайте следующий обработчик события **Paint**:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.Clear(Color.White);
    for (int i = 0; i < 50; i++)
    {
        g.DrawLine(new Pen(Brushes.Black, 2), 10, 4*i+20, 200, 4*i+20);
    }
}
```

Для того чтобы нарисовать 50 горизонтальных линий, вызывается метод **DrawLine** в цикле (рис. 6.2).

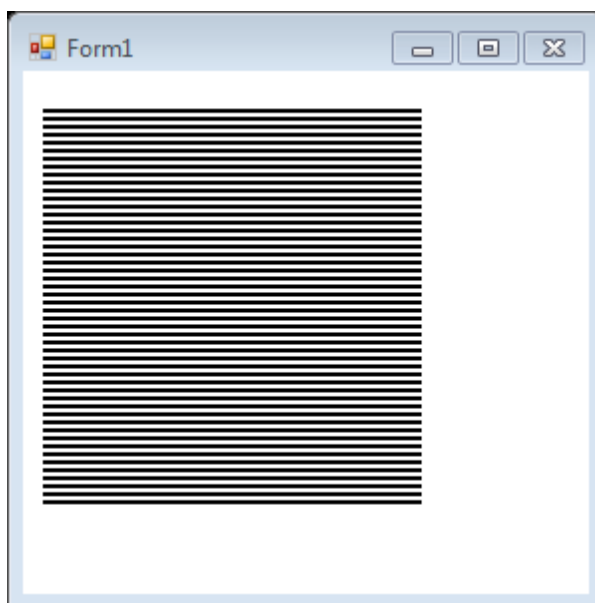


Рис. 6.2. Пример использования *DrawLine*

Вызвав один раз метод `DrawLines`, можно нарисовать сразу несколько прямых линий, соединенных между собой. Иными словами, метод **DrawLines** позволяет соединить между собой несколько точек. Координаты этих точек по горизонтальной и вертикальной оси передаются методу через массив класса **Point** или **PointF**:

```
public void DrawLines(Pen, Point[]);
public void DrawLines(Pen, PointF[]);
```

Для демонстрации возможностей метода **DrawLines** разработайте приложение. Создайте кисть **pen** для рисования линий:

```
Pen pen = new Pen(Color.Black, 2);
```

а также массив точек **points**, которые нужно соединить линиями:

```
for (int i = 0; i < 20; i++)
{
    int xPos;
    if (i % 2 == 0)
    {
        xPos = 10;
    }
    else
    {
        xPos = 400;
    }
    points[i] = new Point(xPos, 10 * i);
}
```

Код будет выглядеть следующим образом:

```
public partial class Form1 : Form
{
    Point[] points = new Point[50];
    Pen pen = new Pen(Color.Black, 2);

    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawLines(pen, points);
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        for (int i = 0; i < 20; i++)
```

```

    {
    int xPos;
    if (i % 2 == 0)
    {
        xPos = 10;
    }
    else
    {
        xPos = 400;
    }
    points[i] = new Point(xPos, 10 * i);
    }
}

```

Результат приведен на рис. 6.3.

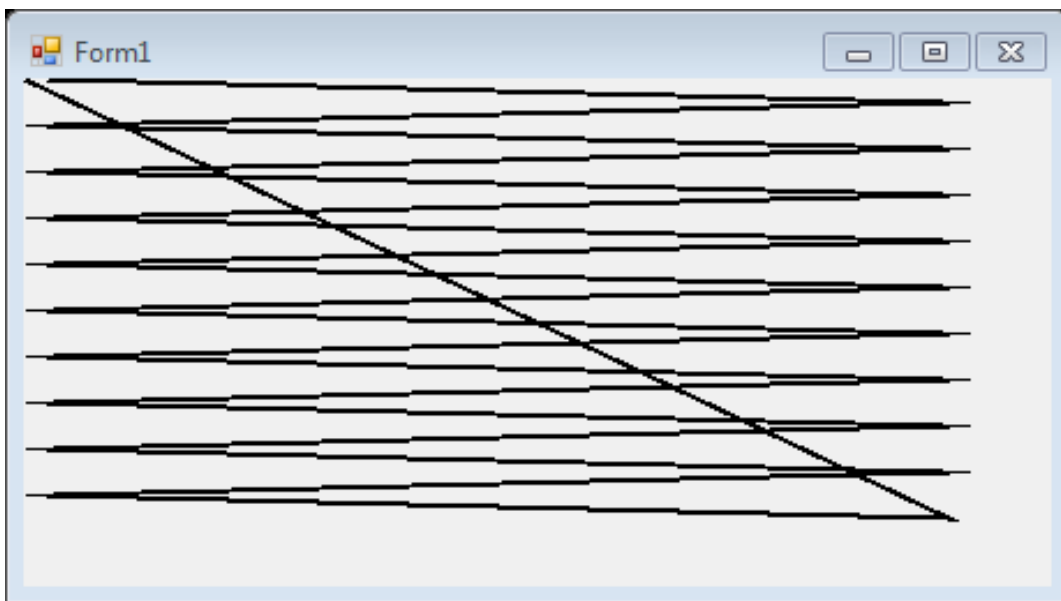


Рис. 6.3. Пример использования массива точек

Для прорисовки прямоугольников можно использовать метод **DrawRectangle(Pen, int, int, int, int)**; В качестве первого параметра передается перо класса Pen. Остальные параметры задают расположение и размеры прямоугольника.

Для прорисовки многоугольников можно использовать метод **DrawPolygon(Pen, Point[])**;

Метод **DrawEllipse** рисует эллипс, вписанный в прямоугольную область, расположение и размеры которой передаются ему в качестве параметров. При помощи метода **DrawArc** программа может нарисовать сегмент эллипса. Сегмент задается при помощи координат прямоугольной области, в которую вписан эллипс, а также двух углов, отсчитываемых в направлении против часовой стрелки. Первый угол Angle1 определяет расположение одного конца сегмента, а второй Angle2 – другого (рис. 6.4).

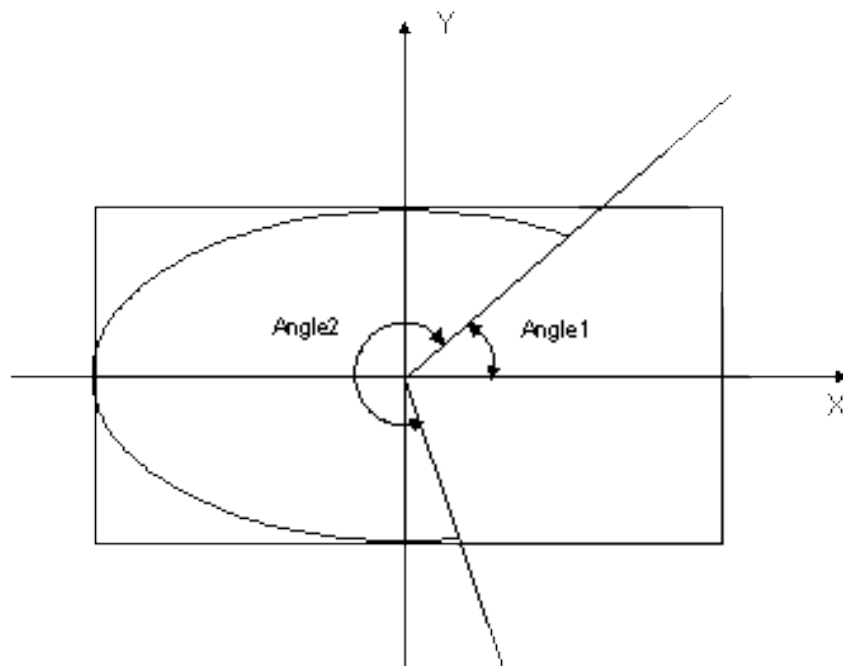


Рис. 6.4. Углы и прямоугольник, задающие сегмент эллипса

В классе **Graphics** определен ряд методов, предназначенных для рисования закрашенных фигур. Имена некоторых из этих методов, имеющих префикс **Fill**: **FillRectangle** (рисование закрашенного прямоугольника), **FillRectangles** (рисование множества закрашенных многоугольников), **FillPolygon** (рисование закрашенного многоугольника), **FillEllipse** (рисование закрашенного эллипса), **FillPie** (рисование закрашенного сегмента эллипса), **FillClosedCurve** (рисование закрашенного сплайна), **FillRegion** (рисование закрашенной области типа **Region**).

Есть два отличия методов с префиксом **Fill** от одноименных методов с префиксом **Draw**. Прежде всего методы с префиксом **Fill** рисуют закрашенные фигуры, а методы с префиксом **Draw** – незакрашенные. Кроме этого, в качестве первого параметра методам с префиксом **Fill** передается не перо класса **Pen**, а кисть класса **SolidBrush**. Ниже приведем пример, выводящий закрашенный прямоугольник:

```
SolidBrush B=new SolidBrush(Color.DeepPink);
g.FillRectangle(B,0,0,100,100);
```

Очевидно, что платформа **.Net** содержит большое число классов со многими методами и свойствами. Нет смысла описывать все классы, методы в каком-либо учебнике или в данном пособии, поскольку по любому методу или классу можно получить MSDN-справку. Для этого необходимо набрать наименование метода в среде **Visual Studio** и нажать клавишу **F1**. Также при наборе метода в редакторе кода среда показывает краткую справку о передаваемых параметрах.

Рисование на холсте в JavaScript

Лучший способ для создания растровых рисунков из векторных примитивов на JavaScript является использование **холста (canvas)**. Холст – это один элемент DOM, в котором находится изображение. Он предоставляет API для рисования форм на том месте, которое занимает элемент. Сам элемент описывается с помощью тега `<canvas>`:

```
<canvas id="my_canvas" width="200" height="200"></canvas>
```

Далее с помощью JavaScript надо обратиться к этому полю. Сначала получим ссылку на него через **getElementById**, а затем создадим объект, в котором будет находиться так называемый «контекст», через методы которого будем рисовать:

```
<script>
  const canvas = document.getElementById('my_canvas');
  const context = canvas.getContext('2d');
</script>
```

Обращаясь к методам контекста (в представленном примере объект `context`), можно рисовать различные фигуры в пределах области холста. Например, создадим красный прямоугольник, перечеркнутый черной линией:

```
context.fillStyle = "red";
context.fillRect(10, 10, 100, 50);
context.beginPath();
context.moveTo(10, 10);
context.lineTo(110, 60);
context.stroke();
```

Следуя рекомендациям по JavaScript, разделим HTML-код и подключаемый к нему javascript-код на два файла:

HTML-код:

```
<html>
<body>
  <canvas id="my_canvas" width="1000" height="1000"></canvas>
  <!-- подключаем файл my.js -->
  <script src="my.js"></script>
  <script language='JavaScript'>
    InitCanvas();
  </script>
</body>
</html>
```

JavaScript-код файла my.js:

```
function InitCanvas()
{
    /** @type {HTMLCanvasElement} */
    const canvas = document.getElementById('my_canvas');
    /** @type {CanvasRenderingContext2D} */
    const context = canvas.getContext("2d");
    context.fillStyle = "red";
    context.fillRect(10, 10, 100, 50);
    context.beginPath();
    context.moveTo(10, 10);
    context.lineTo(110, 60);
    context.stroke();
}
```

Если пользователь работает с MS Visual Studio Code, то строка `/** @type {HTMLCanvasElement} */` позволит подключить IntelliSense подсказки для холста при написании кода, строка `/** @type {CanvasRenderingContext2D} */` позволяет подключать IntelliSense автозаполнение для контекста холста.

Графические примитивы на холсте. Элемент **canvas** содержит начало координат в левом верхнем углу. Ось *y* направлена вниз, ось *x* – вправо. Контекст холста поддерживает ряд методов для рисования прямоугольников:

- **fillRect(x, y, width, height)** – рисование заполненного прямоугольника;
- **strokeRect(x, y, width, height)** – рисование прямоугольного контура;
- **clearRect(x, y, width, height)** – очистка прямоугольной области.

Каждая из приведённых функций принимает несколько параметров:

- **x, y** – устанавливают положение верхнего левого угла прямоугольника в **canvas** (относительно начала координат);
- **width** (ширина) и **height** (высота) – определяют размеры прямоугольника.

Для рисования дуг и окружностей, используются методы **arc()** и **arcTo()**:

- **arc(x, y, radius, startAngle, endAngle, anticlockwise)** – рисует дугу с центром в точке **(x, y)** радиусом **radius**, начиная с угла **startAngle** и заканчивая в **endAngle** в направлении против часовой стрелки **anticlockwise** (по умолчанию по ходу движения часовой стрелки);
- **arcTo(x1, y1, x2, y2, radius)** – рисует дугу с заданными контрольными точками и радиусом, соединяя эти точки прямой линией.

Рассмотрим детальнее метод `arc()`, который имеет пять параметров: `x` и `y` – это координаты центра окружности, в которой должна быть нарисована дуга, `radius` – радиус, углы `startAngle` и `endAngle` определяют начальную и конечную точки дуги в радианах вдоль кривой окружности. Отсчёт происходит от оси `x`. Параметр `anticlockwise` – логическое значение. Если `anticlockwise` установлено в `true`, то дуга рисуется против хода часовой стрелки; иначе – по ходу часовой стрелки.

Использование путей на холсте. Геометрический **путь** (`path`) представляет набор линий, окружностей, прямоугольников и других более мелких деталей, необходимых для построения сложной фигуры. Иногда путь переводят на русский язык как **контур**. Описание пути начинается вызовом метода `beginPath()` и обычно заканчивается вызовом методами отрисовки `fill()` и/или `stroke()`. Между этими двумя методами помещаются методы, формирующие сам путь. Например, нарисуем квадрат:

```
context.beginPath();
context.moveTo(100, 100);
context.lineTo(200, 100);
context.lineTo(200, 200);
context.lineTo(100, 200);
context.closePath();
context.stroke();
```

Метод `moveTo()` устанавливает начальную точку пера, с которой начинается путь. Методы `lineTo()` добавляют в путь отрезок от текущей точки следующей. Конец отрезка становится текущей точкой пера. В конце можно замкнуть путь методом `closePath()`, не добавляя последнее ребро квадрата.

Следующим типом доступных путей являются кривые Безье, доступные в кубическом и квадратичном вариантах:

- `quadraticCurveTo(cp1x, cp1y, x, y)` – рисует квадратичную кривую Безье с текущей позиции пера в конечную точку с координатами `x` и `y`, используя контрольную точку с координатами `cp1x` и `cp1y`;
- `bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)` – рисует кубическую кривую Безье с текущей позиции пера в конечную точку с координатами `x` и `y`, используя две контрольные точки с координатами `(cp1x, cp1y)` и `(cp2x, cp2y)`.

Закраска и стиль линий пути. Для оформления стиля линий контекст холста имеет свойство `lineWidth`, задающее толщину линий в пикселях и `strokeStyle` и определяющее цвет линий, или градиент, или шаблон линии. Добавление таких строк кода позволит нарисовать красный квадрат с толщиной линии пять пикселей:

```
context.strokeStyle = '#ff0000';
context.lineWidth = 5;
```

Для задания различных пунктирных, штрихпунктирных линий пути используется свойство **setLineDash**:

```
// Черточка 5 пикселей и после пробел 3 пикселя
context.setLineDash([5, 3]);
```

Для закраски области внутри пути необходимо применять после завершения построения пути метод **fill()**. По аналогии со свойством **strokeStyle** для задания цвета заполнения фигуры используется свойство **fillStyle**. Цвет при этом может представляться в шестнадцатеричном виде для модели RGB либо по именам HTML -цветов. Приведём пример js-файла:

```
function InitCanvas()
{
    // Настройка автозаполнения в VS Code для холста
    /** @type {HTMLCanvasElement} */
    // Получение холста
    const canvas = document.getElementById('my_canvas');

    // Настройка автозаполнения в VS Code для контекста
    /** @type {CanvasRenderingContext2D} */
    // Получение контекста
    const context = canvas.getContext("2d");

    // Создание пути
    context.beginPath();

    // Задание цвета границы
    context.strokeStyle = '#ff0000'; // красный цвет

    // Задание толщины границы
    context.lineWidth = 5;

    // Задание цвета заполнения
    context.fillStyle = 'HotPink';

    // Задание стиля линии в виде пунктира
    // Черточка 5 пикселей и пробел после 3 пикселя
    context.setLineDash([5, 3]);

    // Установка начальной точки
    context.moveTo(100, 100);
    // Задание сторон квадрата
    context.lineTo(200, 100);
    context.lineTo(200, 200);
    context.lineTo(100, 200);
    // Закрытие пути и создание последней стороны квадрата
    context.closePath();
}
```

```

    // Прорисовка области внутри пути
    context.fill();
    // Прорисовка границы пути
    context.stroke();
}

```

Задание градиента и шаблона заполнения. Градиент – способ заполнения с плавным переходом цвета. Для работы с градиентом должен быть создан один из объектов:

- конический градиент – создается методом **createConicGradient**;
- линейный градиент – создается методом **createLinearGradient**;
- радиальный градиент – создается методом **createRadialGradient**.

Для любого созданного градиента необходимо добавить несколько цветов, между которыми будет происходить переход. Для добавления таких цветов используется метод **addColorStop(offset, color)**, где **color** представляет значение цвета, а **offset** – число от 0 до 1 включительно, обозначающее положение точки цвета на шкале цветов. 0 представляет начало градиента, а 1 – конец. Например, при добавлении таких точек цвета на шкалу цветов

```

addColorStop("0", "red");
addColorStop("0.5", "green");
addColorStop("1.0", "blue");

```

красный цвет будет находиться в начале градиента, зеленый – в середине, синий – в конце шкалы цветов.

Рассмотрим пример, в котором рядом с предыдущим розовым квадратом нарисуем квадрат с вертикальным градиентом по границе в цветах модели CMY, и горизонтальным градиентом в цветах модели RGB для заполнения квадрата:

```

context.beginPath();

context.lineWidth = 5;
context.setLineDash([]); // необходимо сбросить стиль пунктира

context.createConicGradient
context.createLinearGradient
context.createRadialGradient

// Создание градиента для границы
const board_gradient=context.createLinearGradient(0, 100, 0, 200);
board_gradient.addColorStop('0', 'cyan');
board_gradient.addColorStop('0.5', 'magenta');
board_gradient.addColorStop('1.0', 'yellow');

// Создание градиента для заполнения
const fill_gradient=context.createLinearGradient(300, 0, 400, 0);

```

```

fill_gradient.addColorStop("0", "red");
fill_gradient.addColorStop("0.5", "green");
fill_gradient.addColorStop("1.0", "blue");

// Задание цвета границы
context.strokeStyle = board_gradient;
// Задание цвета заполнения
context.fillStyle = fill_gradient;

context.moveTo(300, 100);
context.lineTo(400, 100);
context.lineTo(400, 200);
context.lineTo(300, 200);
context.closePath();
context.fill();
context.stroke();

```

Результат представлен на рис. 6.5.

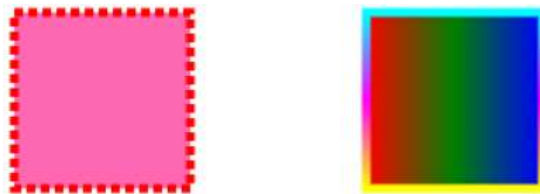


Рис. 6.5. Результат отрисовки квадратов с разными заливками

Подобно созданию объектов для заполнения градиентом можно создать объект **CanvasPattern** для заполнения графического примитива каким-либо изображением или видео. Для этого используется метод **createPattern()**. Этот метод повторяет указанный элемент (изображение, видео или холст) в указанном направлении:

```

// загрузка изображения с html-страницы
const smile = document.getElementById('smile_icon');

// создание шаблона заполнения из изображения
const pat = context.createPattern(smile, "repeat");
// создание нового пути
context.beginPath();
// создание прямоугольника
context.rect(500, 100, 100, 100);
// установка шаблона заполнения для заливки области
context.fillStyle = pat;
// прорисовка области
context.fill();

```

Чтобы данный пример работал, на web-странице должен находиться тег **img** с id:

```
<img id = "smile_icon" src = "SmileIcon.gif" style="display: none">
```

Кроме заполнения по шаблону и градиентом для контекста можно установить свойство **globalAlpha** в диапазоне от 0 до 1 для задания полупрозрачности при наложении объектов друг на друга:

```
context.globalAlpha = 0.5; // прозрачность для наложения
```

2D-преобразования на холсте. Для созданных фигур на холсте можно применять различные 2D-преобразования:

- метод **rotate(angle)** обеспечивает поворот на угол **angle**, заданный в радианах. Поворот происходит относительно начала координат;
- метод **translate(x, y)** используется для переноса в новое место. Первый параметр указывает на смещение по оси X, а второй параметр – по оси Y;
- метод **scale(xScale, yScale)** предназначен для масштабирования. Параметр **xScale** указывает на масштабирование по оси X, а **yScale** – по оси Y.

Рассмотрим пример использования поворота и переноса.

```
function draw()
{
  /** @type {HTMLCanvasElement} */
  const canvas = document.getElementById('my_canvas');
  /** @type {CanvasRenderingContext2D} */
  const ctx = canvas.getContext("2d");

  // перенос на 75 единиц влево и вправо
  // теперь точка 75, 75 является центром координат
  // и можно производить повороты вокруг этого центра
  ctx.translate(75, 75);
  // внешний цикл для создания 6 рядов кружков
  for (let i=1; i<6; i++)
  {
    // задание случайного цвета для ряда
    let r = Math.floor(Math.random() * 255);
    let g = Math.floor(Math.random() * 255);
    let b = Math.floor(Math.random() * 255);
    ctx.fillStyle = 'rgb(' + r + ', ' + g + ', ' + b + ')';

    // цикл для прорисовки кружков в одном ряду
    for (let j=0; j<i*6; j++)
    {
      // поворот вокруг точки 75,75 на 60 градусов
      ctx.rotate(Math.PI*2/(i*6));
      ctx.beginPath();
      // прорисовка кружочка
      ctx.arc(0, i*12.5, 5, 0, Math.PI*2, true);
      ctx.fill();
    }
  }
}
```

Данный пример позволяет построить несколько окружностей из кружков случайных цветов (рис. 6.6).



Рис. 6.6. Результат работы программы с преобразованиями поворота

Часто совместно с методами преобразований на плоскости используются методы контекста **save()**, помещающий текущее состояние контекста (толщина линий, цвет заполнения, угол поворота, прочее) в стек, и **restore()**, извлекающий текущее состояние контекста холста из стека.

Кроме этого, можно определить свойство **transform** через метод **setTransform (m11, m12, m21, m22, dx, dy)**, что позволяет напрямую задавать матрицу двумерных преобразований размером три на три:

$$\begin{pmatrix} m11 & m12 & dx \\ m21 & m22 & dy \\ 0 & 0 & 1 \end{pmatrix}$$

Фактически это трансформированная матрица преобразований, рассмотренная в лекциях. При таком описании матрица преобразований умножается на матрицу вершин. Каждая вершина представляет собой вектор столбец:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Для более детального ознакомления с 2D-преобразованиями на холсте рекомендуется ознакомиться с примерами по ссылке: <https://bucephalus.org/text/CanvasHandbook/CanvasHandbook.html#transformations>. Также для получения более подробной справки о холсте рекомендуется обратиться к документации – https://developer.mozilla.org/ru/docs/Web/API/Canvas_API.

Задание по лабораторной работе

Разработайте программу для вывода изображения на форму, если выбрана образовательная траектория с C# и Visual Studio, либо в окне браузера, если выбрана траектория на JavaScript.

Варианты заданий

1. Напишите программу, рисующую поздравительную открытку. Фоном открытки являются флажки и колпаки в виде конусов. Для прорисовки фона создайте функции, которые вызовите в цикле со случайными параметрами. Параметрами функций должны быть местоположение, размер, цвет. На открытке напишите фразу «С днем рождения». Буквы заполните шаблоном из файла. Используйте градиентную заливку.

2. Реализуйте программу, рисующую морской пейзаж. Волны и лучи от солнца выводятся в цикле. Для прорисовки солнца используйте радиальный градиент. Для рисования волны задайте шаблон заполнения. В волнах нарисуйте белоснежный пароход. Для отрисовки дыма из трубы также примените цикл и случайные числа.

3. Создайте программу для рисования новогодней ели, украшенной игрушками. Используйте градиент для заливки фона. Игрушки на елки развешиваются случайным образом. Используйте шаблон заполнения с изображениями из файлов.

4. Реализуйте программу для рисования снеговика с подарками. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. Для прорисовки подарков напишите функцию, которую вызовите из цикла.

5. Создайте программу для рисования персонажа из мультфильма. Используйте градиент для заливки фона. Задайте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. Напишите функцию для рисования цветков и вызовите эту функцию в цикле для заполнения фона цветами.

6. Разработайте программу, рисующую дорогу, заполненную автомобилями. Для этого напишите функцию рисования автомобиля. Параметрами функции должны быть размер автомобиля, его местоположение и цвет деталей. Используйте градиентную заливку и шаблоны заполнения областей изображениями из файлов.

7. Напишите программу, рисующую набор детских пирамидок. Для этого напишите функцию рисования пирамидки. Параметрами функции должны быть размер пирамидки, ее местоположение и угол поворота. Используйте градиентную заливку и шаблоны заполнения областей изображениями из файлов.

8. Разработайте программу, рисующую Московский Кремль с башнями и стеной. Создайте людей перед Кремлем в цикле, используя случайные числа. Стена Кремля также формируется в цикле. Используйте разные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов.

9. Напишите программу, рисующую набор предметов (тыквы и летучие мыши) для Хэллоуина случайным образом. Для этого напишите функции рисования этих предметов. Параметрами функции должны быть размер предметов, их местоположение и угол поворота. Используйте градиентную заливку и шаблоны заполнения областей изображениями из файлов.

10. Создайте программу для рисования звездного неба с различными летающими объектами: звезды, спутник, самолет, ракета, НЛО. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. Напишите функцию для рисования звезд различного размера и цвета. Некоторые звезды объедините в созвездия.

11. Разработайте программу, рисующую натюрморт. Для этого напишите функции рисования предметов различного размера и цвета. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов.

12. Реализуйте программу, создающую фон из камней различного размера и цвета в случайных местах. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. На данном фоне нарисуйте любое животное.

13. Создайте программу для рисования шкафа, заполненного книгами. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. Напишите функцию для рисования книг различного размера и цвета. Кроме книг на полках шкафа разместите другие объекты.

14. Разработайте программу, рисующую пасхальный кулич. Кулич нарисуйте на фоне из множества яиц различного размера и цвета в случайных местах. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов.

15. Реализуйте программу, рисующую модель солнечной системы. Для прорисовки планет создайте функцию с параметрами: местоположение, цвет или шаблон заполнения, размер. Нарисуйте орбиты движения планет и их спутники. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов.

16. Реализуйте программу, рисующую весенний пейзаж со множеством птиц. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. Для прорисовки птиц создайте по крайней мере две функции в разных позах. Параметрами функций должны быть: местоположение, цвет, угол поворота. Вызовите эти функции в цикле со случайными параметрами.

17. Реализуйте программу, рисующую осенний пейзаж со множеством опадающих листьев. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов.

Для прорисовки листьев создайте функцию, которую вызовите в цикле со случайными параметрами. Параметрами функций должны быть: местоположение, размер, цвет, угол поворота.

18. Разработайте программу, рисующую новогоднюю гирлянду из лампочек на фоне фейерверков. Для прорисовки фейерверков создайте функцию, которую вызовите в цикле со случайными параметрами. Параметрами функций должны быть: местоположение, размер, цвет. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов. Гирлянды из лампочек рисуйте также в цикле.

19. Напишите программу, рисующую весенний цветок на фоне шариков. Для прорисовки шариков создайте функцию, которую вызовите в цикле со случайными параметрами. Параметрами функций должны быть: местоположение, размер, цвет. Вербочку к шарикю нарисуйте в виде кривой Безье. Используйте различные типы градиентной заливки и шаблоны заполнения областей изображениями из файлов.

20. Разработайте программу, рисующую домик на лугу. Создайте траву на лугу в цикле, используя случайные числа. Стены домика закрасьте, используя шаблон с изображением материала. Небо залейте градиентной заливкой.

Контрольные вопросы

Контрольные вопросы по первой образовательной траектории

1. Что такое сообщение **WM_PAINT** и событие **Paint**?
2. С помощью какого объекта происходит прорисовка графических примитивов в окне?
3. С помощью какого класса можно задать цвет и толщину графических примитивов?
4. С помощью какого класса можно задать цвет и стиль заполнения графических примитивов?

Контрольные вопросы по второй образовательной траектории

1. На каком элементе DOM можно создавать векторные изображения?
2. Где находится центр координат и как направлены оси на хосте?
3. Для чего можно использовать путь на холсте?
4. Как можно задать цвет графических примитивов на JavaScript?

Лабораторная работа 7 ПРОСТЕЙШАЯ АНИМАЦИЯ

Цель: в соответствии с первой образовательной траекторией изучить возможности С# по созданию простейшей анимации либо в соответствии со второй образовательной траекторией изучить возможности анимации в JavaScript; написать и отладить программу, выводящую на экран анимированное изображение.

Общие сведения

Работа с анимацией в Visual Studio

Работа с таймером. Класс для работы с таймером (Timer) формирует в приложении повторяющиеся события. События повторяются с периодичностью, указанной в миллисекундах, в свойстве **Interval**. Установка свойства **Enabled** в значение **true** запускает таймер. Каждый тик таймера порождает событие **Tick**, обработчик которого обычно и создают в приложении. В этом обработчике могут изменяться какие-либо величины и может вызываться принудительная перерисовка окна. Напоминаем, что вся отрисовка при анимации должна находиться в обработчике события **Paint**.

Создание анимации. Для создания простой анимации достаточно использовать таймер, при тике которого будут изменяться параметры изображения (например, координаты концов отрезка) и обработчики события **Paint** для рисования по новым параметрам. При таком подходе не надо заботиться об удалении старого изображения (как в идеологии MS DOS), ведь оно создается в окне заново.

В качестве примера рассмотрим код анимации секундной стрелки часов:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        //описываем переменные,
        //доступные в любом обработчике событий класса
        private int x1, y1, x2, y2, r;
```

```

private double a;
private Pen pen = new Pen(Color.DarkRed, 2);
public Form1()
{
    InitializeComponent();
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    //рисуем секундную стрелку
    g.DrawLine(pen, x1, y1, x2, y2);
}

private void Form1_Load(object sender, EventArgs e)
{
    //определяем центр экрана
    x1 = ClientSize.Width / 2;
    y1 = ClientSize.Height / 2;
    r = 150; //задаем радиус
    a = 0; //задаем угол поворота
    //определяем конец часовой стрелки с учетом центра
    x2 = x1 + (int) (r * Math.Cos(a));
    y2 = y1 - (int) (r * Math.Sin(a));
}

private void timer1_Tick(object sender, EventArgs e)
{
    {
        a -= 0.1; //уменьшаем угол на 0,1 радиан
        //определяем конец часовой стрелки с учетом центра
        x2 = x1 + (int)(r * Math.Cos(a));
        y2 = y1 - (int)(r * Math.Sin(a));
        //принудительный вызов перерисовки (Paint)
        Invalidate();
    }
}
}
}

```

Движение по траектории. Движение по траектории реализуется аналогично рассмотренному выше примеру. Для программирования движения по прямой приращиваются на определённые константы переменные, являющиеся узловыми точками (в примере переменные x_2 , y_2). Для задания более сложной траектории можно использовать различные параметрические кривые. В случае движения на плоскости обычно изменяют один параметр. Рассмотрим пример реализации движения окружности по декартову листу.

Декартов лист – плоская кривая третьего порядка, удовлетворяющая уравнению в прямоугольной системе $x^3 + y^3 = 3axy$ (рис. 7.1).

Параметр $3a$ определяется как диагональ квадрата, сторона которого равна наибольшей хорде петли.

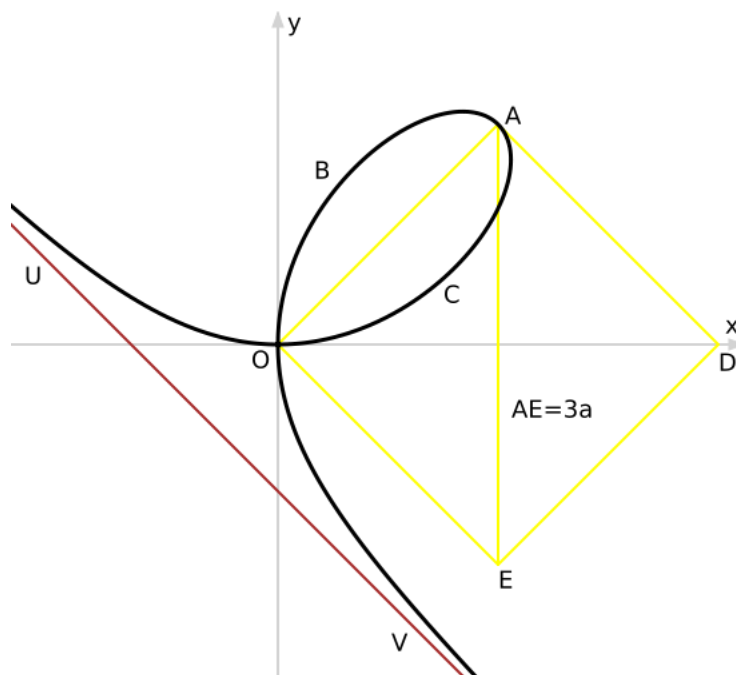


Рис. 7.1. Декартов лист

При переходе к параметрическому виду получаем:

$$\begin{cases} x = \frac{3at}{1+t^3}; \\ y = \frac{3at^2}{1+t^3}, \end{cases}$$

где $t = \operatorname{tg}\varphi$.

Программная реализация выглядит следующим образом:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        private int x1, y1, x2, y2;
        private double a, t, fi;
        private Pen pen = new Pen(Color.DarkRed, 2);
```

```

public Form1()
{
    InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e)
{
    x1 = ClientSize.Width / 2;
    y1 = ClientSize.Height / 2;
    a = 150;
    fi = -0.5;
    t = Math.Tan(fi);
    x2 = x1 + (int)((3 * a * t) / (1 + t * t * t));
    y2 = y1 - (int)((3 * a * t * t) / (1 + t * t * t));
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawEllipse(pen, x2, y2, 20, 20);
}

private void timer1_Tick(object sender, EventArgs e)
{
    fi += 0.01;
    t = Math.Tan(fi);
    x2 = x1 + (int)((3 * a * t) / (1 + t * t * t));
    y2 = y1 - (int)((3 * a * t * t) / (1 + t * t * t));
    Invalidate();
}
}
}

```

Описание ряда интересных кривых для создания траектории движения можно найти в «Википедии» в статье «Циклоидальная кривая».

Анимация в JavaScript

Для создания анимации в JavaScript существует несколько способов:

- анимация с использованием **setInterval()**;
- анимация с использованием **setTimeout()**;
- анимация с использованием CSS-переходов;
- анимация с изменением свойств элемента;
- анимация с использованием библиотек;
- анимация через **requestAnimationFrame()**.

Анимация методами **setInterval()** и **setTimeout()** использует таймер и имеет ряд недостатков: например, при прорисовке сложных сцен может снижаться fps. Более правильный способ создания анимации – применение метода **requestAnimationFrame()**. В качестве параметра он принимает функцию обратного вызова (англ. – *callback*), которая и осуществляет прорисовку очередного кадра анимации. Метод **requestAnimationFrame()** не вызывает

функцию отрисовки немедленно, а просит браузер запустить эту функцию в наиболее подходящий момент времени. Обычно таким моментом является момент окончания прорисовки предыдущего кадра, то есть момент, когда аппаратные компоненты освободились для рендеринга нового кадра.

Рассмотрим пример для рисования движения кружочка по синусоиде:

```
// подсказываем для VS Code тип переменной canvas, для IntelliSense
/** @type {HTMLCanvasElement} */
let canvas = null;
// подсказываем для VS code тип переменной context, для IntelliSense
/** @type {CanvasRenderingContext2D} */
let ctx = null;

// описываем глобальные переменные для блока
let x = 50; // значение x в пикселях
let alfa = 0; // значение угла альфа в радианах

// функция реализующая бесконечный цикл прорисовки кадров
function AnimLoop()
{
  // clearRect очистка холста при необходимости
  // ctx.clearRect(0, 0, canvas.width, canvas.height);

  // создание кружочка на синусоиде
  ctx.beginPath();
  ctx.arc(x, 100 - Math.sin(alfa)*50, 3, 0, 2*Math.PI, false);
  ctx.stroke();
  ctx.fill();
  ctx.closePath();

  // изменение x и угла альфа
  x++;
  alfa += 0.05;

  // запрос на прорисовку следующего кадра
  requestAnimationFrame(AnimLoop);
}
function InitCanvas()
{
  canvas = document.getElementById('my_canvas');
  ctx = canvas.getContext('2d');

  // установка текущих параметров границы и заполнения для контекста
  ctx.lineWidth = 2;
  ctx.strokeStyle = "green";
  ctx.fillStyle = "yellow";

  // запрос на прорисовку первого кадра
  let requestId = requestAnimationFrame(AnimLoop);
}
```

При использовании строчки `ctx.clearRect(0, 0, canvas.width, canvas.height);` в каждом кадре будет очищаться холст и можно увидеть движение кружка по синусоиде. Если не использовать очистку холста, то увидим все предыдущие кадры, то есть визуализируем таким образом траекторию движения.

Для создания движения объекта по параметрическим кривым предлагается ознакомиться с анимацией движения по декартовому листу выше в разделе «Работа с анимацией в JavaScript».

Рассмотрим немного более сложный пример анимации из документации, реализующий движение планеты Земля вокруг Солнца и движения Луны вокруг Земли:

```
/** @type {HTMLCanvasElement} */
let canvas = null;
/** @type {CanvasRenderingContext2D} */
let ctx = null;

// создание изображений для Солнца, Луны и Земли
const sun = new Image();
const moon = new Image();
const earth = new Image();
// функции инициализации: загрузки изображений из файлов
// и вызов цикла прорисовки для первого кадра
function init() {
  sun.src = "canvas_sun.png";
  moon.src = "canvas_moon.png";
  earth.src = "canvas_earth.png";
  window.requestAnimationFrame(draw);
}
// функция прорисовки очередного кадра
function draw()
{
  // получение холста и его контекста
  canvas = document.getElementById('my_canvas');
  ctx = canvas.getContext('2d');

  // задание свойства вывода перекрывающихся изображений
  ctx.globalCompositeOperation = "destination-over";
  ctx.clearRect(0, 0, 300, 300); // очистка холста

  // задание стиля заполнения с полупрозрачностью 0.4
  ctx.fillStyle = "rgba(0, 0, 0, 0.4)";
  // задание стиля обводки
  ctx.strokeStyle = "rgba(0, 153, 255, 0.4)";

  ctx.save(); // сохранение контекста
  // перенос в центр изображения солнца
  // само изображение Солнца имеет размер 300 на 300 пикселей
  ctx.translate(150, 150);
```

```

// прорисовка Земли в зависимости от времени
const time = new Date();
// поворот вокруг центра изображения Солнца (точки 150, 150)
ctx.rotate(
  ((2 * Math.PI) / 60) * time.getSeconds() +
  ((2 * Math.PI) / 60000) * time.getMilliseconds(),
);
// Прорисовка тени позади Земли
ctx.translate(105, 0);
ctx.fillRect(0, -12, 40, 24);
// Прорисовка самой Земли
ctx.drawImage(earth, -12, -12);

// Прорисовка Луны
ctx.save(); // сохранение контекста
// поворот Луны вокруг Земли
ctx.rotate(
  ((2 * Math.PI) / 6) * time.getSeconds() +
  ((2 * Math.PI) / 6000) * time.getMilliseconds(),
);
ctx.translate(0, 28.5);
// Прорисовка Луны со сдвигом -3.5, -3.5
// сдвиг делается, чтобы центр Луны совпал с точкой
// на орбите вокруг Земли. Значение 3.5 определены как
// половина размера изображения Луны (7 на 7 пикселей)
ctx.drawImage(moon, -3.5, -3.5);
ctx.restore();//восстановление контекста после поворота Луны

ctx.restore();//восстановление контекста после поворота Земли

// Прорисовка орбиты Земли как окружности
ctx.beginPath();
ctx.arc(150, 150, 105, 0, Math.PI * 2, false);
ctx.stroke();

// Прорисовка Солнца и пространства вокруг
ctx.drawImage(sun, 0, 0, 300, 300);

// вызов отрисовки следующего кадра
window.requestAnimationFrame(draw);
}

init();

```

Атрибут **GlobalCompositeOperation** задает то, как прорисовываются изображения, наложенные друг на друга. При этом одно изображение называется **целевым (Destination)** и оно описывается раньше в коде, что соответствует более низкому слою. Второе изображение является **изображением источника (Source)**, и оно приводится в коде позже и,

соответственно, занимает более высокий слой. Однако значение **destination-over** выводит сначала изображение источника, а потом целевое изображение. Поэтому порядок вывода в каждом кадре такой: сначала изображение Солнца, потом орбита Земли, далее Луна, потом сама Земля, далее тень от Земли, которая иногда перекрывает саму Луну.

Более подробную информацию о **GlobalCompositeOperation** можно найти по ссылке: https://www.w3schools.com/tags/canvas_globalcompositeoperation.asp. Рекомендуется также изучить другие примеры анимации из документации (https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Basic_animations) и возможности изменения скорости движения по траектории в соответствии с кривыми (<https://learn.javascript.ru/js-animation>).

Задание по лабораторной работе

Изучите с помощью справки MSDN методы и свойства классов **Graphics**, **Color**, **Pen** и **SolidBrush** либо методы и свойства классов JavaScript. Создайте собственное приложение для **анимации** в соответствии с индивидуальным заданием.

Варианты заданий

1. Создайте программу движения окружности по **циклоиде**.
2. Разработайте программу движения окружности по **гипоциклоиде** при $k = 3$, $k = 4$, $k = 6$, $k = 2,1$, $k = 5,5$.
3. Создайте программу движения окружности по **эпициклоиде** при различных значениях k .
4. Разработайте программу, отображающую процесс построения **гипотрохида**.
5. Создайте программу, моделирующую построение кривых с помощью **спирографа**. R , r , d задаются произвольно.
6. Создайте программу, отображающую движение окружности по **кривым Безье**.
7. Создайте приложение, отображающее полет ядра из пушки, для различных масс ядер.
8. Разработайте программу движения окружности по **трактрисе** (кривой погони).
9. Создайте программу, показывающую движение окружности по **трисектрисе Каталана** (Кубика Чирнгауза).
10. Разработайте программу, отображающую процесс построения **фигур Лиссажу**, с произвольными задаваемыми параметрами.
11. Разработайте приложение, отображающее процесс построения **связанных звезд**, с произвольным числом вершин.

12. Создайте программу, отображающую движения **маятника** с затуханием.

13. Создайте программу, анимирующую процесс построения различных **спиралей** (параболических, логарифмических, архимедовых спираль Корню, клотоида).

14. Разработайте программу, отображающую процесс построения **Лемнискаты Бернулли**.

15. Создайте программу движения объекта по **кривой Персея** при различных значениях a , b и c .

16. Разработайте программу движения точки вдоль **кривой Безье** четвертого порядка. Узловые точки задаются произвольно пользователем до построения кривой.

17. Разработайте программу **анимации падения снежинки**, которые падают по разным траекториям и с разными скоростями.

18. Разработайте программу **анимации летающего бумеранга**.

19. Создайте программу, показывающую **падение нескольких звезд** одновременно.

20. Создайте приложение, отображающее **хаотичное движение звезды** в окне.

Контрольные вопросы

Контрольные вопросы по первой образовательной траектории

1. Для чего необходим таймер?
2. Нужно ли удалять старый кадр анимации при использовании обработчика события **Paint**?
3. Что нужно сделать в обработчике события **Tick** таймера, чтобы реализовать движение по траектории?
4. Как надо организовать обработку события **Paint** для создания анимации?

Контрольные вопросы по второй образовательной траектории

1. Какие существуют способы создания анимации в JavaScript?
2. Необходимо ли очищать холст при создании анимации?
3. Что задает атрибут **GlobalCompositeOperation**?
4. Для чего нужен метод **requestAnimationFrame**?

Лабораторная работа 8 РАБОТА С РАСТРОВЫМИ ИЗОБРАЖЕНИЯМИ

Цель: в соответствии с первой образовательной траекторией изучить возможности С# по обработке растровых изображений либо в соответствии со второй образовательной траекторией изучить возможности обработки растровых изображений в JavaScript; написать и отладить программу для обработки растровых изображений.

Общие сведения

Работа с растровыми изображениями в Visual Studio

Отображение графических файлов. Обычно для отображения точечных рисунков, рисунков из метафайлов, значков, рисунков из файлов в формате BMP, JPEG, GIF или PNG используется объект **PictureBox**, то есть элемент управления **PictureBox** действует как контейнер для картинок. Можно выбрать изображение для вывода, присвоив значение свойству **Image**. Свойство **Image** может быть установлено в окне **Свойства** или в коде программы. При этом указывается на рисунок, который следует отображать.

Элемент управления **PictureBox** содержит и другие полезные свойства, в том числе **AutoSize**, определяющее, будет ли изображение растянуто в элементе **PictureBox**, и **SizeMode**, которое может использоваться для растягивания, центрирования или увеличения изображения в элементе управления **PictureBox**.

Перед добавлением рисунка к элементу управления **PictureBox** в проект обычно добавляется файл рисунка в качестве *ресурса*³. После этого можно повторно использовать его. Например, может потребоваться отображение одного и того же изображения в нескольких местах.

Необходимо отметить, что поле **Image** само является классом для работы с изображениями, у которого есть свои методы. Например, метод **FromFile** используется для загрузки изображения из файла. Кроме класса **Image** существует класс **Bitmap**, который расширяет возможности класса **Image** за счет дополнительных методов для загрузки, сохранения и обработки растровых изображений. Так, метод **Save** класса **Bitmap** позволяет

³ В приложениях Visual Studio часто содержатся данные, не являющиеся исходным кодом. Такие данные называются ресурсами проекта и могут включать двоичные данные, текстовые файлы, аудио- и видеофайлы, таблицы строк, значки, изображения, XML-файлы или любой другой тип данных, необходимых для приложения. Данные ресурсов проекта хранятся в формате XML в файле с расширением RESX (имя по умолчанию – Resources.resx), который можно открыть в Обозревателе решений.

сохранять изображения в разных форматах, а методы **GetPixel** и **SetPixel** позволяют получить доступ к отдельным пикселям рисунка.

Компоненты `OpenFileDialog` и `SaveFileDialog`. Компонент **OpenFileDialog** является стандартным диалоговым окном. Он аналогичен диалоговому окну «Открыть файл» операционной системы Windows.

Компонент **OpenFileDialog** позволяет пользователям просматривать папки личного компьютера или любого компьютера в сети, а также выбирать файлы, которые требуется открыть. Для вызова диалогового окна для выбора файла можно применить метод **ShowDialog()**, который возвращает **true** при корректном выборе.

Диалоговое окно возвращает путь и имя файла, который был выбран пользователем в специальном свойстве **FileName**.

Простой графический редактор. Создайте приложение, реализующее простой графический редактор. Функциями этого редактора должны быть: открытие рисунка, рисование поверх него простой кистью, сохранение рисунка в другой файл. Для этого создайте форму и разместите на ней элементы управления **button** и **picturebox** (рис. 8.1).

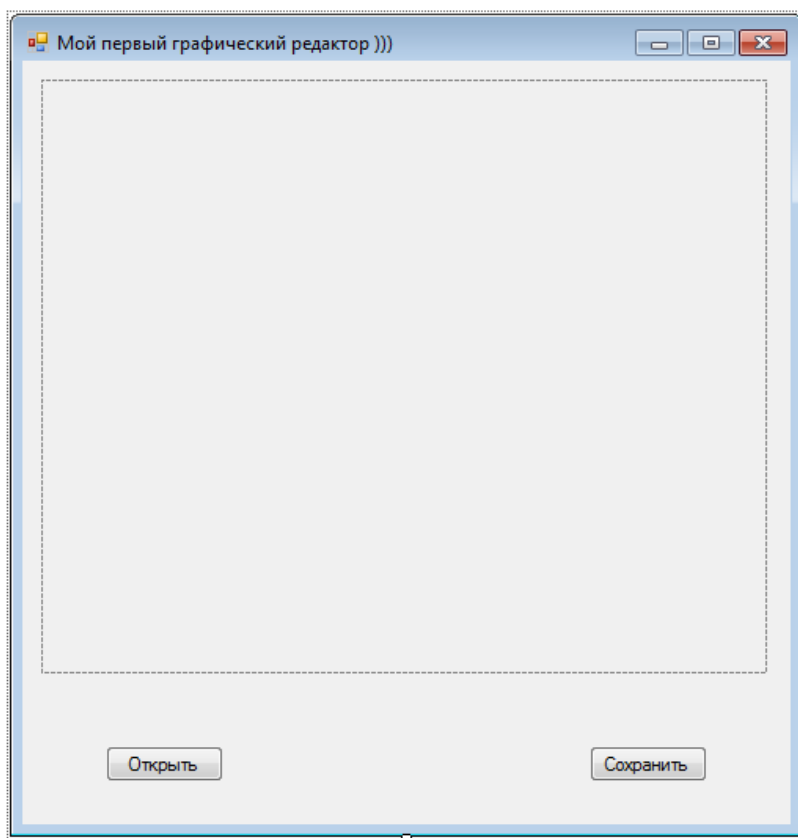


Рис. 8.1. Форма для графического редактора

Дополнительно на форме разместите компоненты диалоговых окон `OpenFileDialog` и `SaveFileDialog` с панели элементов. Эти элементы будут

порождены динамически в ходе выполнения программы с помощью конструктора. Например:

```
OpenFileDialog dialog = new OpenFileDialog();
```

Далее они будут вызываться с помощью метода **ShowDialog()**.

Для кнопок «Открыть» и «Сохранить» создайте свои обработчики события. Также добавьте обработчик события **Load** для формы. Для элемента управления **picturebox1** создайте обработчики события **MouseDown**, **MouseMove**. Код приложения будет выглядеть следующим образом:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        //Объявляем переменные, доступные в каждом обработчике события
        private Point PreviousPoint, point; //Точка до перемещения курсора
                                           // мыши и текущая точка

        private Bitmap bmp;
        private Pen blackPen;
        private Graphics g;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            blackPen = new Pen(Color.Black, 4); //подготавливаем перо
        }

        private void button1_Click(object sender, EventArgs e)
        {
            //открытие файла
            OpenFileDialog dialog = new OpenFileDialog();
            //задаем расширения файлов
            dialog.Filter = "Image files (*.BMP, *.JPG, *.GIF, *.TIF,
                           *.PNG, *.ICO, *.EMF, *.WMF)|*.bmp;*.jpg;*.gif;
                           *.tif;*.png;*.ico;*.emf;*.wmf";
            //Вызываем диалоговое окно
            if (dialog.ShowDialog() == DialogResult.OK)

```

```

    {
        //Загружаем в image изображение из выбранного файла
        Image image = Image.FromFile(dialog.FileName);
        int width = image.Width;
        int height = image.Height;
        pictureBox1.Width = width;
        pictureBox1.Height = height;

        bmp = new Bitmap(image, width, height); //создаем и загружаем из
                                                //image изображение в формате bmp

        pictureBox1.Image = bmp; //записываем изображение в формате bmp
                                //в pictureBox1
        //подготавливаем объект Graphics для рисования в pictureBox1
        g = Graphics.FromImage(pictureBox1.Image);
    }
}

private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{ // обработчик события нажатия кнопки на мыши
  // записываем в предыдущую точку
  //(PreviousPoint) текущие координаты
  PreviousPoint.X = e.X;
  PreviousPoint.Y = e.Y;
}

private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{ //Обработчик события перемещения мыши по pictureBox1
  //Проверяем нажатие левой кнопки
  if (e.Button == MouseButton.Left)
  { //запоминаем в point текущее положение курсора мыши
    point.X = e.X;
    point.Y = e.Y;
    //соединяем линией предыдущую точку с текущей
    g.DrawLine(blackPen, PreviousPoint, point);

    //текущее положение курсора мыши сохраняем в PreviousPoint
    PreviousPoint.X = point.X;
    PreviousPoint.Y = point.Y;
    //Принудительно вызываем перерисовку
    pictureBox1.Invalidate();
  }
}

private void button2_Click(object sender, EventArgs e)
{ //сохранение файла
  SaveFileDialog savedialog = new SaveFileDialog();
  //задаем свойства для savedialog
  savedialog.Title = "Сохранить картинку как ...";
  savedialog.OverwritePrompt = true;
  savedialog.CheckPathExists = true;
}

```

```

savedialog.Filter =
    "Bitmap File(*.bmp)|*.bmp|" +
    "GIF File(*.gif)|*.gif|" +
    "JPEG File(*.jpg)|*.jpg|" +
    "TIF File(*.tif)|*.tif|" +
    "PNG File(*.png)|*.png";
savedialog.ShowHelp = true;
if (savedialog.ShowDialog() == DialogResult.OK)
{
    // в fileName записываем полный путь к файлу
    string fileName = savedialog.FileName;
    // Убираем из имени три последних символа (расширение)
    string strFileExtn = fileName.Remove(0, fileName.Length - 3);
    // Сохраняем файл в нужном формате и с нужным расширением
    switch (strFileExtn)
    {
        case "bmp":
            bmp.Save(fileName, System.Drawing.Imaging.ImageFormat.Bmp);
            break;
        case "jpg":
            bmp.Save(fileName, System.Drawing.Imaging.ImageFormat.Jpeg);
            break;
        case "gif":
            bmp.Save(fileName, System.Drawing.Imaging.ImageFormat.Gif);
            break;
        case "tif":
            bmp.Save(fileName, System.Drawing.Imaging.ImageFormat.Tiff);
            break;
        case "png":
            bmp.Save(fileName, System.Drawing.Imaging.ImageFormat.Png);
            break;
        default:
            break;
    }
}
}}}

```

Далее добавим в проект кнопку для перевода изображения в градации серого цвета:

```

private void button3_Click(object sender, EventArgs e)
{
    //циклы для перебора всех пикселей на изображении
    for (int i = 0; i < bmp.Width; i++)
        for (int j = 0; j < bmp.Height; j++)
        {
            int R = bmp.GetPixel(i, j).R; //извлекаем долю красного цвета
            int G = bmp.GetPixel(i, j).G; //извлекаем долю зеленого цвета
            int B = bmp.GetPixel(i, j).B; //извлекаем долю синего цвета
            int Gray = (R + G + B)/3; // высчитываем среднее
        }
}

```

```

    Color p = Color.FromArgb(255, Gray, Gray, Gray);
    //255-показывает степень прозрачности
    //остальные значения одинаковы для трех каналов R,G,B
    bmp.SetPixel(i, j, p); //записываем полученный цвет в точку
  }
  Refresh(); //вызываем функцию перерисовки окна
}

```

Данный код демонстрирует возможность обращения к отдельным пикселям. Цвет каждого пикселя хранится в модели RGB и состоит из трех составляющих: красного, зеленого и синего цвета, называемых каналами. Значение каждого канала может варьироваться в диапазоне от 0 до 255.

Редактирование растровых изображений в JavaScript

До сих пор холст (**canvas**) рассматривался как инструмент для создания на нем изображений из различных графических примитивов. Необходимо отметить, что холст дает возможность обращаться попиксельно к нарисованным изображениям. Для этого у элемента canvas существует объект **ImageData**.

Объект **ImageData** содержит следующие атрибуты только для чтения:

- **width** – ширина изображения в пикселях;
- **height** – высота изображения в пикселях;
- **data** – одномерный массив, содержащий данные в порядке RGBA, с целыми значениями от 0 до 255. Можно получить доступ к размеру массива пикселей в байтах, прочитав атрибут массива **length**.

Для обращения к атрибуту data, то есть к массиву пикселей, существует метод **getImageData(left, top, width, height)**, параметры которого определяют область считывания пикселей в координатах холста. Этот метод возвращает объект **ImageData**. Для записи пиксельных данных в контекст холста, то есть для отрисовки используется метод **putImageData(myImageData, dx, dy)**, где параметры **dx, dy** указывают смещение внутри контекста холста. Чтобы нарисовать все изображение, представленное **myImageData**, в верхнем левом углу контекста необходимо сделать следующее:

```
ctx.putImageData(myImageData, 0, 0);
```

Рассмотрим приложение для попиксельной обработки изображений. Для этого подготовим HTML-страницу с четырьмя кнопками. Первая – для загрузки изображения из файла, вторая – для копирования загруженного изображения на холст, третья – для перевода изображения на холсте

в негатив и четвертая – для перевода изображения на холсте в градации серого цвета:

```
<html>
<body>
  <input id="uploadImage" type="file" onchange="Open_Image();" />
  <button onclick="CopyToCanvas();"> копируем на канвас </button>
  <button onclick="convert();"> негатив </button>
  <button onclick="GrayScale();"> градации серого </button>

  <!-- подключаем файл my.js -->
  <script src="my.js"></script>
  <p>
    <img src="" id="my_image"/>
    <canvas id="my_canvas"></canvas>
  </p>
</body>
</html>
```

Страница при работе с изображениями будет выглядеть, как показано на рис. 8.2.

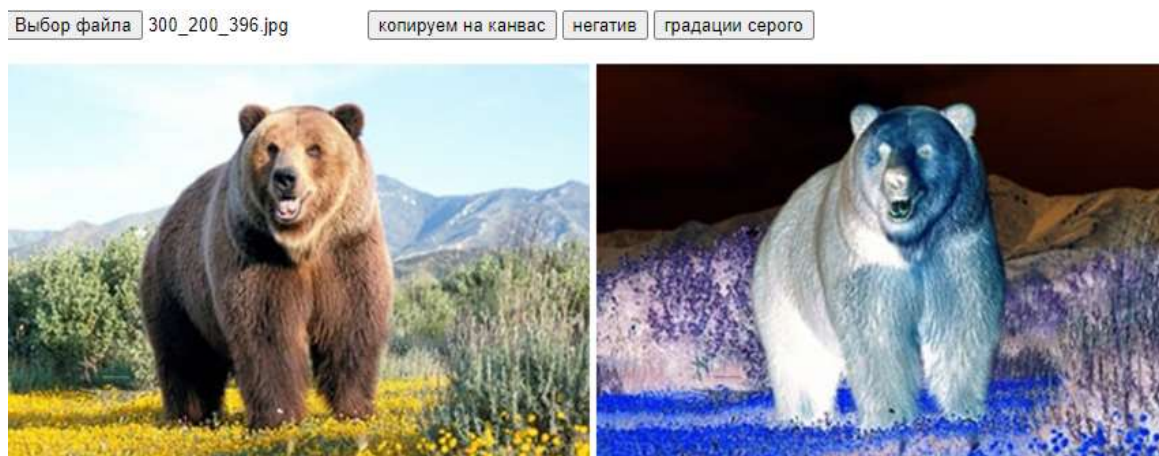


Рис. 8.2. Вид приложения для редактирования изображений

Код, реализующий данные функции, будет выглядеть следующим образом:

```
/** @type {HTMLCanvasElement} */
let canvas = null;
/** @type {CanvasRenderingContext2D} */
let ctx = null;

// функция для открытия изображения с помощью FileReader
function Open_Image()
```

```

{
  let Pic_Reader = new FileReader();
  Pic_Reader.readAsDataURL(document.getElementById('uploadImage').
                           files[0]);
  Pic_Reader.onload = function (PREvent)
    { document.getElementById('my_image').src =
      PREvent.target.result; }
}

function CopyToCanvas()
{
  // получаем холст и изображение
  canvas = document.getElementById('my_canvas');
  let img = document.getElementById('my_image');

  // устанавливаем размер холста как размер изображения
  canvas.width = img.naturalWidth;
  canvas.height = img.naturalHeight;

  // получаем контекст холста
  ctx = canvas.getContext('2d');
  // рисуем изображение на холсте
  ctx.drawImage(img, 0, 0);
}

// функция для получения негатива изображения
function convert()
{
  // получаем холст, контекст и изображение
  canvas = document.getElementById('my_canvas');
  ctx = canvas.getContext('2d');
  let img = document.getElementById('my_image');

  // Загружаем ImageData
  let myImagedata = ctx.getImageData(0, 0, img.naturalWidth,
                                     img.naturalHeight);

  // Копируем массив data из ImageData,
  // чтобы можно было вносить изменения
  let pix = myImagedata.data;
  // Цикл, перебирающий все пиксели на холсте и инвертирующий их цвета
  for (let i = 0, n = pix.length; i < n; i += 4)
  {
    pix[i ] = 255 - pix[i ]; // red
    pix[i+1] = 255 - pix[i+1]; // green
    pix[i+2] = 255 - pix[i+2]; // blue
    // i+3 хранит информацию об альфа-канале
  }

  // Запись и отрисовка ImageData на холсте
  ctx.putImageData(myImagedata, 0, 0);
}

```

```

// функция для перевода изображения в градации серого
function GrayScale()
{
    canvas = document.getElementById('my_canvas');
    ctx = canvas.getContext('2d');
    let img = document.getElementById('my_image');

    let R_canal;
    let G_canal;
    let B_canal;

    // Загружаем ImageData
    let myImagedata = ctx.getImageData(0, 0, img.naturalWidth,
                                        img.naturalHeight);

    // Копируем массив data из ImageData,
    // чтобы можно было вносить изменения
    let pix = myImagedata.data;

    // Цикл, перебирающий все пиксели на холсте и
    // переводящий в серый цвет
    for (let i = 0, n = pix.length; i < n; i += 4)
    {
        R_canal = pix[i ]; // red
        G_canal = pix[i+1]; // green
        B_canal = pix[i+2]; // blue
        // i+3 хранит информацию об альфа-канале
        pix[i ] = (R_canal + G_canal + B_canal) / 3;
        pix[i+1] = (R_canal + G_canal + B_canal) / 3;
        pix[i+2] = (R_canal + G_canal + B_canal) / 3;
    }

    // Запись и отрисовка ImageData на холсте
    ctx.putImageData(myImagedata, 0, 0);
}

```

В этом приложении картинка сначала загружается в изображение (тэг **img**), далее копируется на холст, где с ним и производятся манипуляции.

Как видим, основой обработки изображения является цикл, перебирающий попиксельно массив data объекта ImageData с холста. В цикле берется четыре байта, которые представляют каналы RGB-модели, и один байт альфа-канала.

Для чтения изображения из файла используется объект FileReader. Дополнительные сведения о загрузке из файла и применении FileReader можно найти по ссылке <https://learn.javascript.ru/file>.

Дополнительные сведения о попиксельной обработке изображений на холсте можно найти в документации по ссылке https://developer.mozilla.org/ru/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas.

Задание по лабораторной работе

Изучите представленный пример. Добавьте в приложение для обработки растровых изображений новую функцию в соответствии с индивидуальным заданием.

Варианты заданий

1. Расширьте приложение путем добавления возможности регулировки **яркости** и **контрастности** как в целом сразу по трём каналам, так и по каждому каналу отдельно.
2. Расширьте приложение путем добавления различных **фильтров размытия** с различным размером ядра.
3. Расширьте приложение путем добавления различных **фильтров повышения контрастности** с различным размером ядра.
4. Расширьте приложение путем добавления различных фильтров **нахождения границ**.
5. Расширьте приложение путем добавления **медианного фильтра**.
6. Создайте функцию, переводящую изображение в черно-белый формат (**реализуйте пороговую фильтрацию**). Пороговое значение задайте с помощью элемента управления.
7. Разработайте функцию **построения гистограммы** по трем каналам (R, G, B) вместе и по каждому каналу в отдельности.
8. Реализуйте функцию **поворота** выделенного участка изображения.
9. Реализуйте функцию вертикального и горизонтального **отражения** выделенного участка изображения.
10. Разработайте функцию, оставляющую на изображении только один из каналов (R, G, B). Канал выбирается пользователем.
11. Создайте приложение, которое будет заменять все цвета, близкие к зеленому, на оттенки синего.
12. Создайте функцию, которая модифицирует изображение следующим образом. В центре изображения область пикселей в виде окружности перекрашивается в градации серого цвета. Все точки вне окружности остаются неизменными. Радиус окружности задается пользователем.
13. Создайте функцию, которая изменяет изображение следующим образом. На изображение появляется область пикселей в виде треугольника. Для всех точек вне треугольника оставьте только канал B. Все точки внутри треугольника переводятся в градации серого цвета.
14. Создайте функцию, которая формирует на изображении область пикселей в виде ромба. Все точки вне ромба переводятся в градации серого цвета. Для всех точек внутри ромба оставьте только канал G.
15. Разработайте функцию, которая каждую четную строку изображения переводит в градации серого цвета, а каждый четный столбец

будет содержать только канал В. На пересечении этих строк и столбцов поставьте белые пиксели.

16. Разработайте функцию, которая переводит каждый нечетный столбец пикселей (вертикальные линии) в градации серого цвета и каждую нечетную строку преобразует в зеленую линию.

17. Создайте функцию, разбивающую изображение на четыре равные части. В каждой оставьте значение только одного канала R, G и B, а в четвертой выведите градации серого цвета.

18. Разработайте функцию, заменяющую все точки синего цвета на точки красного цвета.

19. Создайте функцию, **инвертирующую** изображение в градациях серого цвета **в негатив**.

20. Создайте функцию, переводящую изображение в черно-белый формат в соответствии с пороговым значением, которое ввел пользователь. Для анализа используйте только один из каналов (R, G, B).

Контрольные вопросы

Контрольные вопросы по первой образовательной траектории

1. Какой объект можно использовать как контейнер для растровых изображений?
2. Какие методы класса **Bitmap** позволяют получить доступ к отдельным пикселям?
3. Какие компоненты используются для доступа к окнам открытия и сохранения файла?
4. Что передается в четырех параметрах метода **Color.FromArgb**?

Контрольные вопросы по второй образовательной траектории

1. Для чего необходим объект **ImageData**?
2. Какой объект используется для открытия изображения в JavaScript?
3. Что делает следующий код на JavaScript?

```
for (let i = 0, n = pix.length; i < n; i += 4)
{
    pix[i] = 255 - pix[i]; // red
    pix[i+1] = 255 - pix[i+1]; // green
    pix[i+2] = 255 - pix[i+2]; // blue
    // i+3 хранит информацию о альфа-канале
}
```

4. Как необходимо подготовить HTML-код для работы с растровыми изображениями?

Лабораторная работа 9 ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ

Цель: изучить, как производятся двухмерные преобразования с помощью однородных координат и матрицы преобразования 3×3 ; написать и отладить программу для 2D-преобразований.

Общие сведения

Простейшие преобразования на плоскости

Рассмотрим преобразования на плоскости. Для начала заметим, что точки на плоскости задаются с помощью двух ее координат. Таким образом, геометрически каждая точка задается значениями координат вектора относительно выбранной системы координат. Координаты точек можно рассматривать как элементы матрицы $[x, y]$, то есть в виде вектор-строки или вектор-столбца. Положением этих точек управляют путем преобразования матрицы.

Точки на плоскости x, y можно перенести в новые позиции путем добавления к координатам этих точек констант переноса:

$$[x^* \ y^*] = [x \ y] + [a \ b] = [x + a \ y + b].$$

Рассмотрим результаты матричного умножения матрицы $[x, y]$, определяющей точку P и матрицы преобразований 2×2 общего вида:

$$[x \ y] \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = [(ax + cy) \ (bx + dy)] = [x^* \ y^*].$$

Проведем анализ полученных результатов, рассматривая x^* и y^* как преобразованные координаты. Для этого исследуем несколько частных случаев.

Рассмотрим случай, когда $a = d = 1$ и $c = b = 0$. Матрица преобразований приводит к матрице, идентичной исходной:

$$[x \ y] \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [(1x + 0y) \ (0x + 1y)] = [x \ y] = [x^* \ y^*].$$

При этом изменений координат точки P не происходит.

Если теперь $d = 1, b = c = 0, a = \text{const}$, то

$$[x \ y] \cdot \begin{bmatrix} a & 0 \\ 0 & 1 \end{bmatrix} = [(ax + 0y) \ (0x + 1y)] = [ax \ y] = [x^* \ y^*].$$

Как видно, это приводит к изменению масштаба в направлении x , так как $x^* = ax$. Следовательно, данное матричное преобразование эквивалентно перемещению исходной точки в направлении x .

Теперь положим $b = c = 0$, то есть

$$\begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix} = \begin{bmatrix} (ax + 0y) & (0x + dy) \end{bmatrix} = \begin{bmatrix} ax & dy \end{bmatrix} = \begin{bmatrix} x^* & y^* \end{bmatrix}.$$

В результате получаем изменение масштабов в направлениях x и y . Если $a \neq d$, то перемещения вдоль осей неодинаковы. Если $a = d > 1$, то имеет место увеличение масштаба координат точки P . Если $0 < a = d < 1$, то будет иметь место уменьшение масштаба координат точки P .

Если a или (и) d отрицательны, то происходит отображение координат точек. Рассмотрим это, установив $b = c = 0$, $d = 1$ и $a = -1$, тогда

$$\begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} (-1x + 0y) & (0x + 1y) \end{bmatrix} = \begin{bmatrix} -x & y \end{bmatrix} = \begin{bmatrix} x^* & y^* \end{bmatrix}.$$

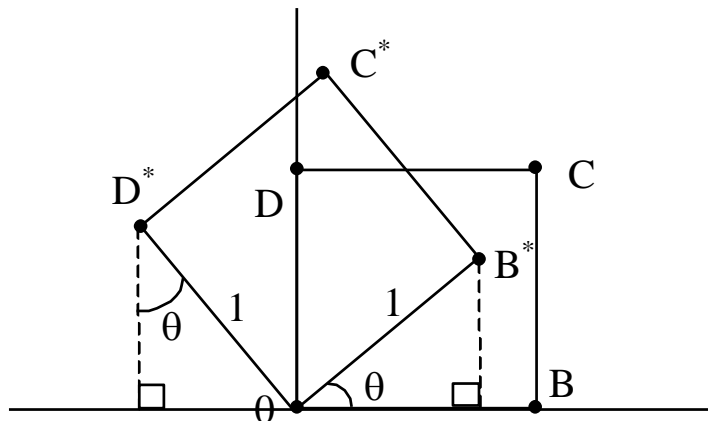
Произошло отображение точки относительно оси y . В случае $b = c = 0$, $a = 1$, $d = -1$ отображение происходит относительно оси x . Если $b = c = 0$, $a = d < 0$, то отображение будет происходить относительно начала координат.

Заметим, что отображение и изменение масштаба вызывают только диагональные элементы матрицы преобразования.

Преобразование общего вида, примененное к началу координат, не приведет к изменению координат точки $(0, 0)$. Следовательно, начало координат инвариантно при общем преобразовании. Это ограничение преодолевается за счет использования однородных координат.

Преобразование поворота и отражения

Общую матрицу 2×2 , которая осуществляет вращение фигуры относительно начала координат, можно получить из рассмотрения вращения единичного квадрата вокруг начала координат.



Как следует из рисунка, точка B с координатами $(1, 0)$ преобразуется в точку B^* , для которой $x^* = (1) \cos\theta$ и $y = (1) \sin\theta$, а точка D , имеющая координаты $(0, 1)$ переходит в точку D^* с координатами $x^* = (-1) \sin\theta$ и $y^* = (1) \cos\theta$.

Матрица преобразования общего вида записывается так:

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}.$$

Для частных случаев: поворот на 90° можно осуществить с помощью матрицы преобразования

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

Если использовать матрицу координат вершин, то получим, например:

$$\begin{bmatrix} 3 & -1 \\ 4 & 1 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -1 & 4 \\ -1 & 2 \end{bmatrix}.$$

Поворот на 180° получается с помощью матрицы $\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$.

В то время как чистое двумерное вращение в плоскости xu осуществляется вокруг оси, перпендикулярной к этой плоскости, отображение определяется поворотом на 180° вокруг оси, лежащей в плоскости xu .

Такое вращение вокруг линии $y = x$ происходит при использовании матрицы $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.

Преобразованные новые выражения определяются соотношением

$$\begin{bmatrix} 8 & -1 \\ 7 & 3 \\ 6 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 8 \\ 3 & 7 \\ 2 & 6 \end{bmatrix}.$$

Вращение вокруг $y = 0$ получается при использовании матрицы

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Однородные координаты

Преобразования переноса, масштабирования и поворота записываются в матричной форме в виде

$$P^* = P + T;$$

$$P^* = P \cdot S;$$

$$P^* = P \cdot R.$$

Очевидно, что перенос, в отличие от масштабирования и поворота, реализуется с помощью сложения. Это обусловлено тем, что вводить константы переноса внутрь структуры общей матрицы размера 2×2 не представляется возможным. Желательным является представление преобразований в единой форме – с помощью умножения матриц. Эту проблему можно решить за счет введения третьей компоненты в векторы точек $[x \ y]$ и $[x^* \ y^*]$, то есть представляя их в виде $[x \ y \ 1]$ и $[x^* \ y^* \ 1]$. Матрица преобразования после этого становится матрицей размера 3×3 , например:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ m & n & 1 \end{bmatrix}.$$

Используя эту матрицу, получаем преобразованный вектор $[x^* \ y^* \ 1]$. Добавление третьего элемента к вектору положения и третьего столбца к матрице преобразования позволяет выполнить смещение вектора положения. Третий элемент здесь можно рассматривать как дополнительную координату вектора положения. Итак, вектор положения $[x \ y \ 1]$ при воздействии на него матрицы 3×3 становится вектором положения в общем случае вида $[X \ Y \ H]$. Представленное преобразование было выполнено так, что $[X \ Y \ H] = [x^* \ y^* \ 1]$.

Преобразование, имеющее место в трехмерном пространстве, в данном случае ограничено плоскостью, поскольку $H = 1$. Если, однако, тре-

тий столбец $\begin{bmatrix} p \\ q \\ s \end{bmatrix}$ матрицы преобразования T размера 3×3 отличен от 0, то в

результате матричного преобразования получим $[x \ y \ 1] \cdot T = [X \ Y \ H]$, где $H \neq 1$.

Плоскость, в которой теперь лежит преобразованный вектор положения, находится в трехмерном пространстве.

Преобразованные обычные координаты получаются за счет **нормализации** однородных координат, то есть

$$x^* = \frac{X}{H} \quad \text{и} \quad y^* = \frac{Y}{H}.$$

Геометрически все преобразования x и y происходят в плоскости $H = 1$ после нормализации преобразованных однородных координат.

Преимущество введения однородных координат проявляется при использовании матрицы преобразований общего вида порядка 3×3 :

$$\begin{bmatrix} a & b & p \\ c & d & q \\ m & n & s \end{bmatrix},$$

с помощью которой можно выполнять и другие преобразования, такие как смещение, операции изменения масштаба и сдвига, обусловленные матричными элементами a, b, c и d . Указанные операции рассмотрены ранее.

Основная матрица преобразования размера 3×3 для двумерных однородных координат может быть подразделена на четыре части:

$$\left[\begin{array}{cc|c} a & b & p \\ c & d & q \\ \hline m & n & s \end{array} \right].$$

Как видим, a, b, c и d осуществляют изменение масштаба, сдвиг и вращение, m и n выполняют смещение, а p и q – получение проекций. Оставшаяся часть матрицы, элемент s , производит полное изменение масштаба. Чтобы показать это, рассмотрим преобразование

$$[X \ Y \ H] = [x \ y \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & s \end{bmatrix} = [x \ y \ s],$$

где $X = x$; $Y = y$; $H = s$. Это дает $x^* = x/s$ и $y^* = y/s$.

В результате преобразования $[x \ y \ 1] \rightarrow [x/s \ y/s \ 1]$ имеет место однородное изменение масштаба вектора положения. При $s < 1$ происходит увеличение масштаба, а при $s > 1$ – уменьшение.

Комбинированные преобразования

Рассмотрим комбинированные преобразования на примере поворота вокруг произвольной точки.

Ранее было рассмотрено вращение изображения около начала координат. Однородные координаты обеспечивают поворот изображения вокруг точек, отличных от начала координат. В общем случае вращение около произвольной точки может быть выполнено путем переноса центра вращения в начало координат, поворотом относительно начала координат, а затем переносом точки вращения в исходное положение. Таким образом, поворот вектора положения $[x \ y \ 1]$ около точки (m, n) на произвольный угол может быть выполнен с помощью преобразования

$$[x \ y \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -m & -n & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ m & n & 1 \end{bmatrix} = [X \ Y \ H].$$

Выполнив две операции умножения матриц, запишем:

$$[X \ Y \ H] = [x \ y \ 1] \cdot \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ -m(\cos \theta - 1) + n(\sin \theta) & -m(\sin \theta) - n(\cos \theta - 1) & 1 \end{bmatrix}.$$

Программная реализация в Visual Studio

На первом этапе спроектируйте оконный интерфейс. Разместите на форме кнопки для переноса, масштабирования и поворота. При необходимости разместите элементы для ввода значений, например для ввода значений углов поворота. Пример приведен на рис. 9.1.

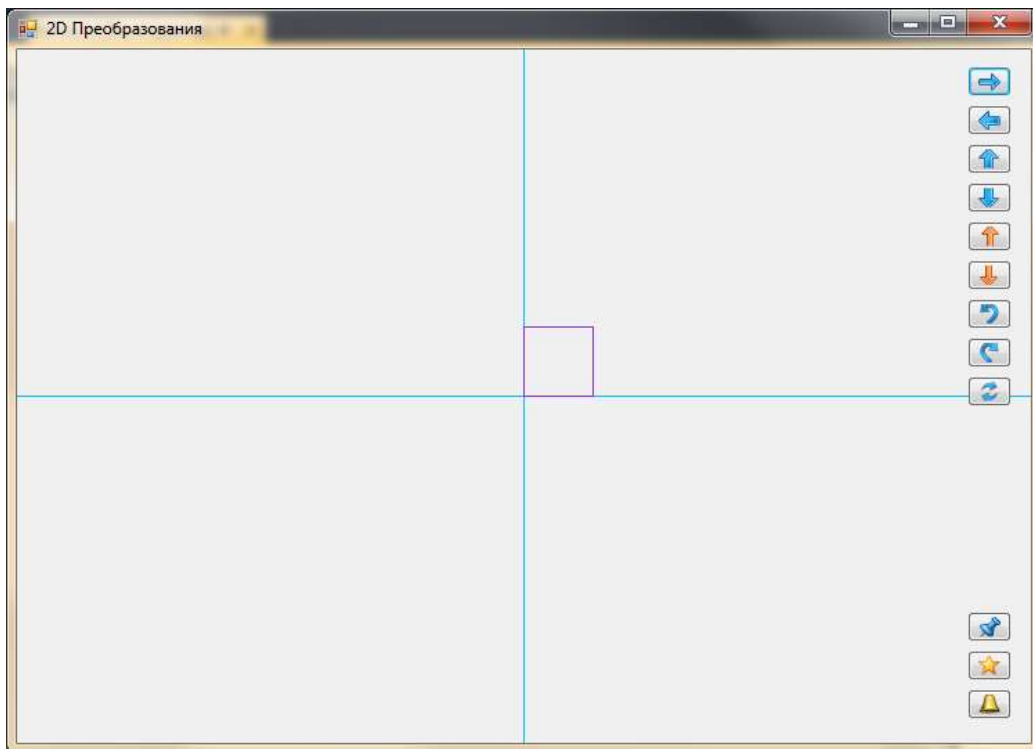


Рис. 9.1. Интерфейс для преобразований на плоскости

Опишите и задайте начальные значения для матрицы, где будут храниться координаты двумерной фигуры. Назовем эту матрицу матрицей тела и опишем как двумерный массив **Sq** типа **double**. Опишите и задайте как единичную матрицу преобразования 3×3 (двухмерный массив **T** типа **double**). Поскольку массивы **Sq** и **T** будут использоваться повсеместно, то целесообразно задать их в качестве полей класса, доступных во всех методах, и не тратить время на передачу в качестве параметров.

При нажатии на кнопку должен запускаться процесс, приведённый на рис. 9.2.



Рис. 9.2. Основной процесс визуализации

Этап задания коэффициентов матрицы преобразования сводится к определению элементов двумерного массива.

Как видим из рис. 9.2, на втором этапе выбрана стратегия сохранения изменений при преобразованиях в матрице тела, а не накапливание изменений в матрице преобразований. Исходя из этого тело метода, реализующего данный этап, может выглядеть следующим образом:

```
b = new double[3]; //массив для хранения промежуточных данных
//Цикл по вершинам фигуры (4 вершины для квадрата)
for (int j=0; j<4; j++)
{
    for (int i = 0; i < 3; i++)
    {
        b[i] = 0;
        for (int k = 0; k < 3; k++)
            b[i] = b[i] + Sq[j, k]*T[k, i];
    }
    for (int k=0; k<3; k++)
        Sq[j, k]= b[k];
}
```

Третий этап нормализации приводит к единице последнюю координату путем деления:

```
for (int j = 0; j < 4; j++)
{
    Sq[j, 0] = Sq[j, 0] / Sq[j, 2];
    Sq[j, 1] = Sq[j, 1] / Sq[j, 2];
    Sq[j, 2] = 1;
}
```

Перерисовка должна находиться в обработчике события **Paint**:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics gfx = e.Graphics;
    Pen skyBluePen = new Pen(Brushes.DeepSkyBlue);
    int cx = ClientSize.Width/2;
    int cy = ClientSize.Height/2;
    gfx.DrawLine(skyBluePen, cx, 0, cx, ClientSize.Height);
    gfx.DrawLine(skyBluePen, 0, cy, ClientSize.Width, cy);

    Pen SqPen = new Pen(Brushes.BlueViolet);

    gfx.DrawLine(SqPen, (int) (cx + Sq[0, 0]), (int) (cy - Sq[0, 1]),
                    (int) (cx + Sq[1, 0]), (int) (cy - Sq[1, 1]));
    gfx.DrawLine(SqPen, (int) (cx + Sq[1, 0]), (int) (cy - Sq[1, 1]),
                    (int) (cx + Sq[2, 0]), (int) (cy - Sq[2, 1]));
    gfx.DrawLine(SqPen, (int) (cx + Sq[2, 0]), (int) (cy - Sq[2, 1]),
                    (int) (cx + Sq[3, 0]), (int) (cy - Sq[3, 1]));
    gfx.DrawLine(SqPen, (int) (cx + Sq[3, 0]), (int) (cy - Sq[3, 1]),
                    (int) (cx + Sq[0, 0]), (int) (cy - Sq[0, 1]));
}
```

Переменные **cx** и **cy** – это координаты центра окна, используются для прорисовки осей и самой фигуры. Приведение к целому типу, а следовательно, округление, производится непосредственно перед прорисовкой и не сохраняется в матрице **Sq**. Поэтому не будет происходить накопления ошибок округления. Запустите и отладьте программу. По результатам подготовьте отчет.

Программная реализация на JavaScript

На первом этапе проектируется пользовательский интерфейс с помощью средств HTML. Для этого можно использовать таблицы, кнопки с изображениями и элементы ввода значений.

```
<html>
<body>
<h1> 2D преобразования</h1>
<!-- подключаем файл my.js -->
<script src="my.js"></script>

<table>
<tr>
    <td>
        <!-- пример описания одной кнопки с обработчиком события onclick -->
        <p>
            <button onclick="move_left();">
            
            </button>
```

```

<!-- пример описания поля для ввода значений -->
<input type="number" value="5" style="width: 5em;" id="input_left">
</p>
<!-- остальные кнопки и поля ввода описываются аналогичным образом -->
</td>

<!-- описание холста с обработчиком события onmousedown -->
<td><canvas width="400" height="400"
           onmousedown="mousedown(event);"></canvas></td>
</tr>
</table>

<script language='JavaScript'> InitCanvas(); </script>
</body>
</html>

```

Вид возможного интерфейса приведен на рис. 9.3.

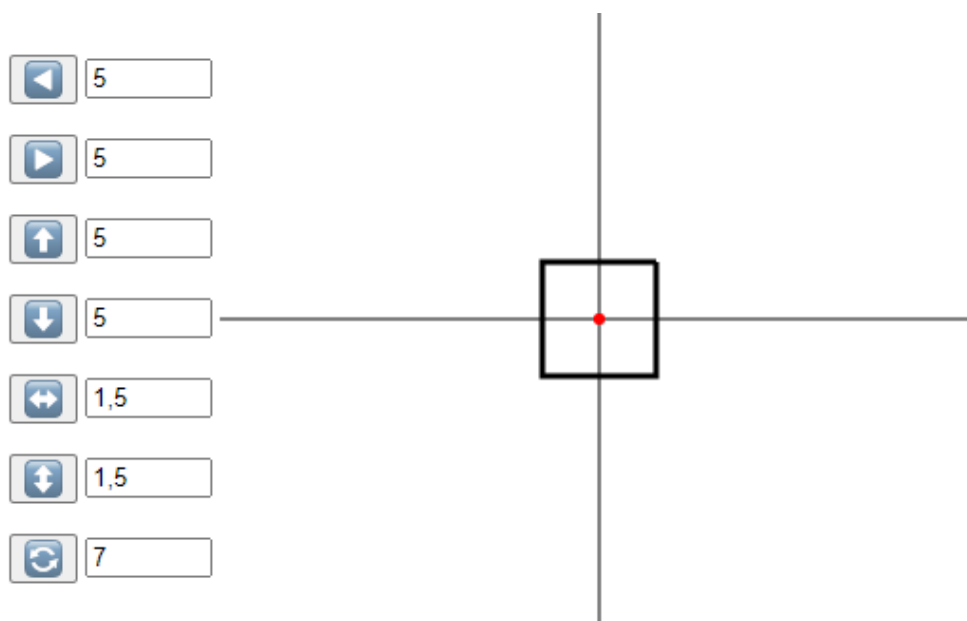


Рис. 9.3. Вид Web-страницы для 2D-преобразований

В подключаемом **js**-файле опишите и задайте начальные значения для матрицы, где будут храниться координаты двухмерной фигуры. Координаты вершин фигуры будут храниться в массиве **fig**. Опишите и задайте единичную матрицу преобразования 3×3 (двухмерный массив **t**).

При нажатии на кнопку должен запускаться процесс, приведённый на рис. 9.2.

Этап задания коэффициентов матрицы преобразования сводится к присвоению элементов двухмерного массива.

Как видим из рис. 9.2, на втором этапе выбрана стратегия сохранения изменений при преобразованиях в матрице тела, а не накапливание изменений в матрице преобразований.

Приведем пример метода для переноса фигуры влево.

```
function move_left()
{
    let x = document.getElementById("input_left");
    t = [
        [1, 0, 0],
        [0, 1, 0],
        [-x.value, 0, 1],
    ]

    fig = MultiplyMatrix(fig, t)

    DrawAll();
}
```

Функция **MultiplyMatrix()** реализует умножение матриц:

```
function MultiplyMatrix(A, B)
{
    let rowsA = A.length, colsA = A[0].length,
        rowsB = B.length, colsB = B[0].length,
        C = [];
    if (colsA !== rowsB) return false;
    for (let i = 0; i < rowsA; i++) C[ i ] = [];
    for (let k = 0; k < colsB; k++)
    { for (let i = 0; i < rowsA; i++)
      { let t = 0;
        for (let j = 0; j < rowsB; j++) t += A[ i ][j]*B[j][k];
        C[ i ][k] = t;
      }
    }
    return C;
}
```

Необходимо помнить про третий этап нормализации. Нормализация приведет к единице последнюю координату путем деления X и Y на величину H .

Функция **DrawAll()** реализует очистку холста, прорисовку осей, точки поворота и самой фигуры. Фрагмент кода для отрисовки фигуры приведен ниже.

```
//Вычисление центра холста
let CenterX = canvas.width / 2;
let CenterY = canvas.height / 2;

//Очистка холста
ctx.clearRect(0,0,800,800);
```

```

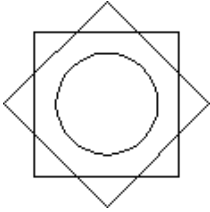
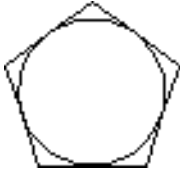
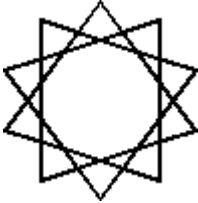

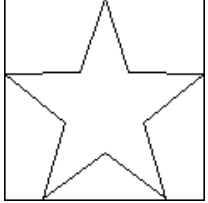
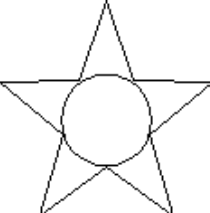
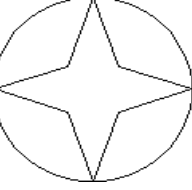

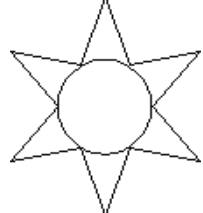

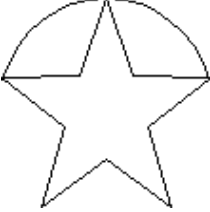
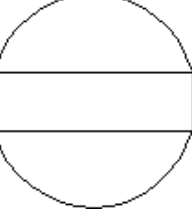
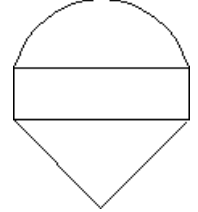
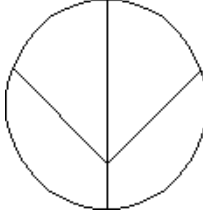
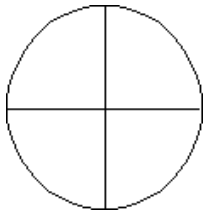
//Отрисовка фигуры
ctx.lineWidth = 3;
ctx.strokeStyle = "black";
ctx.beginPath();
ctx.moveTo(CenterX + fig[0][0], CenterY - fig[0][1]);
ctx.lineTo(CenterX + fig[1][0], CenterY - fig[1][1]);
ctx.lineTo(CenterX + fig[2][0], CenterY - fig[2][1]);
ctx.lineTo(CenterX + fig[3][0], CenterY - fig[3][1]);
ctx.lineTo(CenterX + fig[0][0], CenterY - fig[0][1]);
ctx.stroke();

```

Задание по лабораторной работе

Реализуйте с заданной совокупностью фигур все виды аффинных преобразований: перенос по оси OX и оси OY , отражение относительно координатных осей и прямой $Y = X$, общее масштабирование и масштабирование отдельно по осям, поворот на заданные углы относительно центра координат и относительно произвольной точки, указываемой в ходе выполнения программы. Необходимо предусмотреть восстановление исходной позиции фигур, управление организовать как через интерфейсные элементы (меню, кнопки, строки редактирования и пр.), так и через горячие клавиши.

Варианты заданий



Контрольные вопросы

1. Для чего введены однородные координаты?
2. В чем заключается процесс нормализации координат?
3. Какими способами можно масштабировать объекты?
4. Как производятся комбинированные преобразования?

Лабораторная работа 10 3D-ПРЕОБРАЗОВАНИЯ И ПОЛУЧЕНИЕ ПРОЕКЦИЙ

Цель: изучить, как производятся преобразования в пространстве; освоить способ получения проекций с помощью матрицы 4×4 ; написать и отладить программу для 3D-преобразований.

Общие сведения

Правосторонняя система координат

Рассмотрим трехмерную декартову систему координат, являющуюся **правосторонней** (рис. 10.1). Примем соглашение, в соответствии с которым будем считать положительными такие повороты, при которых (если смотреть с конца полуоси в направлении начала координат) поворот на 90° против часовой стрелки будет переводить одну полуось в другую. На основе этого соглашения строится следующая таблица, которую можно использовать как для правых, так и для левых систем координат:

Если ось вращения	Положительным будет направление поворота
X	от y к z
Y	от z к x
Z	от x к y

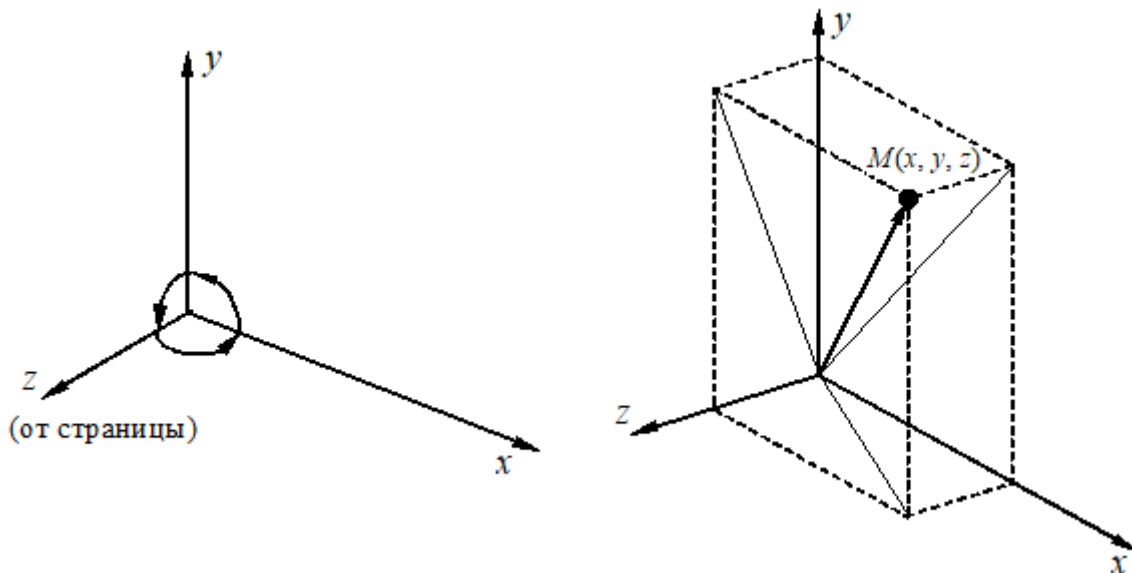


Рис. 10.1. Трехмерная система координат

Преобразования в пространстве

Аналогично тому, как точка на плоскости описывается вектором (x, y) , точка в трехмерном пространстве описывается вектором (x, y, z) .

Как и в случае двухмерного преобразования, для возможности реализаций трехмерных преобразований с помощью матриц перейдем к однородным координатам:

$$[x, y, z, 1] \text{ или } [X, Y, Z, H];$$
$$[x^*, y^*, z^*, 1] = \left[\frac{X}{H}, \frac{Y}{H}, \frac{Z}{H}, 1 \right],$$

где $H \neq 1, H \neq 0$.

Обобщенная матрица преобразования 4×4 для трехмерных однородных координат имеет вид

$$T = \begin{bmatrix} a & b & c & p \\ d & e & f & q \\ h & i & j & r \\ l & m & n & s \end{bmatrix}.$$

Данная матрица

$$\begin{bmatrix} 3 \times 3 & 3 \times 1 \\ 1 \times 3 & 1 \times 1 \end{bmatrix}$$

может быть представлена в виде четырех отдельных частей:

- матрица 3×3 осуществляет линейное⁴ преобразование в виде изменения масштаба, сдвига и вращения;
- матрица 1×3 производит перенос;
- матрица 3×1 осуществляет преобразования в перспективе;
- скалярный элемент 1×1 выполняет общее изменение масштаба.

Рассмотрим воздействие матрицы 4×4 на однородный вектор $[x, y, z, 1]$.

Трехмерный перенос. Трехмерный перенос является простым расширением двумерного:

$$T(Dx, Dy, Dz) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Dx & Dy & Dz & 1 \end{bmatrix},$$

то есть $[x, y, z, 1] \cdot T(Dx, Dy, Dz) = [x + Dx, y + Dy, z + Dz, 1]$.

⁴ **Линейное преобразование** трансформирует исходную линейную комбинацию векторов в некоторую линейную их комбинацию.

Трёхмерное изменение масштаба. Рассмотрим **частичное** изменение масштаба. Оно реализуется следующим образом:

$$S(Sx, Sy, Sz, 1) = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

то есть $[x, y, z, 1] \cdot S(Sx, Sy, Sz) = [Sx \cdot x, Sy \cdot y, Sz \cdot z, 1]$.

Общее изменение масштаба. Общее изменение масштаба производится, если в матрице преобразований элемент S отличен от единицы.

$$[x \ y \ z \ 1]^* \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & S \end{bmatrix} = [x \ y \ z \ S] = [x^* \ y^* \ z^* \ 1] = \left[\frac{X}{S}, \frac{Y}{S}, \frac{Z}{S}, 1 \right].$$

Такой же результат можно получить при равных коэффициентах частичных изменений масштабов. В этом случае матрица преобразования такова:

$$S = \begin{bmatrix} \frac{1}{S} & 0 & 0 & 0 \\ 0 & \frac{1}{S} & 0 & 0 \\ 0 & 0 & \frac{1}{S} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Трёхмерный сдвиг. Недиагональные элементы матрицы 3×3 осуществляют сдвиг в трех измерениях, то есть

$$[x \ y \ z \ 1]^* \begin{bmatrix} 1 & b & c & 0 \\ d & 1 & f & 0 \\ h & i & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x + yd + hz, bx + y + iz, cx + fy + z, 1].$$

Трёхмерное вращение. Двухмерный поворот, рассмотренный ранее, является в то же время трёхмерным поворотом вокруг оси Z . В трёхмерном пространстве поворот вокруг оси Z описывается матрицей

$$R_z(\Theta) = \begin{bmatrix} \cos(\Theta) & \sin(\Theta) & 0 & 0 \\ -\sin(\Theta) & \cos(\Theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матрица поворота вокруг оси X имеет вид

$$R_x(\Theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\Theta) & \sin(\Theta) & 0 \\ 0 & -\sin(\Theta) & \cos(\Theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матрица поворота вокруг оси Y имеет вид

$$R_y(\Theta) = \begin{bmatrix} \cos(\Theta) & 0 & -\sin(\Theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\Theta) & 0 & \cos(\Theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Результатом произвольной последовательности поворотов вокруг осей x , y , z является матрица

$$A = \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ h & i & j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Получение проекций

Каждую из проекций можно описать матрицей 4×4 . Этот способ оказывается удобным, поскольку появляется возможность объединить матрицу проецирования с матрицей преобразования.

Получение одноточечной перспективной проекции. Центральная (перспективная) проекция получается путем перспективного преобразования и проецирования на некоторую двухмерную плоскость «наблюдения». Центр проекции находится в точке с координатами $(0, 0, -k)$ (рис. 10.2), плоскость проецирования $Z = 0$. Перспективная проекция в этом случае обеспечивается преобразованием

$$[X Y Z H] = [x y z 1]^* \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x y 0 (rz+1)].$$

где $r = \frac{1}{k}$ (рис. 10.2).

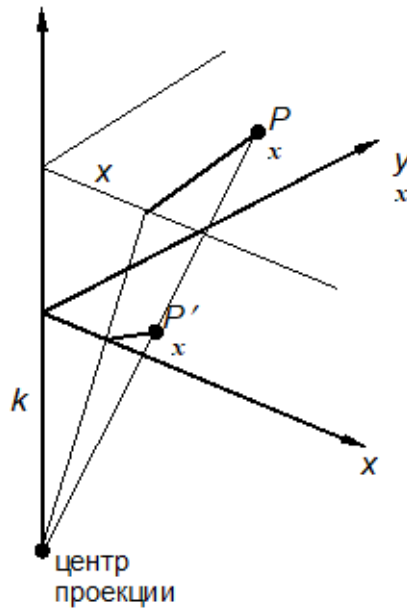


Рис. 10.2. Вычисление одноточечной перспективы

Матрица для такого преобразования называется *матрицей проецирования* или *матрицей проекции*. После умножения на матрицу проекции необходимо провести процедуру нормализации:

$$x^* = \frac{X}{H} = \frac{x}{rz+1};$$

$$y^* = \frac{Y}{H} = \frac{y}{rz+1};$$

$$z^* = \frac{Z}{H} = \frac{0}{rz+1}.$$

Получение косоугольных проекций. Рассмотрим теперь косоугольную проекцию (рис. 10.3). Матрица может быть записана исходя из значений α и l .

Проекцией точки $P(0, 0, 1)$ является точка $P'(l \cos \alpha, l \sin \alpha, 0)$, принадлежащая плоскости xu . Направление проецирования совпадает с отрезком PP' , проходящим через две эти точки. Это направление есть

$P' - P = (l \cos\alpha, l \sin\alpha, -1)$. Направление проецирования составляет угол β с плоскостью xu .

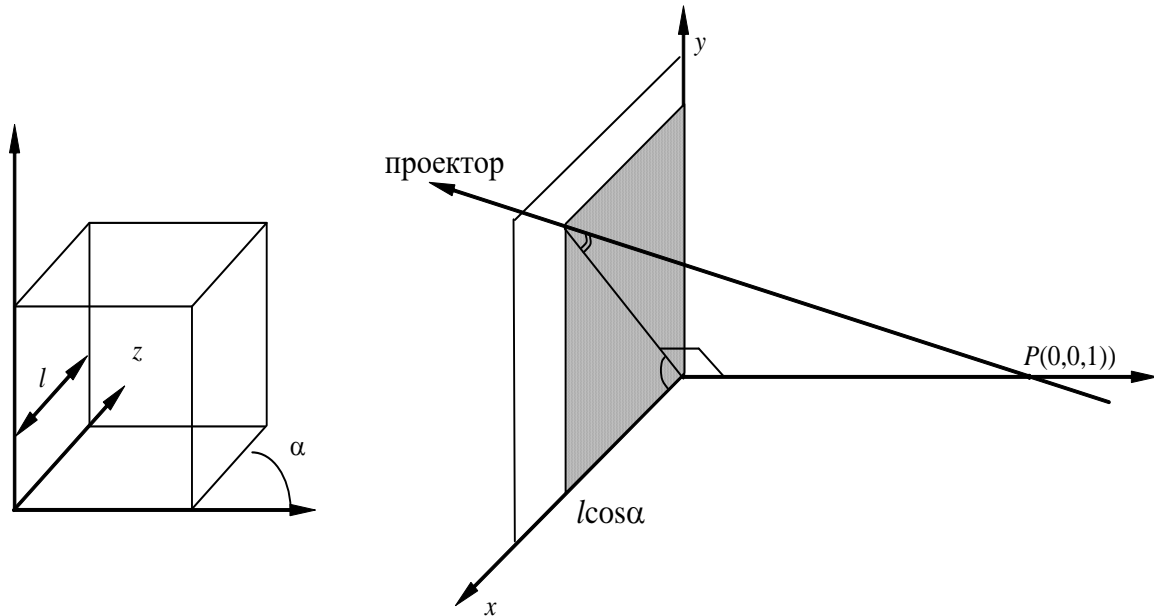


Рис. 10.3. Косоугольные проекции

Теперь рассмотрим проекцию точки x, y, z и определим ее косоугольную проекцию (x_p, y_p) на плоскости xu :

$$x_p = x + z(l \cos\alpha);$$

$$y_p = y + z(l \sin\alpha).$$

Таким образом, матрица 4×4 , которая выполняет эти действия и, следовательно, описывает косоугольную проекцию, имеет вид

$$M_{\text{кос}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ l \cos\alpha & l \sin\alpha & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Применение матрицы $M_{\text{кос}}$ приводит к сдвигу и последующему проецированию объекта: плоскости с постоянной координатой $z = z_1$ переносятся в направлении x на $z_1 l \cos\alpha$ и в направлении y на $z_1 l \sin\alpha$, затем проецируются на плоскость $z = 0$. Сдвиг сохраняет параллельность прямых, а также углы и расстояния в плоскостях, параллельных оси z .

Для проекции Кавалье $l = 1$, поэтому угол $\beta = 45^\circ$. Для проекции Кабине $l = 1/2$, а $\beta = \arctg(2) = 63,4^\circ$. В случае ортографической проекции $l = 0$ и $\beta = 90^\circ$, поэтому матрица ортографического проецирования является частным случаем косоугольной проекции.

Построение вида спереди. Получение вида спереди возможно также с помощью матрицы 4×4 , которая имеет следующий вид

$$M_{\text{орт}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Как видим, третий столбец матрицы полностью обращает z -координату в ноль.

Программная реализация

Программная реализация данной лабораторной весьма похожа на реализацию преобразований на плоскости. Дополнительно описываются матрица для получения проекций и матрица для хранения двумерного представления.

Вычислительный процесс, запускаемый при нажатии на какую-либо кнопку, приведен на рис. 10.4.



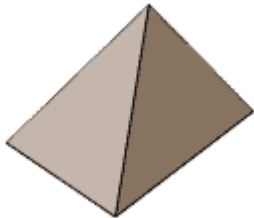
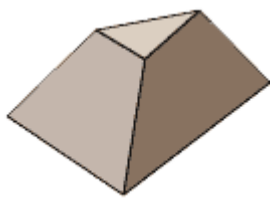
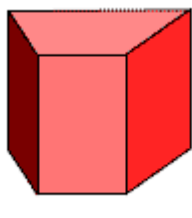
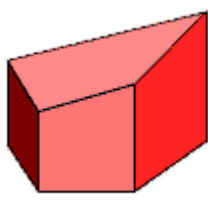
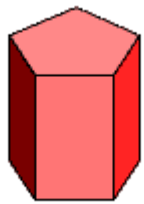
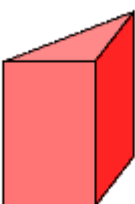
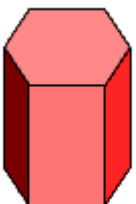
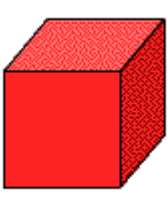
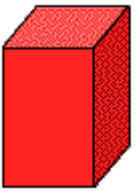
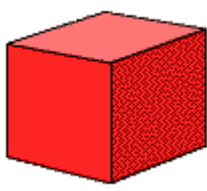
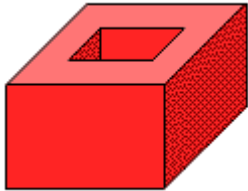
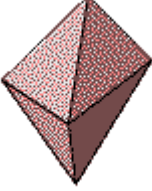
Рис. 10.4. Процесс визуализации трехмерного тела

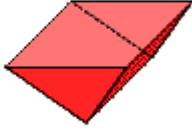
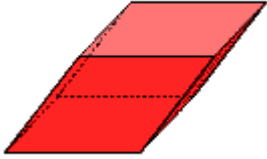
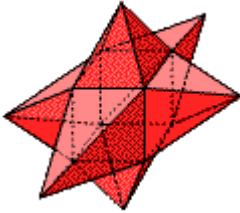
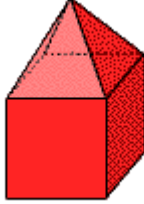
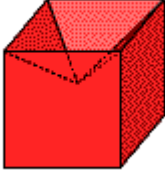
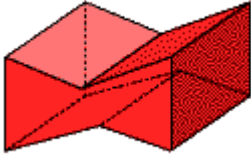
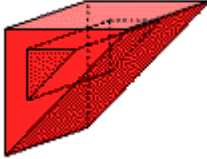
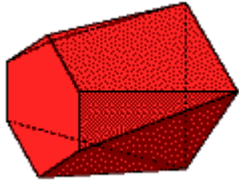
На четвертом этапе, при умножении на матрицу проецирования, исходная матрица тела не должна меняться, поскольку в этом случае будет потеряна информация о координате z . Поэтому результат умножения на этом этапе сохраняется в специальной матрице, по которой и происходит отрисовка.

Задание по лабораторной работе

Реализуйте с трехмерным телом все виды преобразований в пространстве: перенос вдоль координатных осей, отражение относительно основных плоскостей, масштабирование, поворот на заданные углы относительно координатных осей. Предусмотрите восстановление исходной позиции тела. Реализуйте получение следующих проекций: вид спереди, центральной односточечной, косоугольной. Управление организуйте как через интерфейсные элементы (меню, кнопки, строки редактирования и пр.), так и через горячие клавиши.

Варианты заданий

			
Задание № 1	Задание № 2	Задание № 3	Задание № 4
			
Задание № 5	Задание № 6	Задание № 7	Задание № 8
			
Задание № 9	Задание № 10	Задание № 11	Задание № 12

			
Задание № 13	Задание № 14	Задание № 15	Задание № 16
			
Задание № 17	Задание № 18	Задание № 19	Задание № 20

Контрольные вопросы

1. Что такое правосторонняя система координат?
2. Какая размерность матриц для преобразований в пространстве и для получения проекций?
3. Какие коэффициенты в матрице проецирования отвечают за получение перспективных проекций?
4. Какие этапы в процессе визуализации трехмерного тела?

Лабораторная работа 11 ПОСТРОЕНИЕ ТРЕХМЕРНЫХ СЦЕН

Цель: в соответствии с первой образовательной траекторией изучить принципы построения статических 3D-сцен с помощью XAML (eXtensible Application Markup Language) в WPF (windows presentation foundation) либо в соответствии со второй образовательной траекторией изучить возможности создания трехмерных сцен на JavaScript с помощью библиотеки ThreeJS; написать и протестировать программу для получения трехмерной сцены.

Общие сведения

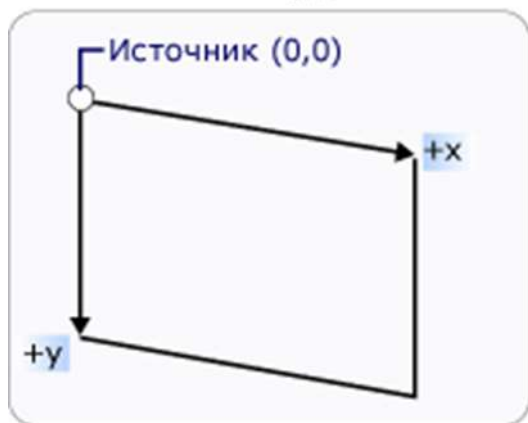
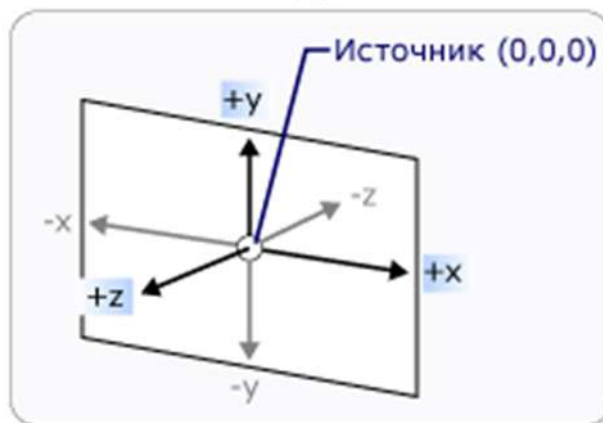
Построение 3D сцен в XAML

Система координат и размещение камеры в XAML. Реализация трехмерных сцен в приложении WPF возможно как в разметке XAML, так и в процедурном коде. В данной лабораторной работе рассмотрим возможности XAML.

Графическое содержимое трехмерных сцен в приложении WPF инкапсулировано в элементе `Viewport3D`, который может участвовать в структуре двумерного элемента. Графическая система рассматривает объект `Viewport3D` как двумерный визуальный элемент в ряду других элементов приложения WPF. Объект `Viewport3D` функционирует как окно просмотра трехмерной сцены. Говоря точнее, это поверхность, на которую проецируется 3D-сцена.

В традиционном приложении двумерный элемент `Viewport3D` используется как любой другой контейнерный элемент, например «Grid» или «Canvas».

Начало системы координат WPF для двумерной графики расположено в левом верхнем угле области отрисовки (обычно экрана или окна). В этом случае положительные значения оси x откладываются вправо, а положительные значения оси y – сверху вниз. Однако в трехмерной системе начало координат располагается в центре отрисовываемой области, положительные значения оси x откладываются вправо, оси y – снизу вверх, а оси z – из центра к наблюдателю (рис. 11.1). В WPF принята правосторонняя система координат, поэтому указание положительного значения угла поворота приведет к повороту против часовой стрелки вокруг оси.

2D-Система координат**2D-Система координат***Рис. 11.1. 2D- и 3D-система координат*

Поскольку трехмерная сцена выглядит по-разному в зависимости от точки наблюдения, необходимо указать эту точку наблюдения или, если сказать иначе, задать положение камеры. Для работы с камерой используется класс **Camera**. Для описания способа проецирования используются классы **OrthographicCamera** и **PerspectiveCamera** (для получения ортографической и перспективной проекции соответственно).

Свойства **NearPlaneDistance** и **FarPlaneDistance** камеры определяют расстояния до передней секущей плоскости и задней секущей плоскости области видимого объема.

Рассмотрим пример описания получения перспективной проекции⁵:

```
<Viewport3D.Camera>
  <PerspectiveCamera
    Position="-250,250,200"
    LookAtPoint="0,0,0"
    UpDirection="0,1,0"
    FieldOfView="40"
    NearPlaneDistance="1"
    FarPlaneDistance="500"
  />
```

Свойство **Position** устанавливает позицию камеры в мировых координатах (центр проекции). Свойство **LookAtPoint** задает вектор, который определяет направление просмотра камеры в мировых координатах. Свойство **UpDirection** задает объект вектор, определяющий направление вверх для камеры. Свойство **FieldOfView** определяет угол между боковыми секущими плоскостями видимого объема.

⁵ При получении перспективной проекции определяется центр проекции, в котором и размещается камера.

Освещение сцены в XAML. Источники света в трехмерной графике выполняют ту же роль, что и реальные источники света: они делают поверхности видимыми. Более того, источники света определяют, какая часть сцены будет включена в проекцию. Объекты источников света в приложении WPF создают различные эффекты света и тени. Они смоделированы на основе поведения различных реальных источников света. Сцена должна включать как минимум один источник света, иначе модели будут невидимыми.

Указанные ниже источники света являются производными от базового класса **Light**.

AmbientLight создает рассеянное освещение, при котором все объекты освещены одинаково, независимо от их расположения или ориентации.

DirectionalLight создает освещение, аналогичное удаленному источнику света. Другими словами, это свойство создает прямые параллельные лучи, которые идут в направлении, указанном в свойстве **Direction** (объект **Vector3D**).

PointLight создает освещение как от ближнего источника света. Источники света **PointLights** занимают определенное положение и испускают свет из этого положения. Объекты на сцене освещаются в зависимости от их положения и расстояния относительно источника света. **PointLightBase** предоставляет свойство **Range**, которое определяет расстояние, за пределами которого модели не будут освещены светом. Класс **PointLight** также предоставляет свойства затухания, то есть интенсивность ослабления источника света в зависимости от расстояния. Можно указать константу, линейную или квадратичную интерполяцию затухания источника света.

Через свойство **Color** устанавливается цвет источника освещения.

Источники света являются объектами **Model3D**, поэтому можно преобразовывать и анимировать свойства источников света, включая положение, цвет, направление и диапазон.

Пример задания направленного источника белого света приведен ниже.

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <DirectionalLight Color="#FFFFFF"
      Direction="-1,-1,-1" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

Задание объектов трехмерной сцены в XAML. Для задания объектов на трехмерной сцене используется класс **GeometryModel3D**, в котором можно определить полигональную сетку, состоящую из треугольников, с

помощью класса **MeshGeometry3D**. Каждый треугольник в полигональной сетке представляется с помощью индексов в списке вершин. Список вершин определяется свойством **Positions**, а набор примитивов задается через свойство **TriangleIndices**.

Приведем пример:

```
<GeometryModel3D>
  <GeometryModel3D.Geometry>
    <MeshGeometry3D
      Positions="-1 -1 0 1 -1 0 -1 1 0 1 1 0"
      TriangleIndices="0 1 2 1 3 2" />
    </GeometryModel3D.Geometry>
  </GeometryModel3D>
```

В итоге на сцене представлены два треугольника с двумя общими вершинами (с индексами «0» и «1»). В списке вершин находится четыре вершины.

При построении полигональной сетки важен порядок перечисления вершин для каждого полигона. Для видимых граней этот порядок должен идти по направлению движения часовой стрелки, для невидимых – в другую сторону. Порядок перечисления вершин позволяет вычислить нормали для полигонов, что необходимо для удаления невидимых граней и расчета освещенности. Однако нормали можно задать также вручную через свойство **Normals**.

Применение материалов к модели в XAML. Для определения характеристик поверхности модели приложение WPF использует абстрактный класс **Material**. Конкретные подклассы класса **Material** задают некоторые характеристики внешнего вида поверхности модели, и каждый из них предоставляет свойство **Brush** (кисть), которому можно передать значение **SolidColorBrush**, **TileBrush** или **VisualBrush**.

Класс **DiffuseMaterial** определяет, что кисть будет применена к модели, как если бы она была освещена рассеянным светом. Поверхности, описанные с помощью класса **DiffuseMaterial**, не являются блестящими и не отбрасывают блики.

Класс **SpecularMaterial**, наоборот, позволяет описать гладкие поверхности, которые способны отбрасывать блики. Можно установить степень гладкости или «глянца» текстуры, задав значение свойства **SpecularPower**.

Класс **EmissiveMaterial** позволяет указать, что текстура будет применена, как если бы модель излучала свет, эквивалентный цвету кисти. Это не делает модель светящейся; однако это иначе влияет на затенение, чем если бы текстура была создана с помощью класса **DiffuseMaterial** или **SpecularMaterial**.

Для повышения производительности противоположные поверхности объекта **GeometryModel3D** (границы, которые невидимы, поскольку находятся на противоположной стороне модели относительно камеры) удаляются из сцены. Чтобы указать класс **Material** для применения к противоположной поверхности модели, например плоскости, задайте свойство **BackMaterial** модели.

Следующий пример кода демонстрирует применение сплошного цвета и использование полупрозрачности.

```
<GeometryModel3D.Material>
  <DiffuseMaterial>
    <DiffuseMaterial.Brush>
      <SolidColorBrush Color="Cyan" Opacity="0.3"/>
    </DiffuseMaterial.Brush>
  </DiffuseMaterial>
</GeometryModel3D.Material>
```

Трехмерные преобразования в XAML. Каждый объект модели имеет свойство **Transform**, с помощью которого можно перемещать модель, изменять ее направление или размер. Для трехмерных преобразований используются классы **TranslateTransform3D** (преобразование переноса), **ScaleTransform3D** (масштабирование) и **RotateTransform3D** (поворот). WPF также предоставляет класс **MatrixTransform3D**, который позволяет указать те же преобразования в более коротких матричных операциях.

TranslateTransform3D перемещает все точки в **Model3D** в направлении выбранного вектора смещения со свойствами **OffsetX**, **OffsetY** и **OffsetZ**.

ScaleTransform3D изменяет масштаб модели с помощью указанного вектора масштаба относительно центральной точки. По умолчанию **ScaleTransform3D** растягивает или сжимает вершины по отношению к точке отсчета (0,0,0). Однако если преобразуемая модель строится не от начала координат, то при ее масштабировании от начала координат она будет находиться «не на своем месте». В то же время, если вершины модели умножаются на вектор масштабирования, операция масштабирования приведет и к преобразованию модели, и к ее масштабированию.

Трехмерную модель можно поворачивать несколькими способами. При обычном преобразовании поворота указывается ось и угол поворота вокруг этой оси. Класс **RotateTransform3D** позволяет определить **Rotation3D** со свойством **Rotation**. Затем для задания преобразования следует указать свойства **AxisAngle** в **Rotation3D**.

Приведем пример поворота вокруг оси y на 40°.

```
<GeometryModel3D.Transform>
  <RotateTransform3D>
```

```

        <RotateTransform3D.Rotation>
            <AxisAngleRotation3D Axis="0,1,0" Angle="40" />
        </RotateTransform3D.Rotation>
    </RotateTransform3D>
</GeometryModel3D.Transform>

```

Пример описания простой трехмерной сцены на XAML. Приведем полный пример описания простой трехмерной сцены, состоящий из источника света и одного повернутого полигона.

```

<Window x:Class="WpfApplication1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300">
    <Grid>
        <!-- Viewport3D определяет поверхность и область,
            куда проецируется сцена. -->
        <Viewport3D ClipToBounds="True" Width="150" Height="150">

            <!-- Определяем положение и направление камеры. -->
            <Viewport3D.Camera>
                <PerspectiveCamera Position="0,0,2" LookDirection="0,0,-1"
                    FieldOfView="45" />
            </Viewport3D.Camera>

            <!-- Определяем дочерние объекты для Viewport3D -->
            <Viewport3D.Children>

                <!-- Устанавливаем направление
                    и цвет источника освещения. -->
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <DirectionalLight Color="#FFFFFF"
                            Direction="-1,-1,-1" />
                    </ModelVisual3D.Content>
                </ModelVisual3D>

                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <GeometryModel3D>

                            <!-- Определяем 3D объект -->
                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D
                                    TriangleIndices="0,1,2"
                                    Positions="-0.5,-0.5,0.5 0.5,-0.5,0.5
                                        0.5,0.5,0.5">
                                </MeshGeometry3D>
                            </GeometryModel3D.Geometry>

```



```

        <!-- Зададим материал (цвет) объекта -->
        <GeometryModel3D.Material>
            <MaterialGroup>
                <DiffuseMaterial Brush="Blue" />
            </MaterialGroup>
        </GeometryModel3D.Material>

        <!-- повернем объект на 40 градусов
        вокруг оси Y -->
        <GeometryModel3D.Transform>
            <RotateTransform3D>
                <RotateTransform3D.Rotation>
                    <AxisAngleRotation3D Axis="0,1,0"
                    Angle="40" />
                </RotateTransform3D.Rotation>
            </RotateTransform3D>
        </GeometryModel3D.Transform>

        </GeometryModel3D>
        </ModelVisual3D.Content>
    </ModelVisual3D>
</Viewport3D.Children>
</Viewport3D>
</Grid>
</Window>

```

Создание 3D-сцены с помощью библиотеки ThreeJS

Альтернативой XAML и WPF являются различные библиотеки для работы с 3D-графикой, такие как WebGL и OpenGL. Эти библиотеки преимущественно низкоуровневые, поэтому часто используют библиотеки более высокого уровня, позволяющие автоматизировать рутинную работу. Например, библиотека ThreeJS, которая является надстройкой над WebGL. Three.js, – это мощная библиотека для визуализации сложных и интерактивных 3D-сцен на языке JavaScript для Web. С помощью Three.js можно разрабатывать игры, приложения, 3D-модели и другие интерактивные элементы.

Библиотеку Three.js можно скачать с официального сайта библиотеки по адресу <https://threejs.org/>. Все, что официально связано с проектом three.js, находится в репозитории: исходный код, сотни примеров, демонстрирующих, как использовать каждую часть библиотеки, документация, интерактивный редактор сцен и огромное количество плагинов и бесплатных ресурсов, таких как 3D-модели, текстуры, звуки и 3D-шрифты.

Папка **build** является самой важной папкой в репозитории GIT, поскольку она содержит основные файлы библиотеки.

Библиотека предоставляется в нескольких вариантах:

- **three.module.js** – модуль с полным текстом библиотеки. Модули не работают локально, только через HTTP(s). Если попытаетесь открыть веб-страницу локально, через протокол `file://`, то обнаружите, что директивы **import/export** не работают. Для тестирования модулей используйте локальный веб-сервер;

- **three.module.min.js** – модуль с полным текстом библиотеки, где удалены все комментарии и отступы. Представляет из себя одну длинную практически не читаемую строку;

- **three.js** – полный текст библиотеки, который может быть размещен как на сервере, так и локально. Отладка такого кода возможна без локального сервера. Будет поддерживаться до 160 сборки, для совместимости со старыми браузерами;

- **three.min.js** – аналог `three.js` с минимизированным текстом библиотеки, из которого удалены пробелы и комментарии.

Установка и настройка ThreeJS в VS Code. Одним из самых удачных редакторов JavaScript кода является бесплатный редактор кода Visual Studio Code (<https://code.visualstudio.com/>). Для создания проекта, работающего с ThreeJS, необходимо создать папку, в которой должен будет храниться как минимум один HTML-документ, определяющий Web-страницу, и желательно Javascript-файл с вызовами методов библиотеки ThreeJS.

В папку проекта можно скачать и сохранить файлы библиотеки, например **three.module.js** с Github из папки `build` (<https://github.com/mrdoob/three.js/tree/dev/build>), либо обратиться к ним, используя подобные URL – <https://threejs.org/build/three.module.js>. Подготовьте **index.html** и пустой **js**-файл в папке будущего проекта. Желательно, чтобы путь к файлам не содержал кириллицы. В эту папку скопируйте файл **three.module.js**. Далее откройте папку проекта в VS Code командой «открыть папку» (`open folder`).

Для подключения автозаполнения (**IntelliSense**) рекомендуется использовать специальный менеджер пакетов **nmp** (`node package manager`), поставляемый вместе с NodeJS. Для работы этого пакета можно установить сам NodeJS с сайта <https://nodejs.org/en>. Проверить версию пакета NodeJS можно в терминале VS Code командой **node -v**, которая выведет информацию о версии. После этого можно использовать менеджер пакетов **nmp** для установки пакета ThreeJS командой **npm install --save three**, которую также вводят в терминале VS Code. Для корректной работы автозаполнения рекомендуется закрыть VS Code и заново его запустить.

После этого рекомендуется установить локальный сервер **Vite** для запуска и отладки проекта с помощью команды **npm install --save-dev vite** (рис. 11.2).

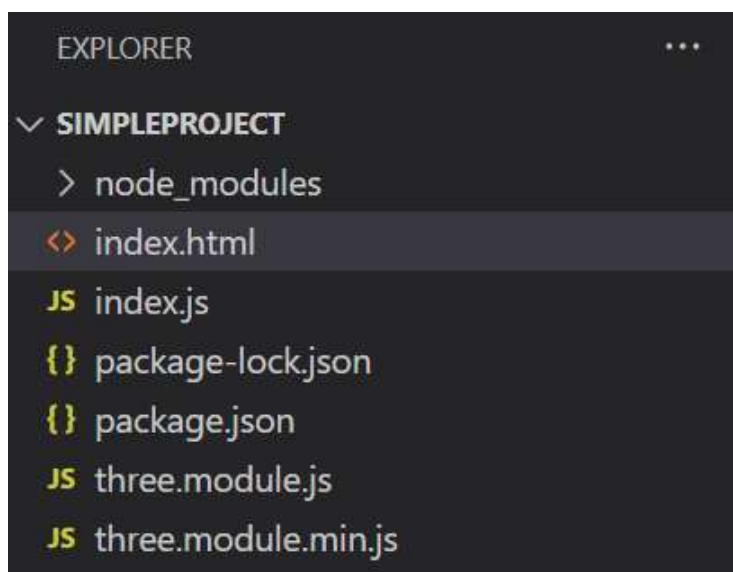


Рис. 11.2. Структура проекта для работы с ThreeJS

Папка **node_modules** содержит все пакеты, добавленные с помощью менеджера **npm**. Файл **package.json** является основным файлом для любого проекта Node.js. В нем хранится информация о проекте, включая его название, версию, зависимости, и другие сведения. В частности, файл **package.json** включает в себя зависимости, которые, в частности, реализуют автозаполнение. Файл **package-lock.json** автоматически генерируется для любых операций, в которых **npm** изменяет либо дерево **node_modules**, либо **package.json**.

Запуск сервера происходит командой **npm run dev**, которая запускает локальный сервер **http://localhost:5173** (рис. 11.3).

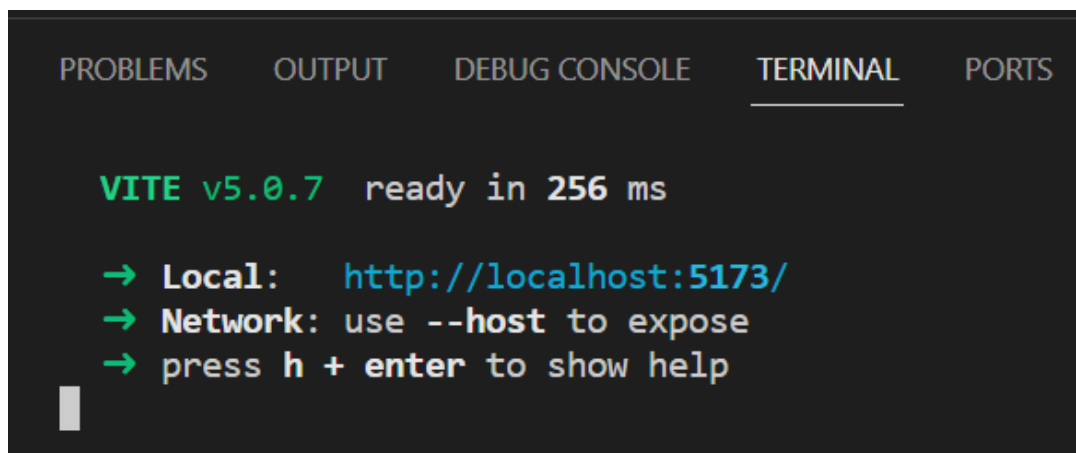


Рис. 11.3. Работа локального сервера Vite

После запуска сервера данный адрес можно использовать и открыть в любом браузере. Если ранее была подготовлена страница `index.html`, то можно просмотреть ее в браузере при вводе адреса `http://localhost:5173`. Для завершения работы локального сервера достаточно в терминале ввести **q**. Для получения справки по локальному серверу нажмите **h + enter**.

Необходимо отметить, что актуальная информация об установке библиотеки ThreeJS находится в разделе **Installation** сайта `threejs.org`.

Подключение библиотеки к HTML-файлу. Для использования библиотеки ThreeJS в HTML-документе должна быть описана область для вывода графики, так называемый холст или `canvas`. Так же в своем проекте ThreeJS необходимо подключить к HTML-документу. Для варианта подключения через локальный сервер используется скрипт карты импорта. И в завершении необходимо вызвать скрипт инициализации и отрисовки сцены из `js` файла. Код HTML-документа будет выглядеть так:

```
<!-- Описание холста -->
<canvas id="c"></canvas>
<!-- Подключение библиотеки из файла three.module.js -->
<script type="importmap">{
  "imports": {
    "three": "three.module.js"
  }
}</script>
<!-- Подключение js-файла -->
<script type="module" src="index.js"> </script>
```

Основные компоненты 3D-сцены в Three.js:

- сцена (**scene**) – это контейнер, в котором размещаются все объекты сцены;
- камера (**camera**) – это объект, который определяет точку обзора сцены;
- объект (**object**) – это любой элемент сцены, который может быть отрисован. Объекты могут быть простыми геометрическими фигурами, такими как грань, куб, сфера, цилиндр, или сложными моделями, загруженными из файла.

Для создания простой 3D-сцены в Three.js необходимо выполнить следующие шаги:

- 1) подключить библиотеку Three.js к проекту;
- 2) создать сцену;
- 3) создать камеру;
- 4) создать объект;
- 5) добавить объект в сцену;
- 6) создать `Renderer` (отрисовщик сцены, или средство визуализации).

Рассмотрим структуру js-файла, в котором реализуются данные шаги:

```
// Подключаем библиотеку Three.js
import * as THREE from 'three';

function main() {
  // Получаем доступ к холсту по его имени
  const canvas = document.querySelector( '#c' );

  // Создаем объект отрисовки (рендер)
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});

  // Создаем переменные для камеры
  const fov = 75;
  const aspect = 2; // the canvas default
  const near = 0.1;
  const far = 5;
  // Создаем камеру
  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
  camera.position.z = 2;

  // Создаем сцену
  const scene = new THREE.Scene();

  // Создаем геометрию для куба размером 1x1x1
  const boxWidth = 1;
  const boxHeight = 1;
  const boxDepth = 1;
  const geometry = new THREE.BoxGeometry(boxWidth, boxHeight,
                                          boxDepth);

  // Создаем материал для куба
  const material = new THREE.MeshBasicMaterial( { color: 0x0000FF } );

  // Создаем полигональную сетку для куба с нужным материалом
  const cube = new THREE.Mesh( geometry, material );

  //Добавляем куб на сцену
  scene.add(cube);

  function render(time) {
    time *= 0.001; // перевод времени в секунды

    //поворот куба вокруг осей X и Y
    cube.rotation.x = time;
    cube.rotation.y = time;
    // Вызов метода отрисовки в рендере
    // В метод передается сцена
    // со всеми объектами на ней и камера
    renderer.render(scene, camera);
  }
}
```

```

        // метод для организации следующей анимации
        // таким образом организуется бесконечный цикл отрисовки
        requestAnimationFrame(render);
    }
    // метод для организации анимации в JavaScript
    // параметром является имя функции, производящей отрисовку
    requestAnimationFrame(render);
}
//вызов главной функции
main();

```

Этот код создает сцену, в которой находится куб. Куб имеет размер $1 \times 1 \times 1$ и синий цвет. Камера находится на расстоянии 2 на оси Z.

Для более сложных сцен необходимо использовать дополнительные компоненты, такие как источники света, текстуры и материалы. Вот некоторые дополнительные компоненты, которые можно добавить в 3D-сценах на JavaScript:

- источники света (*light*) – это объекты, которые излучают свет. Источники света используются для освещения объектов в сцене;
- текстуры (*texture*) – это изображения, которые могут быть наложены на объекты. Текстуры используются для создания более реалистичного внешнего вида объектов;
- материалы (*material*) – это объекты, которые определяют свойства поверхности объектов. Материалы используются для задания цвета, прозрачности и отражения объектов.

Работа с камерой в ThreeJS. Как было сказано ранее, чтобы действительно иметь возможность отображать что-либо с помощью `three.js`, нужны три вещи: сцена, камера и средство визуализации (**renderer**). Рассмотрим ниже создание этих элементов на JavaScript:

```

import * as THREE from 'three';

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
    window.innerHeight, 0.1, 1000);
camera.position.set(0, 2.5, 2.5);

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

```

Как видим, данный пример демонстрирует иной подход связи с HTML-документом через использование метод **appendChild**, который вставляет дочерний элемент в структуру DOM HTML-документа. В этом

случае HTML-страница может не содержать холст (**canvas**) для отрисовки 3D-сцены:

```
<!-- Подключение библиотеки из файла three.module.js -->
<script type="importmap">{
  "imports": {
    "three": "three.module.js"
  }
}</script>
<!-- Подключение js файла -->
<script type="module" src="index.js"> </script>
```

Как видим, создание сцены через `new THREE.Scene()`; – достаточно тривиальная задача. Рассмотрим более подробно установку камеры и ее свойств. В ThreeJS можно применить камеру **PerspectiveCamera** для построения сцен с получением перспективных проекций и камеру **OrthographicCamera** для ортографической проекции. В обоих случаях использование камер предполагает их связь с видимым объемом (рис. 11.4).

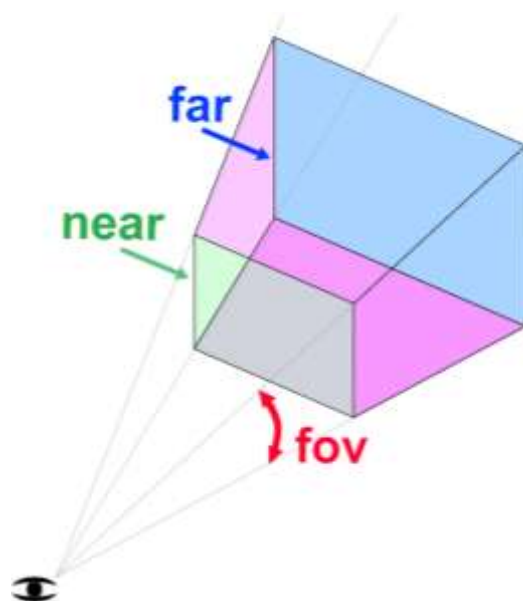


Рис. 11.4. Видимый объем для перспективной проекции

Для перспективной проекции видимый объем представляется в виде усечённой пирамиды (**frustum**) и задаётся четырьмя параметрами:

- **near** – определяет расстояние до передней секущей плоскости;
- **far** – определяет расстояние до задней секущей плоскости;
- **fov** (field of view) – задает угол между проекторами при построении видимого объема (рис.). Обычно угол задают между 60 и 90°;
- **aspect ratio** – задает соотношение сторон. Почти всегда требуется использовать ширину элемента, деленную на высоту, иначе получим искажение пропорций по горизонтали или вертикали.

Конструктор **PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000)** первым параметром содержит угол поля зрения (**fov**), вторым – соотношение сторон, далее – данные по передней и задней секущим плоскостям (**near** и **far**).

Для ортографической проекции используется конструктор **OrthographicCamera(left, right, top, bottom, near, far)**, который определяет видимый объем в виде прямоугольного параллелепипеда. Кроме этих двух самых популярных камер ThreeJS поддерживает ряд других, в том числе и стереоскопическую камеру.

По умолчанию камера создается в начале координат, как и 3D-объекты. Поэтому после создания камеры требуется установить позицию камеры. Для расположения камеры можно использовать свойство `position` и его свойства **x**, **y**, **z** либо метод **set**:

```
camera.position.x = 0;  
camera.position.y = 0;  
camera.position.z = 5;
```

или

```
camera.position.set(0, 2.5, 2.5);
```

Также камеру можно повернуть вокруг осей методами `rotateX`, `rotateY`, `rotateZ` или, например, вокруг оси *x*, через свойство `rotation`: **camera.rotation.x = -89.5;**

Через свойство `up` задается верх камеры:

```
camera.up.x = 0;  
camera.up.y = 0;  
camera.up.z = 1;
```

И наконец, куда направлена камера? По умолчанию это начало координат, но если надо изменить направление камеры, то используется метод **lookAt**:

```
camera.lookAt(new THREE.Vector3(0,0,0));
```

Создание и размещение 3D-объектов на сцене. Для работы с 3D-объектами в ThreeJS существует два понятия: геометрия (**Geometry**), которая определяет форму объекта (куб, цилиндр, конус и т. п.), и полигональная сетка (**Mesh**), которая накладывается на геометрию и содержит информацию о материале (**Material**). Нижеследующий код демонстрирует создание куба сначала как геометрию, а потом создание полигональной сетки на основе этой геометрии и простого материала:


```
const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

Далее куб добавляется на сцену методом `add`: **`scene.add(cube)`**;

Геометрия объекта состоит из вершин (**`vertices`**) и граней (**`faces`**).

Для создания 3D-объектов в ThreeJS существует несколько классов:

- **`BoxGeometry`** – класс для создания куба;
- **`CapsuleGeometry`** – класс для создания капсулы, которая состоит из цилиндра и двух полусфер;
- **`CircleGeometry`** – класс для создания круга;
- **`ConeGeometry`** – класс для создания конуса;
- **`CylinderGeometry`** – класс для создания цилиндра;
- **`DodecahedronGeometry`** – класс для создания додекаэдра;
- **`EdgesGeometry`** – класс для создания ребер;
- **`ExtrudeGeometry`** – класс для создания 3D-объекта методом выдавливания из пути, описанного классом **`Shape`**;
- **`IcosahedronGeometry`** – класс для создания икосаэдра;
- **`LatheGeometry`** – класс для создания 3D-тела выглядящего как ваза;
- **`OctahedronGeometry`** – класс для создания октаэдра;
- **`PlaneGeometry`** – класс для создания плоского квадрата;
- **`PolyhedronGeometry`** – класс для создания многогранника. Этот класс задает массив вершин, проецирует их на сферу, а затем делит их до желаемого уровня детализации. Этот класс используется геометрией додекаэдра, икосаэдра, октаэдра и тетраэдра для создания соответствующих геометрий;
- **`RingGeometry`** – класс для создания плоского кольца;
- **`ShapeGeometry`** – класс для создания односторонней геометрии из одного или нескольких путей;
- **`SphereGeometry`** – класс для создания сферы;
- **`TetrahedronGeometry`** – класс для создания тетраэдра;
- **`TorusGeometry`** – класс для создания тора;
- **`TorusKnotGeometry`** – класс для создания торического узла;
- **`TubeGeometry`** – класс для создания трубы, которая выдавливается вдоль трехмерной кривой;
- **`WireframeGeometry`** – класс для создания каркасной геометрии.

Подробные параметры создания объектов этих классов даны в документации <https://threejs.org/docs/>.

По аналогии с позицией камеры можно задавать местоположение объектов на сцене через свойство `position` и поворот через `rotation`. Так, следующий код приведет к построению сцены на рис. 11.5:

```
const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
cube.position.x = 2;
scene.add(cube);

const myConeGeometry = new THREE.ConeGeometry(1, 2, 20);
const cone = new THREE.Mesh(myConeGeometry, material);
cone.position.x = -2;
scene.add(cone);

const myTorGeometry = new THREE.TorusGeometry(1, 0.2, 16, 100);
const tor = new THREE.Mesh(myTorGeometry, material);
tor.position.y = 2;
tor.rotation.x = 55;
scene.add(tor);
```

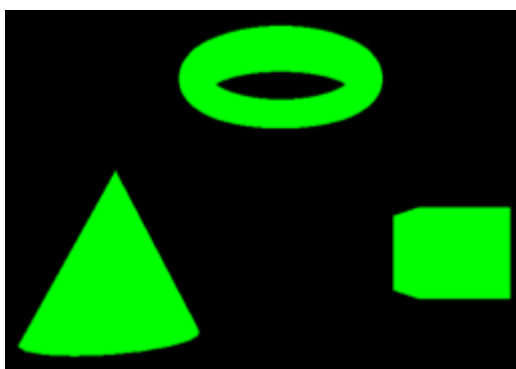


Рис. 11.5. Сцена с кубом, конусом и тором

Визуализация сцены на ThreeJS. Для визуализации сцены используется класс **WebGLRenderer**, для которого через метод **setSize** определяется область вывода:

```
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
```

Далее необходимо организовать цикл визуализации или рендеринга, который будет срабатывать каждый раз при выводе нового кадра отрисовки. Обычно такой цикл может срабатывать 60 раз за секунду. Опишем и вызовем первый раз функцию **animate()** для организации такого цикла.

```
function animate() {
    requestAnimationFrame(animate);
    renderer.render(scene, camera);
}
animate();
```

Функция **requestAnimationFrame(animate)**; вызывает функцию **animate()** каждый раз после прорисовки текущего кадра, а точнее – планирует запуск функции **animate()** на ближайшее время, когда браузер сочтёт возможным осуществить прорисовку. Таким образом реализуется цикл визуализации. Кроме этого, в теле функции вызывается метод **render** класса **WebGLRenderer** для визуализации сцены. В этот метод передается сама сцена и камера.

Полный и рабочий текст примеры выглядит следующим образом:

- код HTML:

```
<!-- Подключение библиотеки из файла three.module.js -->
<script type="importmap">{
  "imports": {
    "three": "three.module.js"
  }
}</script>
<!-- Подключение js-файла -->
<script type="module" src="index.js"> </script>
```

- код на JavaScript:

```
import * as THREE from 'three';
10

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
    window.innerHeight, 0.1, 1000 );

camera.position.z = 5;

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });

const cube = new THREE.Mesh(geometry, material);
cube.position.x = 2;
scene.add(cube);

const myConeGeometry = new THREE.ConeGeometry(1, 2, 20);
const cone = new THREE.Mesh(myConeGeometry, material);
cone.position.x = -2;
scene.add(cone);

const myTorGeometry = new THREE.TorusGeometry(1, 0.2, 16, 100);
const tor = new THREE.Mesh(myTorGeometry, material);
tor.position.y = 2;
tor.rotation.x = 55;
```

```

scene.add(tor);

function animate() {
    requestAnimationFrame(animate);
    renderer.render(scene, camera);
}
animate();

```

Освещение и тени в ThreeJS. Без освещения на сцене будет складываться впечатление, что наблюдатель находится в темной комнате. Помимо этого, с помощью освещения сцене можно придать большую реалистичность. В ThreeJS существует несколько типов источников освещения. Наиболее популярные приведены на рис. 11.6.

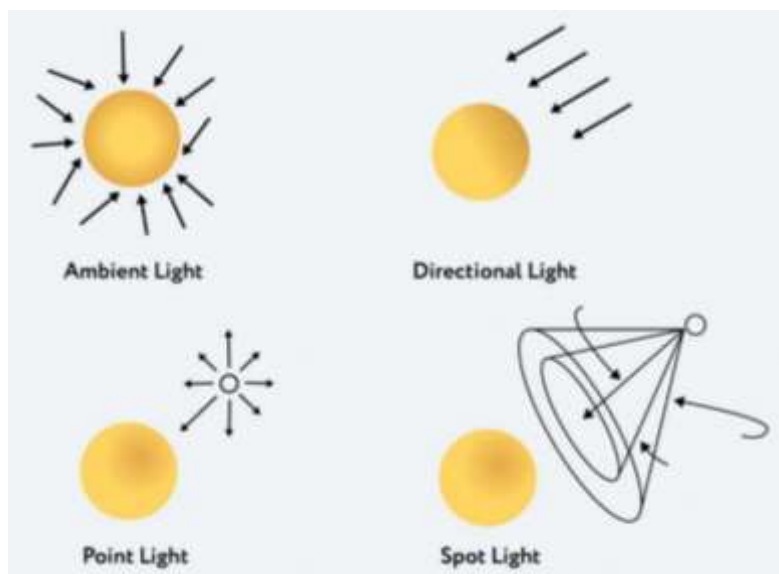


Рис. 11.6. Типы источников освещения в ThreeJS

Ambient Light – фоновое освещение, которое используется для освещения всех объектов сцены одинаково со всех сторон. Такой источник света не может быть использован для создания теней, так как не имеет направления.

Directional Light – свет, который излучается в определенном направлении. Этот свет будет вести себя так, как если бы он был бесконечно далеко, а лучи, излучаемые из него, были параллельны. Данный источник света может отбрасывать тени, так как направлен на конкретный объект.

Point Light – точечный источник света, который излучает свет из одной точки во всех направлениях.

Spot Light – свет, который излучается из одной точки в одном направлении, вдоль конуса, расширяемого по мере удаления от источника света.

Примеры работы различных источников освещения можно найти в соответствующем разделе документации: <https://threejs.org/manual/#en/lights>.

Для того чтобы объекты на сцене освещались корректно, нужно выбрать модель расчета освещения на полигональной сетке. Для этого необходимо перейти к материалам, поддерживающим методы закраски Фонга или Ламберта. Так, для вышеприведенного примера заменим базовый материал для всех 3D-объектов на материал с поддержкой расчета интенсивностей отраженного света методом Фонга:

```
const material = new THREE.MeshPhongMaterial({color: 0xFFFFFF});
```

Далее установим точечный источник света в ту же точку, где находится камера:

```
const color = 0x00FFFF; //цвет источника света
const intensity = 150; //интенсивность источника света
const light = new THREE.SpotLight(color, intensity);
light.position.z = 5;
```

Результат освещения сцены голубым (Cyan) источником света представлен на рис. 11.7.

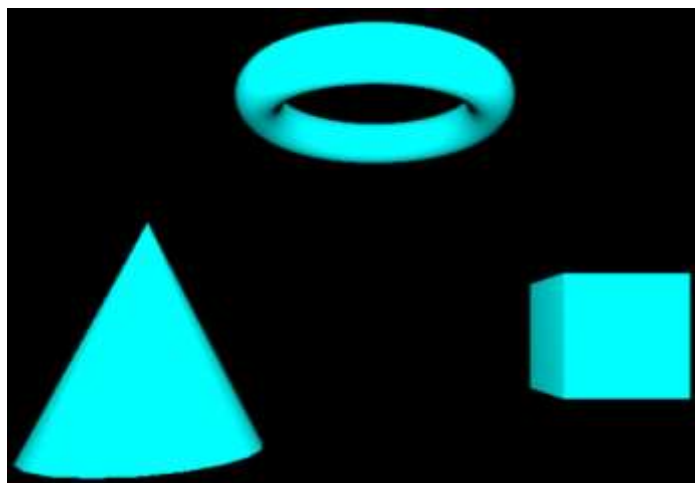


Рис. 11.7. Сцена с точечным источником света

Для получения теней в сцене необходимо, во-первых, включить рендеринг теней, так как по умолчанию он не включен:

```
renderer.shadowMap.enabled = true;
```

Во-вторых, необходимо установить свойство **castShadow** в **true** для источника света и для всех объектов, которые будут отбрасывать тени:

```
light.castShadow = true;
cube.castShadow = true;
cone.castShadow = true;
tor.castShadow = true;
```

В-третьих, необходимо установить свойство **receiveShadow** в **true** для всех объектов, на которые будут отбрасываться тени. Для данного примера создадим плоскость размером 10 на 10 единиц, на которую и будут отбрасываться тени:

```
const myPlaneGeometry = new THREE.PlaneGeometry(10, 10);
const plane = new THREE.Mesh(myPlaneGeometry, material);
plane.receiveShadow = true;
plane.position.z = -2;
scene.add(plane);
```

В-четвертых, для изменения разрешения тени можно установить свойства **light.shadow.mapSize.width** и **light.shadow.mapSize.height**:

```
light.shadow.mapSize.width = 5000;
light.shadow.mapSize.height = 5000;
```

По умолчанию эти свойства установлены 512 на 512. Результат приведен на рис. 11.8.

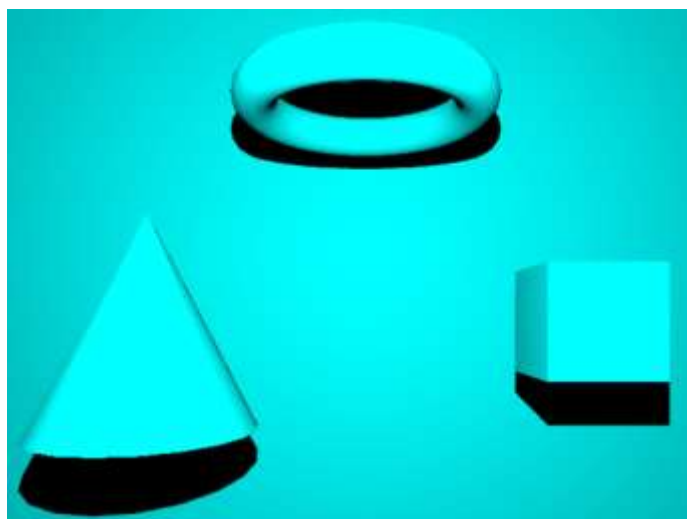


Рис. 11.8. Сцена с тенями

Для получения более полной информации по теням обратитесь к документации: <https://threejs.org/manual/#en/shadows>.

Материалы в ThreeJS. В предыдущем разделе уже рассмотрена необходимость использования материалов для получения теней на примере материала для расчета отраженного света по методу Фонга. Познакомимся с дополнительными материалами, которые предлагает библиотека ThreeJS.

MeshNormalMaterial – многоцветный материал, цвет которого зависит от направления вектора нормали к поверхности. Данный материал задается так:

```
const material = new THREE.MeshNormalMaterial();
```

Результат использования многоцветного материала для тора приведен на рис. 11.9.

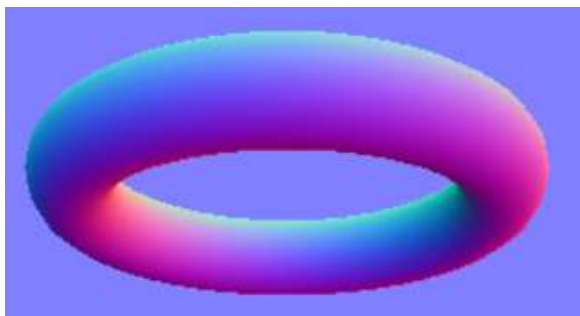


Рис. 11.9. Тор с *MeshNormalMaterial* материалом

MeshBasicMaterial – основной материал, который, например, полезен при отображении скелета объекта:

```
const material = new THREE.MeshBasicMaterial({
  wireframe: true,
  color: 0xdaa520
});
```

Результат использования такого материала для конуса приведен на рис. 11.10.

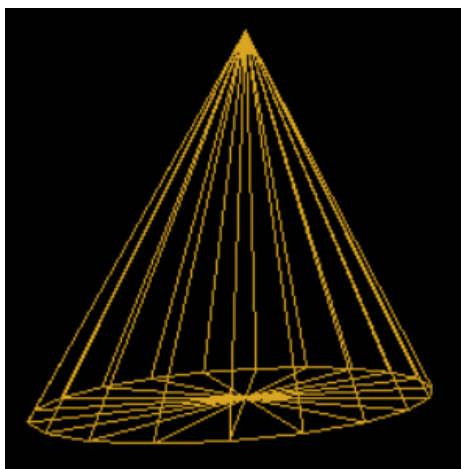


Рис. 11.10. Конус с *MeshBasicMaterial* материалом

MeshLambertMaterial – материал с расчетом отраженного света по методу Ламберта (с интерполяцией нормалей).

```
const material = new THREE.MeshLambertMaterial({
  color: 0xdaa520,
  emissive: 0x111111,
});
```

Параметр **emissive** – это цвет объекта, исходящий от самого объекта (без источника света). В отличие от метода Фонга, который интерполирует

значение интенсивностей отраженного света, метод Ламберта обеспечивает лучшую отрисовку, но требует больших вычислительных затрат.

```
const material = new THREE.MeshStandardMaterial({
  color: 0xfcc742,
  emissive: 0x111111,
  specular: 0xffffffff,
  metalness: 1,
  roughness: 0.55,
});
```

MeshStandardMaterial обеспечивает максимальное качество, хотя и влечет большие вычислительные издержки. **MeshStandardMaterial** используется с дополнительными параметрами **metalness** и **roughness**, каждый из которых принимает значение между 0 и 1.

Параметр **metalness** влияет на то, как материал отражает свет, его глянецовость, то есть возможность формирования отраженного света с учетом доли зеркального отражения. **Roughness** добавляет дополнительный слой для кастомизации. Можно представить его как противоположность глянецовости: 0 – очень гляцевый, 1 – очень матовый.

Тор для этого примера представлен на рис. 11.11.

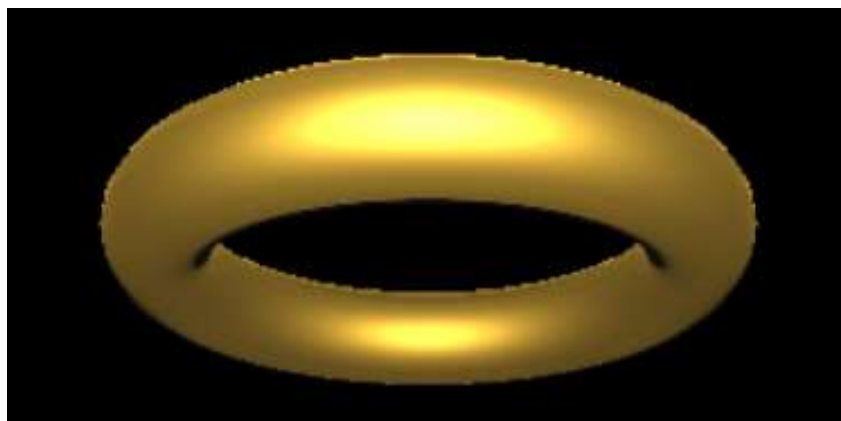


Рис. 11.11. Тор с использованием **MeshStandardMaterial** материала

Задание по лабораторной работе

Напишите программу с помощью средств XAML или библиотеки **ThreeJS** в соответствии с вариантом задания.

Варианты заданий

1. Создайте 3D-сцену с ёлочкой. Ветви елочки представляют собой конусы, которые генерируются в цикле. Добавьте на ёлочку украшения (различные 3D-объекты) из различных материалов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

2. Создайте 3D-сцену с домиком и забором вокруг. Забор представляет собой 3D-объекты, которые генерируются в цикле. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

3. Реализуйте 3D-сцену со снеговиком. Тело снеговика должно состоять из сфер, которые генерируются в цикле. Добавьте шапку в виде ведра, нос, глаза, пуговицы (различные 3D-объекты) из различных материалов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

4. Реализуйте 3D-сцену с машиной. Машина должна состоять из различных 3D-объектов. Используйте различные материалы. Установите на сцене не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

5. Создайте 3D-сцену с обеденным столом. На столе разместите тарелки, которые генерируются в цикле. Добавьте другие различные 3D-объекты из различных материалов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

6. Создайте 3D-сцену с рабочим столом. На столе разместите в случайном порядке книги, которые генерируются в цикле. Добавьте на стол другие различные 3D-объекты из различных материалов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

7. Реализуйте 3D-сцену с кораблем в волнах. Волны представляют собой полусферы различного размера, которые генерируются в цикле. Корабль состоит из различных 3D-объектов и различных материалов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

8. Создайте 3D-сцену с летающими свечами в Хогвартсе. Для визуализации свечи напишите собственную функцию. Каждая свеча в сцене является небольшим источником освещения. Свечи в 3D-пространстве генерируются в цикле. Добавьте другие различные 3D-объекты из различных материалов. В отчете приведите примеры этой сцены с различными положениями камеры.

9. Ребенку на день рождения подарили несколько сборных пирамидок. Создайте 3D-сцену, содержащую все собранные пирамидки. Пирамидки и кольца генерируются в цикле. Вершины пирамидок представлены различными 3D-телами. Кольца в пирамидках должны быть случайных цветов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

10. Создайте 3D-сцену с головой совы. Голова должна состоять из различных 3D-тел и материалов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

11. Реализуйте 3D-сцену с бильярдным столом. На столе располагаются бильярдные шары в случайном порядке, которые генерируются в цикле. На столе также лежат два кия различных цветов. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

12. Создайте 3D-сцену с сосульками на крае крыши. Сосульки генерируются в цикле и имеют разные цвета. Кроме этого, на сцене присутствует водосточная труба и кирпичная стена. Используйте не менее двух источников освещения. В отчете приведите примеры этой сцены с различными положениями камеры.

13. Реализуйте 3D-сцену с несколькими лестничными пролетами. Ступеньки и пролеты генерируются в цикле. На лестничных площадках расположите различные 3D-объекты из различных материалов. Используйте источники освещения на каждой лестничной площадке. В отчете приведите примеры этой сцены с различными положениями камеры.

14. Создайте 3D-сцену с шахматной доской с несколькими фигурами. Некоторые фигуры стоят, как минимум одна фигура лежит. Используйте не менее двух источников освещения. Фигуры должны отбрасывать тень. В отчете приведите примеры этой сцены с различными положениями камеры.

15. Реализуйте 3D-сцену со Стоунхенджем. Менгиры генерируются в цикле и должны отбрасывать тень. Горизонтальные менгиры добавьте вручную. На некоторых менгирах горят факелы, которые освещают сцену. В отчете приведите примеры этой сцены с различными положениями камеры.

16. Реализуйте 3D-сцену с пирамидой Кукулькана. Слои пирамиды генерируются в цикле. Кое-где на пирамиде горят факелы, которые освещают сцену. В отчете приведите примеры этой сцены с различными положениями камеры.

17. Создайте 3D-сцену с комнатой со стульями. Некоторые стулья стоят, некоторые – лежат. Для формирования стула сгруппируйте несколько объектов в иерархию. Добавьте в комнату различные 3D-объекты из различных материалов. Используйте не менее двух источников освещения. Фигуры должны отбрасывать тень. В отчете приведите примеры этой сцены с различными положениями камеры.

18. Создайте 3D-сцену с полянкой с грибами. Грибы должны быть различного размера, шляпки грибов – различных цветов. Грибы распола-

гаются на сцене в случайном порядке, для этого используйте цикл. Разместите на сцене не менее двух источников освещения. Фигуры должны отбрасывать тень. В отчете приведите примеры этой сцены с различными положениями камеры.

19. Создайте 3D-сцену с ожерельем и подвесом. В данном ожерелье используются не только бусины, но и другие 3D-тела из различных материалов. Все объекты располагаются по окружности и генерируются в цикле. Разместите на сцене не менее двух источников освещения. Фигуры должны отбрасывать тень. В отчете приведите примеры этой сцены с различными положениями камеры.

20. Создайте 3D-сцену с лугом с цветами. Каждый цветок имеет лепестки по кругу, которые генерируются в цикле. Цветы расставляются по лугу в цикле, но случайным образом. Цветы располагаются на ножках, которые имеют различный наклон. Используйте не менее двух источников освещения. Фигуры должны отбрасывать тень. В отчете приведите примеры этой сцены с различными положениями камеры.

Контрольные вопросы

Контрольные вопросы по первой образовательной траектории

1. Какой элемент инкапсулирует графическое содержимое трехмерных сцен в приложении WPF?
2. Что задают свойства **NearPlaneDistance** и **FarPlaneDistance** камеры?
3. Как задаются объекты трехмерной сцены в XAML?
4. Через какое свойство объектов производятся преобразования в пространстве?

Контрольные вопросы по второй образовательной траектории

1. Нужен ли какой-либо сервер для отладки проектов ThreeJS?
2. Какие основные компоненты 3D-сцены в Three.js вы знаете?
3. Как взаимосвязаны геометрия (Geometry) и полигональная сетка (Mesh)?
4. Каким образом реализуется цикл визуализации?

Лабораторная работа 12 ТРЕХМЕРНЫЕ ПРЕОБРАЗОВАНИЯ WPF И ThreeJS

Цель: изучить принципы построения динамических 3D-сцен с помощью процедурного кода в WPF или с помощью библиотеки ThreeJS; написать и протестировать программу для анимации трехмерной сцены, состоящей из нескольких объектов.

Общие сведения

Трехмерные преобразования в WPF

Связь процедурного кода и объектов, описанных в XAML. Среди пространств имен XAML можно выделить одно, которое объявляется почти в каждом файле XAML среды выполнения Windows, – это пространство имен для элементов, которые определены в языке XAML. По соглашению собственное пространство имен языка XAML сопоставляется с префиксом «x».

Префикс «x» (собственное пространство имен языка XAML) содержит несколько программных конструкций, которые часто используются в XAML-коде. Приведем только одну – **x:Name**. Она имя экземпляра объекта времени выполнения, созданного в исполняемом коде по итогам обработки элемента, который определяет этот объект в XAML-коде. Включение **x:Name** в код XAML можно рассматривать как объявление именованных переменных в коде.

Поэтому для доступа в процедурном коде к трехмерной модели необходимо включить **x:Name** в **ModelVisual3D**:

```
<ModelVisual3D x:Name="MyModel">  
...  
</ModelVisual3D>
```

Описанной **MyModel** можно будет далее оперировать в процедурном коде. Объект **MyModel** принадлежит классу **ModelVisual3D**.

Трехмерные преобразования в процедурном коде. Рассмотрим пример поворота объекта, созданного в XAML вокруг оси y. Предварительно подготовим необходимые объекты в обработчике события **Loaded**:

```
private void Window_Loaded(object sender, RoutedEventArgs e)  
{  
    myYRotate = new RotateTransform3D();  
    myYAxis = new AxisAngleRotation3D();  
    myYAxis.Axis = new Vector3D(0, 1, 0);  
    myYAxis.Angle = 7;  
    myYRotate.Rotation = myYAxis;  
    myTransform1 = new Transform3DGroup();  
    MyModel.Transform = myTransform1;  
}
```

MyYRotate, **myYAxis**, **myTransform1** описываются выше как поля класса **MainWindow**.

Класс **RotateTransform3D** отвечает за преобразование поворота. С помощью класса **AxisAngleRotation3D** задается ось, вокруг которой будет произведен поворот. Свойство **Axis** описывает направление оси через объект класса **Vector3D**. Свойство **Angle** задает угол поворота в градусах.

Далее создается объект **myTransform1** класса **Transform3DGroup**, который ставится в соответствие полю **Transform** класса 3D-модели. Этот класс удобно использовать для группы последовательных 3D-преобразований. Фактически **Transform3DGroup** представляет собой коллекцию объектов типа **Transform3D**. Отметим, что на данном этапе описанное преобразование поворота и 3D-модель пока никак не связаны.

Далее создадим на форме кнопку для поворота. Обработчик события этой кнопки будет выглядеть следующим образом:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    myTransform1.Children.Add(myYRotate);
}
```

Таким образом, при нажатии на кнопку в коллекцию преобразований с 3D-моделью будет добавляться очередной поворот. Вторым подходом, сокращающим количество элементов коллекции, является способ более раннего связывания 3D-модели с 3D-преобразованиями. Например:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    myYRotate = new RotateTransform3D();
    myYAxis = new AxisAngleRotation3D();
    myYAxis.Axis = new Vector3D(0, 1, 0);
    myYAxis.Angle = 0;
    myYRotate.Rotation = myYAxis;

    myXRotate = new RotateTransform3D();
    myXAxis = new AxisAngleRotation3D();
    myXAxis.Axis = new Vector3D(1, 0, 0);
    myXAxis.Angle = 0;
    myXRotate.Rotation = myXAxis;

    myTransform1 = new Transform3DGroup();

    MyModel.Transform = myTransform1;

    myTransform1.Children.Add(myYRotate);
    myTransform1.Children.Add(myXRotate);
}
```

При обработке события нажатия на кнопке в этом случае достаточно менять параметры созданных преобразований, например увеличивать угол поворота:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    myYAxis.Angle += 7;
    myXAxis.Angle += 7;
}
```

Применение 3D-преобразований к элементам 3D-сцены в WPF. Для применения различных 3D-преобразований к разным элементам 3D-сцены необходимо описать эти элементы как различные 3D-модели, то есть создать различные объекты класса **ModelVisual3D**.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="303" Width="312" Loaded="Win-
dow_Loaded">
    <Grid>
        <!-- Viewport3D определяет поверхность и
            область, куда проецируется сцена. -->
        <Viewport3D ClipToBounds="True" Width="150" Height="150">

            <!-- Определяем положение и направление камеры. -->
            <Viewport3D.Camera>
                <PerspectiveCamera Position="0,0,2" LookDirection="0,0,-1"
                    FieldOfView="45"/>
            </Viewport3D.Camera>

            <!-- Определяем дочерние объекты для Viewport3D -->
            <Viewport3D.Children>

                <!-- Устанавливаем направление и
                    цвет источника освещения. -->
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <DirectionalLight Color="#FFFFFF"
                            Direction="-1,-1,-1" />
                    </ModelVisual3D.Content>
                </ModelVisual3D>

                <!-- Определяем первую 3D-модель -->
                <ModelVisual3D x:Name="MyModel">
                    <ModelVisual3D.Content>
                        <GeometryModel3D>

                            <!-- Определяем 3D-объект -->
                            <GeometryModel3D.Geometry>
```

```

        <MeshGeometry3D
            TriangleIndices="0,1,2"
            Positions="-0.5,-0.5,0.5 0.5,-0.5,0.5
                0.5,0.5,0.5">
        </MeshGeometry3D>
    </GeometryModel3D.Geometry>
    <!-- Зададим материал (цвет) объекта -->
    <GeometryModel3D.Material>
        <MaterialGroup>
            <DiffuseMaterial Brush="Blue"/>
        </MaterialGroup>
    </GeometryModel3D.Material>

    <!-- повернем объект на 40 градусов
        вокруг оси Y -->
    <GeometryModel3D.Transform>
        <RotateTransform3D>
            <RotateTransform3D.Rotation>
                <AxisAngleRotation3D
                    Axis="0,1,0"
                    Angle="40" />
            </RotateTransform3D.Rotation>
        </RotateTransform3D>
    </GeometryModel3D.Transform>

    </GeometryModel3D>
</ModelVisual3D.Content>
</ModelVisual3D>

<!-- Определяем вторую 3D-модель -->
<ModelVisual3D x:Name="MyModel2">
    <ModelVisual3D.Content>
        <GeometryModel3D>

            <!-- Определяем 3D-объект -->
            <GeometryModel3D.Geometry>
                <MeshGeometry3D
                    TriangleIndices="0,1,2"
                    Positions="-0.5,-0.5,0.5 0.5,-0.5,0.5
                        0.5,0.5,0.5">
                </MeshGeometry3D>
            </GeometryModel3D.Geometry>

            <!-- Зададим материал (цвет) объекта -->
            <GeometryModel3D.Material>
                <MaterialGroup>
                    <DiffuseMaterial Brush="Red"/>
                </MaterialGroup>
            </GeometryModel3D.Material>
        </ModelVisual3D.Content>
    </ModelVisual3D>
</ModelVisual3D>

```

```

<!-- повернем объект на 90 градусов
      вокруг оси Z -->
<GeometryModel3D.Transform>
  <RotateTransform3D>
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D
        Axis="0,0,1"
        Angle="90" />
    </RotateTransform3D.Rotation>
  </RotateTransform3D>
</GeometryModel3D.Transform>
</GeometryModel3D>
</ModelVisual3D.Content>
</ModelVisual3D>

</Viewport3D.Children>

</Viewport3D>
<Button Content="Поворот 1-Y" Height="23" HorizontalAlign-
ment="Left" Margin="7,226,0,0" Name="button1" VerticalAlign-
ment="Top" Width="87" Click="button1_Click" />
<Button Content="Поворот 2-X" Height="23" HorizontalAlign-
ment="Left" Margin="103,226,0,0" Name="button2" VerticalAlign-
ment="Top" Width="80" Click="button2_Click" />
<Button Content="Поворот 1-X" Height="23" HorizontalAlign-
ment="Right" Margin="0,226,9,0" Name="button3" VerticalAlign-
ment="Top" Width="87" Click="button3_Click" />
</Grid>
</Window>

```

В данном случае каждая модель представлена всего одним полигоном. Результат приведен на рис. 12.1.

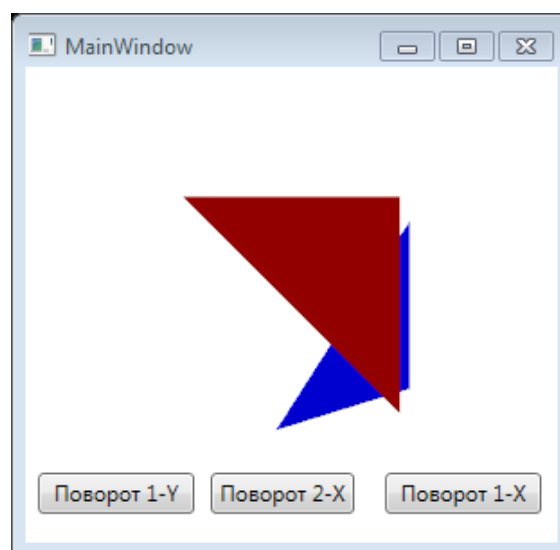


Рис. 12.1. 3D-сцена из двух 3D-моделей

Далее к каждой 3D-модели можно применять свои трехмерные преобразования:

```
private RotateTransform3D myYRotate, myXRotate;
private AxisAngleRotation3D myYAxis, myXAxis;
private Transform3DGroup myTransform1, myTransform2;

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    myYRotate = new RotateTransform3D();
    myYAxis = new AxisAngleRotation3D();
    myYAxis.Axis = new Vector3D(0, 1, 0);
    myYAxis.Angle = 7;
    myYRotate.Rotation = myYAxis;

    myXRotate = new RotateTransform3D();
    myXAxis = new AxisAngleRotation3D();
    myXAxis.Axis = new Vector3D(1, 0, 0);
    myXAxis.Angle = 7;
    myXRotate.Rotation = myXAxis;

    myTransform1 = new Transform3DGroup();
    myTransform2 = new Transform3DGroup();

    MyModel1.Transform = myTransform1;
    MyModel2.Transform = myTransform2;
}

private void button1_Click(object sender, RoutedEventArgs e)
{
    myTransform1.Children.Add(myYRotate);
}

private void button2_Click(object sender, RoutedEventArgs e)
{
    myTransform2.Children.Add(myXRotate);
}

private void button3_Click(object sender, RoutedEventArgs e)
{
    myTransform1.Children.Add(myXRotate);
}
}
```

Создание анимации с помощью таймера. В данном разделе рассмотрим создание простейшей анимации без использования специальных классов для анимации, таких как **Vector3DAnimation**, **Vector3DAnimationUsingKeyFrames** и пр. Анимация будет создаваться с помощью таймера.

Для работы с таймером в WPF используется класс **DispatcherTimer**, который инициализируется в процедурном коде следующим образом:

```
MyTimer = new DispatcherTimer();
MyTimer.Tick += new EventHandler(MyTimer_Tick);
MyTimer.Interval = new TimeSpan(100000);
```

Свойство **Interval** задает частоту срабатывания таймера. Через свойство **Tick** определяется метод (в приведенном примере – **MyTimer_Tick**), который будет запускаться по тикку таймера. Метод, запускающийся по тикку таймера, можно описать следующим образом:

```
private void MyTimer_Tick(object sender, EventArgs e)
{
}
```

Для запуска и остановки таймера необходимо использовать его методы **Start()** и **Stop()**.

Для задания различных трехмерных преобразований к различным объектам не только описываются разные 3D-модели, но и создаются различные объекты из класса **Transform3D**. Рассмотрим пример поворота двух граней вокруг осей *y* и *z* в разных направлениях. В отличие от предыдущего примера добавим цвет обратной стороны грани. XAML-код для элемента **Viewport3D** будет следующим:

```
<Viewport3D ClipToBounds="True" Width="300"
            Margin="102, 0, 102, 71">
  <!-- Определяем положение и направление камеры. -->
  <Viewport3D.Camera>
    <PerspectiveCamera Position="0,0,2" LookDirection="0,0,-1"
                      FieldOfView="45"/>
  </Viewport3D.Camera>
  <!-- Определяем дочерние объекты для Viewport3D -->
  <Viewport3D.Children>
    <!-- Устанавливаем направления и
           цвет источника освещения. -->
    <ModelVisual3D>
      <ModelVisual3D.Content>
        <DirectionalLight Color="#FFFFFF"
                          Direction="-1,-1,-1" />
      </ModelVisual3D.Content>
    </ModelVisual3D>
    <!-- Определяем первую 3D-модель -->
    <ModelVisual3D x:Name="MyModel">
      <ModelVisual3D.Content>
        <GeometryModel3D>
```

```

<!-- Определяем 3D-объект -->
<GeometryModel3D.Geometry>
  <MeshGeometry3D
    TriangleIndices="0,1,2"
    Positions="-0.1,-0.1,0.1 0.1,-0.1,0.1
              0.1,0.1,0.1">
  </MeshGeometry3D>
</GeometryModel3D.Geometry>

<!-- Зададим материал (цвет) объекта -->
<GeometryModel3D.Material>
  <MaterialGroup>
    <DiffuseMaterial Brush="Blue"/>
  </MaterialGroup>
</GeometryModel3D.Material>

<!-- Зададим цвет обратной стороны -->
<GeometryModel3D.BackMaterial>
  <MaterialGroup>
    <DiffuseMaterial Brush="Green"/>
  </MaterialGroup>
</GeometryModel3D.BackMaterial>

<!-- Перенесем объект влево -->
<GeometryModel3D.Transform>
  <TranslateTransform3D OffsetX="-0.5"/>
</GeometryModel3D.Transform>

  </GeometryModel3D>
</ModelVisual3D.Content>
</ModelVisual3D>

<!-- Определяем вторую 3D-модель -->
<ModelVisual3D x:Name="MyModel2">
  <ModelVisual3D.Content>
    <GeometryModel3D>

      <!-- Определяем 3D-объект -->
      <GeometryModel3D.Geometry>
        <MeshGeometry3D
          TriangleIndices="0,1,2"
          Positions="-0.1,-0.1,0.1 0.1,-0.1,0.1
                    0.1,0.1,0.1">
        </MeshGeometry3D>
      </GeometryModel3D.Geometry>

      <!-- Зададим материал (цвет) объекта -->
      <GeometryModel3D.Material>
        <MaterialGroup>
          <DiffuseMaterial Brush="Red"/>
        </MaterialGroup>
      </GeometryModel3D.Material>
    </ModelVisual3D.Content>
  </ModelVisual3D>
</ModelVisual3D>

```

```

        </MaterialGroup>
    </GeometryModel3D.Material>

    <!-- Зададим цвет обратной стороны -->
    <GeometryModel3D.BackMaterial>
        <MaterialGroup>
            <DiffuseMaterial Brush="Yellow"/>
        </MaterialGroup>
    </GeometryModel3D.BackMaterial>

    <!-- Перенесем объект вправо -->
    <GeometryModel3D.Transform>
        <TranslateTransform3D OffsetX="0.5"/>
    </GeometryModel3D.Transform>

    </GeometryModel3D>
    </ModelVisual3D.Content>
</ModelVisual3D>

</Viewport3D.Children>
</Viewport3D>

```

Обработчик события Loaded будет выглядеть следующим образом:

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    //Создаем преобразования для первого объекта
    myYRotate = new RotateTransform3D();
    myYAxis = new AxisAngleRotation3D();
    myYAxis.Axis = new Vector3D(0, 1, 0);
    myYAxis.Angle = 0;
    myYRotate.Rotation = myYAxis;

    myZRotate = new RotateTransform3D();
    myZAxis = new AxisAngleRotation3D();
    myZAxis.Axis = new Vector3D(0, 0, 1);
    myZAxis.Angle = 0;
    myZRotate.Rotation = myZAxis;

    myTransform1 = new Transform3DGroup();
    MyModel.Transform = myTransform1;

    myTransform1.Children.Add(myYRotate);
    myTransform1.Children.Add(myZRotate);

    //Создаем преобразования для объекта
    myYRotate2 = new RotateTransform3D();
    myYAxis2 = new AxisAngleRotation3D();
    myYAxis2.Axis = new Vector3D(0, 1, 0);
    myYAxis2.Angle = 0;
    myYRotate2.Rotation = myYAxis2;
    myZRotate2 = new RotateTransform3D();

```

```

myZAxis2 = new AxisAngleRotation3D();
myZAxis2.Axis = new Vector3D(0, 0, 1);
myZAxis2.Angle = 0;
myZRotate2.Rotation = myZAxis2;

myTransform2 = new Transform3DGroup();
MyModel2.Transform = myTransform2;

myTransform2.Children.Add(myYRotate2);
myTransform2.Children.Add(myZRotate2);

// Подготавливаем таймер к работе
MyTimer = new DispatcherTimer();
MyTimer.Tick += new EventHandler(MyTimer_Tick);
MyTimer.Interval = new TimeSpan(100000); }

```

Тик таймера выглядит следующим образом:

```

private void MyTimer_Tick(object sender, EventArgs e)
{
    myYAxis.Angle += 1;
    myZAxis.Angle += 1;

    myYAxis2.Angle -= 2;
    myZAxis2.Angle -= 2;
}

```

Остается добавить две кнопки для запуска и остановки таймера и их обработчики:

```

private void button1_Click(object sender, RoutedEventArgs e)
{
    MyTimer.Start();
}

private void button2_Click(object sender, RoutedEventArgs e)
{
    MyTimer.Stop();
}

```

Трёхмерные преобразования в ThreeJS

Всякий раз, когда перемещаются объекты в трёхмерном пространстве, это делается с помощью математических операций, называемых трансформациями или 3D-преобразованиями. В предыдущих разделах были приведены сведения о переносе через свойство местоположения объекта (**position**) и свойство поворота (**rotation**). Эти свойства наряду с масштабированием (**scale**) составляют три фундаментальных преобразования: перенос (**Translate**), поворот (**Rotate**), масштабирование (**Scale**) – TRS. Рассмотрим 3D-преобразования более подробно.

Каждый объект, который добавляется в сцену с помощью **Scene.add()**, имеет эти свойства, включая полигональные сетки (**Mesh**), источники света (**Light**) и камеры (**Camera**), а материалы и геометрия – не содержат TRS свойств. Приведём примеры, для управления этими свойствами через метод **set()** для камеры, источника света и объекта на сцене:

```
camera.position.set(0, 0, 10);
light.position.set(10, 10, 10);
cube.rotation.set(-0.5, -0.1, 0.8);
```

Таким образом, объекты на сцене, полигональные сетки, источники света и камеры порождены от базового класса **Object3D**, в котором описаны основные свойства: **position**, **rotation**, **scale**. Метод **add()** также определен в **Object3D** и наследуется классом сцены, как и **position**, **rotation** и **scale**. Все остальные производные классы также наследуют метод **add()**, позволяя записывать следующие вызовы метода: **Light.add()**, **Mesh.add()**, **Camera.add()** и т. д. Это означает, что можно добавлять объекты друг к другу, чтобы создать древовидную структуру со сценой наверху. Эта древовидная структура известна как *граф сцены* (рис. 12.2).

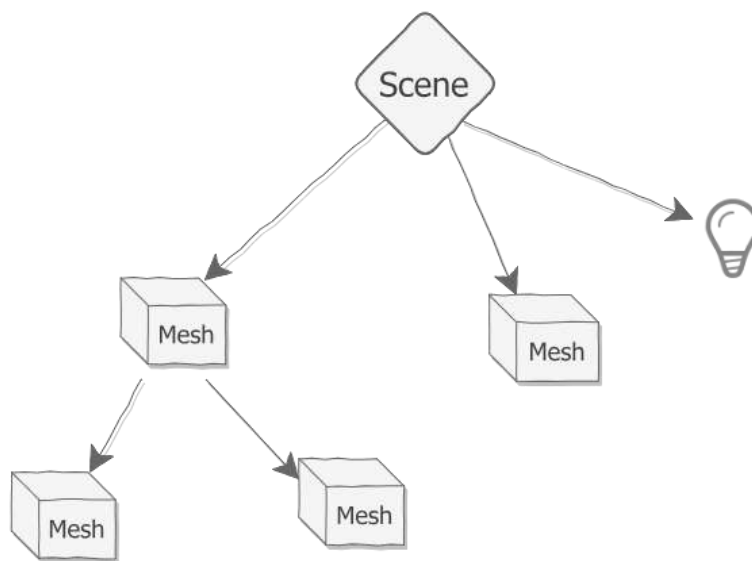


Рис. 12.2. Граф сцены

Когда добавляется один объект к другому, то объект, к которому производится добавление, называется родительским (**parent**), а добавляемый объект – дочерним (**child**): **parent.add(child)**. Каждый объект (кроме сцены верхнего уровня) имеет ровно одного родителя и может иметь любое количество дочерних элементов. Используя методы **add()** и **remove()** каждого объекта, можно создавать граф сцены и манипулировать им.

При вызове средства визуализации, например **renderer.render(scene, camera)**, для отрисовки сцены происходит проход по графу

сцены начиная с корня графа. При этом для каждого объекта графа проверяются свойства **position**, **rotation** и **scale** относительно его родителя, для определения местоположения этого объекта.

Для доступа ко всем дочерним элементам сцены используется массив **children**:

```
//Добавление на сцену Mesh
scene.add(mesh);

//Массив children содержит только что добавленный mesh
scene.children; // -> [mesh]

//Добавление источника света
scene.add(light);

//Массив children содержит mesh и light
scene.children; // -> [mesh, light];

//Теперь можем получить доступ к mesh и light по индексу в массиве
scene.children[0]; // -> mesh
scene.children[1]; // -> light
```

Существуют более сложные способы доступа к конкретному дочернему элементу, например метод **Object3d.getObjectByName**. Однако прямой доступ к массиву **children** полезен, когда имя объекта не известно.

Системы координат в ThreeJS: локальная и глобальная. Трехмерное пространство описывается в трехмерной декартовой системе координат. При использовании Three.js наиболее важными из координатных систем являются глобальная система координат или мировое пространство (**World space**) и локальная система координат (**Local space**).

Глобальная система координат соответствует основной сцене, то есть корню графа сцены. Когда добавляется объект непосредственно на сцену, а затем перемещается, вращается или масштабируется, то его местоположение будет меняться относительно мирового пространства, то есть относительно центра сцены или центра глобальной системы координат.

При создании любого Mesh-объекта или источника света создается новая локальная система координат с Mesh или источником света в ее центре. Эта локальная система координат имеет оси X, Y и Z, как и мировое пространство. Локальную систему координат объекта называют локальным пространством (или иногда пространством объекта).

Когда объект добавляется в сцену с помощью **Scene.add()**, он встраивается в систему координат сцены, в мировое пространство. При переносе объекта он будет перемещаться относительно мирового пространства (или, что то же самое, относительно сцены).

При добавлении объекта к другому объекту, находящемуся глубже в графе сцены, происходит встраивание дочернего объекта в локальное пространство родительского объекта. При переносе дочернего объекта он будет перемещаться относительно системы координат родительского объекта. Системы координат вложены друг в друга, как матрешки (рис. 12.3).

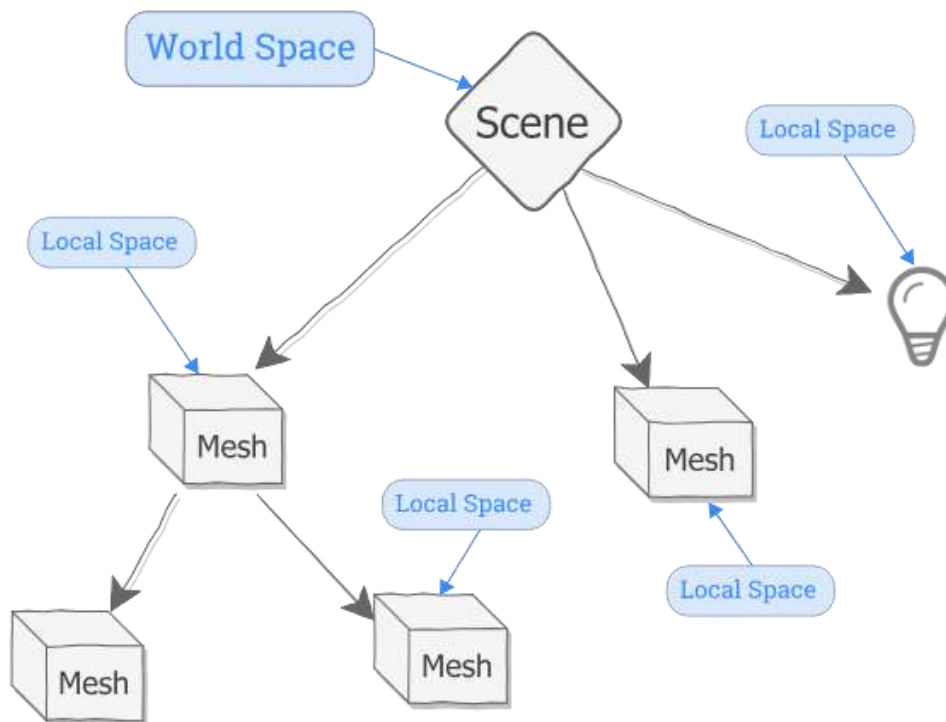


Рис. 12.3. Граф сцены с встроенными локальными системами координат и глобальной системой координат вверху иерархии

Рассмотрим пример фрагмента кода, описывающего сцену с двумя кубами:

```
const geometry = new THREE.BoxGeometry(1, 1, 1);

const green_material = new THREE.MeshBasicMaterial({ color:
0x00ff00 });
const parent_cube = new THREE.Mesh(geometry, green_material);
scene.add(parent_cube);

const red_material = new THREE.MeshBasicMaterial({ color: 0xff0000 });
const child_cube = new THREE.Mesh(geometry, red_material);
child_cube.position.y = 2;
parent_cube.add(child_cube);
function animate() {
    requestAnimationFrame(animate);
    parent_cube.position.x += 0.01;
    renderer.render(scene, camera);
}
animate();
```


Зеленый куб является родительским объектом для красного куба. Красный куб расположен над зеленым. Граф сцены выглядит в этом случае как *Scene*→*parent_cube*→*child_cube*. Родительский куб находится в мировом пространстве, а дочерний куб – в пространстве координат родительского объекта.

В функции **animate()** происходит изменение положения родительского объекта, но поскольку дочерний объект встроен в локальное пространство родительского объекта, то он тоже будет двигаться параллельно родительскому объекту. При попытке добавить управление движением дочернего объекта

```
child_cube.position.x += 0.01;
```

дочерний объект будет двигаться в два раза быстрее, чем родительский, так как сначала происходит пересчет координат родительского объекта и потом в его локальной системе координат рассчитывается увеличение позиции дочернего объекта.

При работе со свойством **position** необходимо понимать, что само свойство описывает точку в пространстве или трёхкомпонентный вектор, основанный на классе **Vector3**, который представляет точку в пространстве в виде трехкомпонентного вектора.

Перенос и масштабирование объектов в ThreeJS. По умолчанию объекты создаются в начале координат. Далее требуется установить его позицию (**position**) через свойства **x**, **y**, **z** либо вызвать метод **set**. Для переноса также можно использовать методы **TranslateX**, **TranslateY**, **TranslateZ**. Данные методы позволяют переносить объект вдоль осей его локальной системы координат (см. пример в разделе «Вращение объектов в ThreeJS»).

Масштабирование объектов в ThreeJS можно производить через свойство **scale** (является также классом) и его метод **set()**. Например, **mesh.scale.set(2, 2, 2)**; увеличит объект в два раза по всем трем осям. Изменение масштаба возможно при обращении к свойствам **x**, **y**, **z** класса **scale**: **mesh.scale.x = 0.5**.

Значения масштаба меньше нуля будут отражать объект, а также уменьшать или увеличивать его. Значение масштаба -1 на любой отдельной оси будет отражать объект, не влияя на размер.

Вращение объектов в ThreeJS. Если преобразование переноса и масштабирования в библиотеке ThreeJS являются достаточно тривиальными, то преобразование поворота стоит рассмотреть подробнее. Вращение требует немного большей осторожности, чем перемещение или масштабирование. Причин этому несколько, но главная из них – **порядок поворотов**. Если при переносе или масштабировании объекта по осям **X**,

Y и Z не имеет значения, какая ось идет первой, то при вращении порядок следования осей имеет значение.

В результате возможностей класса `Vector3`, который использовался как для хранения **position**, так и для свойства **scale**, недостаточно для хранения данных о вращении объекта. Вместо этого в `Three.js` есть не один, а два математических класса для хранения данных о поворотах. Сначала рассмотрим более простой из них: углы Эйлера.

Углы Эйлера представлены в `Three.js` с использованием класса **Euler**. Как и в случае с `position` и `scale`, экземпляр `Euler` автоматически создается и получает значения по умолчанию, когда создается новый объект сцены, точнее полигональная сетка (`mesh`). Для работы с углами Эйлера используется свойство **rotation** (экземпляр класса `Euler`).

По умолчанию `Three.js` будет выполнять вращение вокруг оси X , затем вокруг оси Y и, наконец, вокруг оси Z в локальном пространстве объекта. Можно изменить это, используя свойство **Euler.order**. Порядок по умолчанию называется «XYZ», но также возможны «YZX», «ZXY», «XZY», «YXZ» и «ZYX».

Рассмотрим небольшую модификацию приведенного выше примера:

```
const geometry = new THREE.BoxGeometry(1, 1, 1);

const green_material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const parent_cube = new THREE.Mesh(geometry, green_material);
parent_cube.rotateY(45);
scene.add(parent_cube);

const red_material = new THREE.MeshBasicMaterial({ color: 0xff0000 });
const child_cube = new THREE.Mesh(geometry, red_material);
child_cube.position.y = 2;
parent_cube.add(child_cube);

function animate() {
  requestAnimationFrame(animate);
  child_cube.position.x += 0.01;
  parent_cube.position.x += 0.01;
  renderer.render(scene, camera);
}
animate();
```

В данном примере родительский объект изначально повернут на 45° вокруг оси Y . После этого попытаемся перенести оба объекта вдоль оси X . В том случае зеленый кубик (родительский объект) хоть и повернут, но все-таки будет двигаться вправо вдоль оси X в глобальной системе координат. Красный кубик будет двигаться вдоль оси X локальной системы

координат родительского объекта, то есть вправо и вперед в глобальной системе координат.

Для того чтобы родительский куб двигался в том же направлении, что и дочерний, необходимо заменить оператор `parent_cube.position.x += 0.01;` на `parent_cube.translateX(0.01);`, что позволит и его переносить в его локальной системе координат.

Кроме класса **Euler** для поворотов можно использовать класс **Quaternions**. Для работы с методами этого класса в объекте **Mesh** есть свойство **quaternion**. Можно использовать кватернионы и углы Эйлера как взаимозаменяемые. Когда меняется свойство **mesh.rotation**, свойство **mesh.quaternion** автоматически обновляется, и наоборот. Это означает, что можно использовать углы Эйлера, когда это удобно, и переключаться на кватернионы, когда это удобно.

У углов Эйлера есть несколько недостатков, которые становятся очевидными при создании анимации или математических расчетах, связанных с вращением. В частности, углы Эйлера не всегда можно корректно складывать для создания 3D-анимации. Кватернионы лишены этих недостатков, однако их сложнее использовать, чем углы Эйлера.

У каждого 3D-объекта также есть методы поворота вокруг локальных осей, то есть в объектном (локальном) пространстве **rotateOnAxis (axis : Vector3, angle : Float)** и вокруг глобальных осей **rotateOnWorldAxis (axis : Vector3, angle : Float)**, в мировом пространстве. Ось, вокруг которой происходит поворот объекта, должна задаваться нормализованным (единичным) вектором класса **Vector3**. Угол задается в радианах. Для пересчета из градусов в радианы можно использовать класс **MathUtils** и его метод **degToRad**:

```
THREE.MathUtils.degToRad(90)
```

Для ориентации объекта в пространстве на какую-либо точку (цель) часто применяется метод **lookAt**.

Также в библиотеке ThreeJS существует возможность напрямую обращаться к матрицам преобразования через свойства **matrix** для работы с объектным пространством и **matrixWorld** – для работы с глобальным пространством.

Как итог приведём пример полета бабочки на огонек (красную сферу):

```
import * as THREE from 'three';  
  
// создание сцены и камеры  
const scene = new THREE.Scene();
```

```

const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
                                             window.innerHeight, 0.1,
                                             1000 );

// задание положения камеры в точке (0, 2, 5)
// по направлению к центру координат
camera.position.z = 5;
camera.position.y = 2;

// создание средства отрисовки и добавление его в html-документ
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// создание геометрии квадрата для крыла бабочки
const geometry = new THREE.PlaneGeometry(1, 1);
// создание геометрии сферы для цели
const sphere_geometry = new THREE.SphereGeometry(0.1, 50, 50);

// загрузка текстур из файлов посредством загрузчика
const loader = new THREE.TextureLoader();
const LeftWing_texture = loader.load('LeftWing.png');
const RightWing_texture = loader.load('RightWing.png');
LeftWing_texture.colorSpace = THREE.SRGBColorSpace;
RightWing_texture.colorSpace = THREE.SRGBColorSpace;

// создание красного материала для сферы
const Red_material = new THREE.MeshBasicMaterial({
    color: 0xFF0000,
});

//создание материалов для крыльев из текстур
const LeftWing_material = new THREE.MeshBasicMaterial({
    color: 0xFFFFFFFF,
    map: LeftWing_texture,
    side: THREE.DoubleSide,
});

const RightWing_material = new THREE.MeshBasicMaterial({
    color: 0xFFFFFFFF,
    map: RightWing_texture,
    side: THREE.DoubleSide,
});

// создание левого и правого крыла в центре координат размерами 1 на 1
// крылья смещены влево и вправо на 0,5 единицы от центра
const LeftWing = new THREE.Mesh(geometry, LeftWing_material);
LeftWing.position.x = -0.5;
const RightWing = new THREE.Mesh(geometry, RightWing_material);
RightWing.position.x = 0.5;

// создания тела бабочки в виде пустого объекта (bone)
const body = new THREE.Bone();

```

```

// создание пустых объектов (костей), к которым будут крепиться крылья
const LeftBone = new THREE.Bone();
LeftBone.rotateX(THREE.MathUtils.degToRad(90));
const RightBone = new THREE.Bone();
RightBone.rotateX(THREE.MathUtils.degToRad(90));

// создание графа сцены.
// К корню сцены присоединено тело бабочки
scene.add(body);
// Левая и правая кость являются дочерними объектами тела бабочки
body.add(LeftBone);
body.add(RightBone);
// к левой и правой кости прикреплены левое и правое крыло
LeftBone.add(LeftWing);
RightBone.add(RightWing);

// создание на сцене сферы, на которую летит бабочка
let sphere = new THREE.Mesh(sphere_geometry, Red_material);
sphere.position.z = -10;
scene.add(sphere);

// функция для обработки нажатий кнопок на клавиатуре
// и переноса цели
document.onkeydown = function(e)
{
    console.log(e);
    if (e.keyCode == 37)
    {
        // перенос влево
        sphere.position.x -= 0.1;
    }
    if (e.keyCode == 39)
    {
        // перенос вправо
        sphere.position.x += 0.1;
    }
    if (e.keyCode == 38)
    {
        // перенос вверх
        sphere.position.y += 0.1;
    }
    if (e.keyCode == 40)
    {
        // перенос вниз
        sphere.position.y -= 0.1;
    }
    if (e.keyCode == 188)
    {
        // перенос ближе

```

```

        sphere.position.z -= 0.1;
    }
    if (e.keyCode == 190)
    {
        // перенос дальше
        sphere.position.z += 0.1;
    }
}

// установка начальных значений
let time = Date.now()
let speed = 0.001; // скорость бабочки
let r_speed = 0.005; // скорость поворота крыла
let flag_Up = true; // флаг в какую сторону движется крыло

// установка направления бабочки на цель
let target = new THREE.Vector3();
sphere.getWorldPosition(target)
body.lookAt(target);

// цикл анимации
function animate() {

    // вычисление deltaTime - времени
    // от предыдущего кадра
    const currentTime = Date.now();
    const deltaTime = currentTime - time;
    time = currentTime;

    // установка направления бабочки на цель
    sphere.getWorldPosition(target)
    body.lookAt(target);

    // движение бабочки на цель как перенос тела по z
    // в локальных координатах. Это обеспечивает движение
    // за ним дочерних объектов костей и крыльев
    body.translateZ(speed * deltaTime);

    if (flag_Up) // если надо поднимать крыло
    {
        // поворот костей и за ними поворот крыльев
        LeftBone.rotateY(- r_speed * deltaTime)
        RightBone.rotateY(r_speed * deltaTime)
    }
    else // если надо опускать крылья
    {
        // поворот костей и за ними поворот крыльев
        LeftBone.rotateY(r_speed * deltaTime);
        RightBone.rotateY(- r_speed * deltaTime)
    }
}

```

```

    // смена направления маха крыльев при необходимости
    if ((LeftBone.rotation.y < -0.8) || (LeftBone.rotation.y > 0))
        flag_Up = !flag_Up;
    // запрос на анимацию следующего кадра
    requestAnimationFrame(animate);
    // отрисовка сцены
    renderer.render(scene, camera);
}
// вызов функции анимации
animate();

```

Идея приложения состоит в том, чтобы создать иерархию объектов: сцена → тело бабочки → кость крыла → крыло. При движении тела бабочки вперед вдоль его локальной оси z все его дочерние объекты будут двигаться за ним. Направление движения на цель будет определяться с помощью метода **lookAt**. Махи крыла будут производиться за счет поворота костей, к которым крепятся крылья. Положение костей совпадает с положением тела и таким образом поворот крыла происходит вокруг тела бабочки. Если попытаться производить поворот самого крыла, то получится поворот вокруг центра крыла.

В принципе возможно избавиться от костей крыла и просто вычислять точку поворота на крае крыла через координаты вершин полигональной сетки, образующей крыло.

В данном примере не рассматриваются возможности использования полупрозрачных текстур. Для работы с такими текстурами обратитесь к документации (<https://threejs.org/manual/#en/transparency>).

Работа с гранями в ThreeJS. Сложные сцены в ThreeJS представляют собой полигональные сетки, состоящие из граней. Для каждой грани необходимо описать список вершин и порядок следования вершин в грани, то есть используется способ представления полигональной сетки как задание полигонов с помощью указателей в списке вершин. Для корректной отрисовки лицевых граней, в соответствии с алгоритмом Робертса, важен порядок следования вершин по часовой стрелке. Рассмотрим пример, описывающий две грани, вращающихся вокруг осей x и y :

```

import * as THREE from 'three';

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
                                             window.innerHeight, 0.1, 1000 );

camera.position.z = 2;

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

//создаем материал 1

```



```

const material = new THREE.MeshBasicMaterial({color: 0xff0000});
//создаем материал 2
const material2 = new THREE.MeshBasicMaterial({color: 0x0000ff});

//создаем объект
//описываем список вершин
const my_vertices = [];
my_vertices.push(0.1, 0.1, 0.1); //0 вершина
my_vertices.push(0.5, 0.1, 0.1); //1 вершина
my_vertices.push(0.6, 0.7, 0.1); //2 вершина
my_vertices.push(0.9, 0.8, 0.4); //3 вершина
my_vertices.push(0.5, 0.8, 0.4); //4 вершина
my_vertices.push(0.6, 0.5, 0.4); //5 вершина

//описываем индексы в списке вершин
const my_indices = [];
my_indices.push( 0, 1, 2 ); // грань 1
my_indices.push( 2, 1, 0 ); // обратная сторона грани 1

const my_indices2 = [];
my_indices2.push( 3, 4, 5 ); // грань 2
my_indices2.push( 5, 4, 3 ); // обратная сторона грани 2

// создаём объект 1
const geometry = new THREE.BufferGeometry();
geometry.setIndex( my_indices );
const buffer = new THREE.Float32BufferAttribute( my_vertices, 3 );
geometry.setAttribute( 'position', buffer );

//создаем полигональную сетку для объекта 1 с материалом 1
const face1 = new THREE.Mesh(geometry, material);
//создаем объект 2
const geometry2 = new THREE.BufferGeometry();
geometry2.setIndex( my_indices2 );
const buffer2 = new THREE.Float32BufferAttribute( my_vertices, 3 );
geometry2.setAttribute( 'position', buffer2 );

//создаем полигональную сетку для объекта 2 с материалом 2
const face2 = new THREE.Mesh(geometry2, material2);

//Добавляем полигональную сетку объекта 1 и 2 на сцену
scene.add(face1);
scene.add(face2);

function AnimLoop()
{
    // поворот нулевого дочернего объекта вокруг оси x
    scene.children[0].rotation.x += 0.01;
    // поворот первого дочернего объекта вокруг оси y
    scene.children[1].rotation.y += 0.01;

    renderer.render(scene, camera);
}

```



```

    requestAnimationFrame(AnimLoop);
  }
  AnimLoop();

```

В данном примере создано две парных грани, для отрисовки обратной стороны, хотя этого можно избежать, установив свойство **side** для материала:

```
side: THREE.DoubleSide
```

В рассмотренном примере все вершины граней будут храниться в атрибутах геометрии. В данном случае это **face1.geometry.attributes.position.array**. Нулевой элемент массива будет хранить *x* нулевой вершины, то есть для приведенного примера значение 0,1, первый элемент массива, будет содержать *y* нулевой вершины, второй элемент – *z* нулевой вершины. Третий, четвертый и пятый элементы содержат *x*, *y* и *z* второй вершины и т. д. Поэтому можно менять координаты только одной вершины:

```

function AnimLoop()
{
  // Перенос нулевой вершины по оси X
  face1.geometry.attributes.position.array[0] += 0.01;
  face1.geometry.attributes.position.version++;

  renderer.render(scene, camera);
  requestAnimationFrame(AnimLoop);
}

```

Альтернативой увеличения версии свойства **position** является установка атрибута **needsUpdate** в значение **true**:

```
face1.geometry.attributes.position.needsUpdate = true;
```

Рассмотрим более сложный пример работы с отдельными вершинами:

```

function rotateAroundObjectAxis( object, axis, radians)
{
  // создаем матрицу поворота 4 на 4
  let rotObjectMatrix = new THREE.Matrix4();

  // задаем элементы матрицы для поворота
  // вокруг переданного в параметрах вектора
  rotObjectMatrix.makeRotationAxis(axis.normalize(),
                                   radians);

  // применяем матрицу поворота к объекту
  object.geometry.applyMatrix4(rotObjectMatrix);
}

function AnimLoop()

```

```

{
  // определяем координаты начала отрезка
  const pos = face1.geometry.attributes.position;
  let dx = pos.array[0];
  let dy = pos.array[1];
  let dz = pos.array[2];
  // находим вектор x, y, z, содержащий сторону грани
  let x = pos.array[6] - pos.array[0];
  let y = pos.array[7] - pos.array[1];
  let z = pos.array[8] - pos.array[2];

  // переносим всю грань,
  // чтобы конец грани совпал с началом координат
  face1.geometry.translate(-dx, -dy, -dz);

  // поворачиваем грань вокруг вектора x, y, z
  rotateAroundObjectAxis(face1, new THREE.Vector3(x, y, z),
    0.01);

  // выполняем обратный перенос грани
  face1.geometry.translate(dx, dy, dz);
  renderer.render(scene, camera);
  requestAnimationFrame(AnimLoop);
}

```

Данный пример реализует поворот грани вокруг одной из ее сторон, а именно поворот грани вокруг ребра, соединяющего нулевую и вторую вершину. Для этого производим комбинированные преобразования, а именно:

- 1) переносим всю грань, чтобы нулевая вершина совпала с началом координат;
- 2) производим поворот вокруг вектора, который соответствует ребру между нулевой и второй вершиной. Для нахождения этого вектора используем разницу между вектором второй вершины и нулевой;
- 3) делаем обратный пункту 1 перенос.

Задание по лабораторной работе

Разработайте приложение, анимирующее трехмерную сцену, в соответствии со своим вариантом либо на WPF, либо на ThreeJS.

Варианты заданий

1. Разработайте программу, визуализирующую движение треугольника, прямоугольника, окружности, пирамиды, призмы по разным траекториям с плавно меняющейся скоростью движения. Размеры объектов также меняются.

2. Спроектируйте приложение, отображающее различные объекты (пирамидки, кубики, отдельные грани), плавающие в невесомости. Вращение производится также вокруг глобальной и собственных осей.

3. Реализуйте игру для развития памяти. Перед вами несколько парных карт. На короткое время игра показывает, где какие карты лежат, затем все карты поворачиваются рубашками вверх. Необходимо вспомнить, где расположены одинаковые карты и выбрать их. При нажатии на карту реализуйте поворот карты вокруг собственной оси. При выборе двух одинаковых карт удалите данные карты по трехмерной траектории.

4. Разработайте программу, анимирующую полет бабочки по пространственной кривой Безье.

5. Создайте приложение, анимирующее падение снежинок в 3D-пространстве. Снежинки падают по спирали и вращаются вокруг собственных осей

6. Реализуйте программу, визуализирующую несколько этапов складывания какого-либо оригами.

7. Разработайте приложение, анимирующее падение листочков с дерева в 3D-пространстве.

8. Спроектируйте приложение, отображающее многогранник. При нажатии на кнопку многогранник должен «взорваться», то есть грани должны разлететься в разные стороны, вращаясь вокруг своих осей.

9. Создайте программу визуализации движения планет в солнечной системе по 3D-траекториям. Вокруг хотя бы одной планеты должен вращаться спутник.

10. Разработайте приложение, выводящее 3D-модель книги с переворачивающимися страницами. Пользователь может управлять наклоном книги, скоростью перелистывания страниц и положением источника освещения.

11. Реализуйте программу, имитирующую полет бумеранга в 3D-пространстве.

12. Разработайте 3D-сцену пролета по сложным траекториям двух космических кораблей.

13. Создайте приложение, визуализирующее 3D-модель удара кием по шару в бильярде.

14. Спроектируйте программу, анимирующую бросание игральные костей.

15. Создайте анимированную модель 3D-шкафа. Предусмотрите анимацию открытия дверей и выдвижение ящиков шкафа.

16. Создайте анимированную модель 3D-парусника с косым парусом. Реализуйте возможность движения модели не только в одной плоскости (движение по волнам) и возможность поворота паруса.

17. Разработайте анимированную 3D-модель растения со стеблем и листочками.

18. Разработайте анимированную 3D-модель расцветающего цветка.

19. Создайте анимированную 3D-модель дома. Реализуйте анимацию открывания дверей и окон.

20. Спроектируйте приложение для анимации полета двух самолетов, выполняющих фигуры высшего пилотажа.

Контрольные вопросы

Контрольные вопросы по первой образовательной траектории

1. Какой класс удобно использовать для группы последовательных 3D-преобразований?

2. Что необходимо сделать для применения различных 3D-преобразований к разным элементам 3D сцены?

3. Какие подготовительные действия возможны в обработчике события Loaded?

4. Что нужно включить в описание элемента на XAML, чтобы потом обращаться к этому элементу в WPF-коде?

Контрольные вопросы по второй образовательной траектории

1. Что такое граф сцены?

2. Чем локальная система координат отличается от глобальной?

3. Какие основные способы существуют для трехмерных преобразований?

4. Как описать отдельную грань?

Заключение

Компьютерная графика и геометрия – одни из самых интересных дисциплин, изучаемых студентами IT-направлений. При этом акцент всегда делался не только на изучении программных средств компьютерной графики, а именно различных графических редакторов, но и на моменты, связанные с алгоритмами, математическими основами и особенностью разработки графических приложений. К сожалению, литература по компьютерной графике носит описательный характер и плохо подходит для подготовки и выполнения лабораторных работ по таким дисциплинам, как «Компьютерная графика», «Современные мультимедиа системы и компьютерная графика», «Компьютерная графика». Данный лабораторный практикум составлен с целью оказания помощи студентам при подготовке лабораторных работ по перечисленным дисциплинам.

Несомненным плюсом пособия, с точки зрения автора, является возможность подготовки и выполнения лабораторных работ по двум образовательным трекам. Первый трек подойдёт студентам, изучающим системное программирование и знающим такие языки программирования, как C++, C#. Вторая образовательная траектория больше подойдет студентам, которые изучают web-технологии и frontend-разработку.

В перспективе рассматривается вопрос подготовки лабораторного практикума для выполнения ряда лабораторных работ на основе популярных графических движков (например, Unity, Godot, Unigine) и средств создания 3D-моделей (например, Blender 3D).

Список литературы

1. Демин А.Ю. Теоретические основы компьютерной графики : учебное пособие / А.Ю. Демин. – Томск : Изд-во ТПУ, 2014. – 132 с.
2. Божко А.Н. Компьютерная графика : учебное пособие для вузов / А.Н. Божко, Д.М. Жук, В.Б. Маничев. – Москва : Изд-во МГТУ им. Н.Э. Баумана, 2007. – 392 с.
3. Гринченко В.Т. Введение в нелинейную динамику. Хаос и фракталы / В.Т. Гринченко, В.Т. Мацыпура, А.А. Снарский. – 2-е изд. – Москва : Издательство ЛКИ, 2007. – 264 с.
4. Дегтярев В.М. Компьютерная геометрия и графика / В.М. Дегтярев. – Москва : Академия, 2010. – 192 с.
5. Домасев М. Цвет. Управление цветом, цветовые расчеты и измерения / М. Домасев, С. Гнатюк. – Санкт-Петербург : Питер, 2009. – 224 с.
6. Никулин Е.А. Компьютерная геометрия и алгоритмы машинной графики / Е.А. Никулин. – Санкт-Петербург : БХВ – Петербург, 2003. – 560 с.
7. Роджерс Д. Алгоритмические основы машинной графики / Д. Роджерс ; пер. с англ. – Москва : ЁЁ Медиа, 2024. – 512 с.
8. Роджерс Д. Математические основы машинной графики ; пер. с англ. / Д. Роджерс, Дж. Адамс. – Москва : Мир, 2001. – 604 с.
9. Тихомиров Ю. Программирование трехмерной графики / Ю. Тихомиров. – Санкт-Петербург : БХВ – Петербург, 2000. – 256 с.
10. Боресков А.В. Компьютерная графика : учебник и практикум для среднего профессионального образования / А.В. Боресков, Е.В. Шикин. – Москва : Юрайт, 2024. – 219 с.
11. Мацуда Коичи, Ли Роджер. WebGL. Программирование трехмерной графики / Коичи Мацуда, Роджер Ли. – Москва : ДМК Пресс, 2018. – 494 с.

Учебное издание

ДЕМИН Антон Юрьевич

КОМПЬЮТЕРНАЯ ГРАФИКА И ГЕОМЕТРИЯ

Лабораторный практикум

Учебно-методическое пособие

Научный редактор
доктор технических наук
О.В. Шефер

Корректурa *Е.Л. Тен*
Компьютерная верстка *Д.В. Сотникова*
Дизайн обложки *А.И. Сидоренко*

Подписано к печати 12.11.2024. Формат 60×84/16. Бумага «Снегурочка».
Печать CANON. Усл. печ. л. 9,95. Уч.-изд. л. 8,99.
Заказ 398-24. Тираж 100 экз.



Издательство

ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ