

# **Практикум по компьютерной графике**

*Методическое пособие*

*Версия 1.7*

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Простое приложение с отрисовкой</b>	<b>5</b>
Условие задачи . . . . .	10
<b>2 Растеризация, интерполяция и кривые</b>	<b>13</b>
Алгоритмы отрисовки . . . . .	14
Задача интерполяции . . . . .	24
Барицентрические координаты . . . . .	26
Растеризация треугольника . . . . .	28
Вспомогательный проект . . . . .	30
Варианты заданий . . . . .	31
<b>3 Трёхмерное пространство</b>	<b>34</b>
Структура трёхмерных объектов . . . . .	34
Задача триангуляции . . . . .	37
Расчёт нормалей . . . . .	38
Некоторые важные правила разработки . . . . .	39
О тестах и их важности . . . . .	44
Чтение Obj-формата . . . . .	46
Аффинные преобразования . . . . .	50
Варианты заданий . . . . .	56
<b>4 Софтверный рендер</b>	<b>59</b>
Графический конвейер . . . . .	59
Наложение текстуры . . . . .	67
Простейшая модель освещения . . . . .	68
Алгоритм Z-буфера . . . . .	70
Вспомогательный проект . . . . .	71
Варианты заданий . . . . .	73
<b>5 Быстрее, выше, сильнее!</b>	<b>78</b>
Возможные задачи . . . . .	78
<b>Приложение 1. Работа с Git через консоль</b>	<b>79</b>
Что такое Git? . . . . .	79
Куда вбивать команды? . . . . .	80
С чего начать работу с репозиторием? . . . . .	80
Как делать коммиты? . . . . .	81
Как залить коммиты на сервер? . . . . .	83
Как работать с ветками? . . . . .	84

Как влить себе чужие коммиты? . . . . .	86
Как сохранить свои изменения, но не делать коммит? . . . . .	86
Как откатить коммиты и несохраненные правки? . . . . .	87
Как отменить добавление файлов к коммиту? . . . . .	88
Как откатить изменения до состояния последнего коммита? . . . . .	89
Как откатить изменения до состояния любого коммита? . . . . .	89
Как запретить Git видеть некоторые файлы? . . . . .	90
<b>Приложение 2. Операции над векторами</b>	<b>91</b>
Скалярное произведение . . . . .	91
Векторное произведение . . . . .	93
Вектора-строки и вектора-столбцы . . . . .	94
<b>Приложение 3. Введение в теорию сплайн-функций</b>	<b>96</b>
Интерполяционный многочлен Лагранжа . . . . .	97
Кубический интерполяционный сплайн . . . . .	100
Кривые Безье . . . . .	105
Вспомогательные проекты . . . . .	107
<b>Список дополнительной литературы</b>	<b>110</b>

# Введение

*Компьютерная графика* (КГ) в широком смысле — сфера деятельности человека, связанная с созданием, обработкой и представлением графических изображений с помощью компьютера. За редкими исключениями любой современный человек взаимодействует с КГ каждый день: когда смотрит на экран телефона, играет в игры, ходит в кино, читает книги, рекламу или состав продуктов в магазине, делает фотографии, едет в поезде или за рулём нового автомобиля, и много где ещё.

Кроме того, как только компьютеры начали входить в человеческую жизнь, потребовалось вести общение с ними через удобный интерфейс. Без решения этой проблемы вычислительные машины теперь не могли бы применяться в широком диапазоне задач. На текущий момент без КГ сложно представить сферу развлечений (videogры, анимацию, фильмы), графический и промышленный дизайн, образование, научные исследования, медицину, архитектуру, проектирование и пр.

С практической точки зрения КГ — дисциплина, лежащая на пересечении нескольких наук. Именно поэтому в целях саморазвития она может представлять интерес для широкого круга исследователей. Так, например, разработчик игрового движка для задач симуляции будет вынужден использовать законы физики, классической механики. А продуктовой компании, делающей фоторедактор, чтобы не отстать от конкурентов, придётся заняться машинным обучением.

Даже для предложенных ниже детских задач потребуются как навыки математики, в первую очередь, линейной алгебры и математического анализа, так и разработки: хорошего понимания алгоритмов и структур данных, объектно-ориентированного программирования, грамотного планирования проекта и его тестирования. Подразумевается, что студент, берущийся за задания в рамках курса по КГ, уже всем этим владеет. Но частично необходимая теория всё же будет изложена непосредственно перед задачами.

Читаемый в стенах университета стандартный курс можно считать лишь введением в предмет. Разбираемым на лекциях и в этом пособии алгоритмам уже несколько десятилетий. Некоторые родились еще на заре создания компьютеров, в середине XX века. Тем не менее, без их понимания осмысленное движение дальше просто невозможно.

Данное пособие не стоит рассматривать отдельно от лекционных занятий. Информация здесь будет подана очень сжато и лишь в том объёме, что необходим для выполнения заданий. Предполагается, что студенты внимательно слушают лекции, а затем перечитывают материал с целью освежить знания в памяти.

В рамках семестра будут предложены 4 обязательные задачи и одна по желанию. Первые две будут посвящены двумерной графике: знакомству с инструментами и реализации некоторых известных алгоритмов. Третья задача научит работать в коде с трёхмерными объектами. Четвёртое задание, софтверный рендер, можно считать финальным проектом для всего курса. В его рамках мы попробуем

смоделировать реальную разработку в команде в сжатые сроки. Как показывает опыт, для большинства студентов времени едва хватает, чтобы успеть закончить проект до экзамена.

В задачах КГ нашли своё применение разные языки, однако главным является C++. Обусловлено это, в первую очередь, его гибкостью и острой необходимостью в оптимизации кода. К сожалению, большинство студентов на первых курсах приемлемо владеет только Java и еще не имеет навыка быстрого переключения между инструментами для нужд проекта. Поэтому в рамках пособия мы будем вынуждены опуститься от языка творчества, науки и искусства до языка ремесленников. Именно на Java будет описано большинство примеров.

Исходный код, вспомогательные проекты и прочее для нужд курса можно найти в открытом репозитории по ссылке:

<https://github.com/kv1tr4vn/CGVSU>

Желаем удачи в выполнении работ!

# 1 Простое приложение с отрисовкой

В рамках данного задания познакомимся с тем, как с помощью готовых библиотек можно что-нибудь визуализировать на экране. Использовать для примеров в этой главе будем фреймворк Swing (Java). Он достаточно прост в ознакомлении и к тому же прекрасно подходит для нашего случая.

Разберём нарочито примитивный пример. Заодно повторим некоторые вещи, которые студенты уже должны знать, если доучились до курса по КГ. Код из примеров можно найти в проекте по ссылке:

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task1/SimpleDrawingApp1>

Начнем с того, что создадим файл *Main.java* с методом *main()*, а также пустой класс *DrawingPanel*. Напомним, что для создания экземпляров классов при объектно-ориентированном подходе используются конструкторы. В нашем случае он пока будет пустым, но это не помешает вызвать его внутри метода *main()*:

```
1 class DrawingPanel {  
2     // empty constructor  
3     public DrawingPanel() {  
4     }  
5 }  
6  
7 public class Main {  
8     public static void main(String[] args) {  
9         // constructor call  
10        new DrawingPanel();  
11    }  
12 }
```

Если все настроено правильно, после запуска можно будет увидеть надпись:

Process finished with exit code 0

Метод *main()* после своей работы вернул код 0. Это означает, что приложение отработало без ошибок.

Теперь превратим класс *DrawingPanel* в окошко, которое откроется при вызове конструктора. К счастью, сложную логику для этого писать не придётся. Мы унаследуем класс от библиотечного *JFrame* и вызовем его методы для настройки в конструкторе.

Будущий класс для отрисовки должен выглядеть так:

```
1 import javax.swing.*;
2
3 class DrawingPanel extends JFrame {
4     private final int BACKGROUND_WIDTH = 800;
5     private final int BACKGROUND_HEIGHT = 800;
6
7     public DrawingPanel() {
8         setTitle("Drawing panel");
9         setSize(BACKGROUND_WIDTH, BACKGROUND_HEIGHT);
10        setVisible(true);
11        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
12    }
13}
```

Здесь на 8-й строчке задается надпись вверху окна, метод `setSize()` устанавливает его размер. Одинадцатая строка нужна для того, чтобы завершать процесс при закрытии приложения.

Теперь при запуске откроется будущий холст. Часть задачи решена, осталось научиться на нем рисовать.

Для того, чтобы получить окно приложения, мы воспользовались наследованием. Оно позволяет в классе-потомке обращаться к части функционала класса-родителя. Но теперь требуется, чтобы потомок был не просто точным клоном своего отца, но и при наличии папиных глазок умел что-то делать иначе. Так, мы планируем переопределить метод `paint()`, отвечающий за то, что конкретно рисуется на окне. У `JFrame` он, конечно, останется старым, а вот у нашего класса будет реализовывать совсем другую логику. Такая возможность давать одноименным вещам разные реализации часто называется полиморфизмом.

Переопределение метода может выглядеть так:

```
1 @Override
2     public void paint(Graphics g) {
3         Graphics2D g2d = (Graphics2D) g;
4
5         g2d.setColor(Color.BLACK);
6         g2d.drawOval(200, BACKGROUND_HEIGHT / 2,
7                     BACKGROUND_WIDTH / 10, BACKGROUND_WIDTH / 10);
8         g2d.setColor(Color.GREEN);
9         g2d.fillOval(200, BACKGROUND_HEIGHT / 2,
10                     BACKGROUND_WIDTH / 10, BACKGROUND_WIDTH / 10);
11 }
```

Разберёмся, что тут происходит. Для начала обратите внимание на аннотацию `@Override` на первой строке. Ее использование является правилом хорошего тона и обычно прописано в стиле кода. Несмотря на то, что переопределение сработает и без таких подписей, аннотация не только улучшит читаемость, но и спасет от багов в некоторых ситуациях. Допустите опечатку в названии `paint()` из примера и посмотрите, что произойдет.

А внутри метода описывается отрисовка фигур. Для этого сначала экземпляр класса `Graphics` приводится к типу `Graphics2D`, так как у последнего больше возможностей. А затем для рисования контура эллипса вызывается метод `drawOval()` и для его заполнения — `fillOval()`.

Метод `setColor()` и прочие, настраивающие кисть, распространяют результат своей работы на весь код ниже до тех пор, пока не будет новая смена настроек.

Таким образом мы сообщаем фреймворку, что хотим отрисовать зеленый круг с черной обводкой по заданным координатам. Результат работы программы изображён на рисунке 1.

`Graphics2D` позволяет гораздо больше, чем рисование кружков. Прямоугольники, прямые и кривые линии, дуги, надписи, различные настройки кисти. Об этом много информации в сети. Начать можно, например, с документации:

<https://docs.oracle.com/javase/7/docs/api/java.awt/Graphics2D.html>

Также могут помочь чужие готовые проекты.

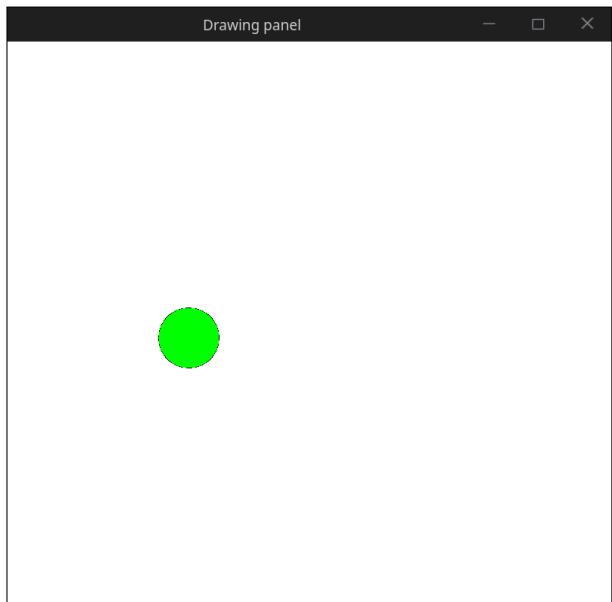


Рис. 1: Окно с отрисовкой

Итак, мы смогли нарисовать фигуру в открытом окошке. Попробуем сделать приложение более интересным и заставим круг двигаться. Код теперь будет выглядеть примерно так:

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5
6 class DrawingPanel extends JFrame implements
    ActionListener {
```

```
7     private final int BACKGROUND_WIDTH = 800;
8     private final int BACKGROUND_HEIGHT = 800;
9     private final int TIMER_DELAY = 1000;
10    private final Timer timer = new Timer(TIMER_DELAY,
11        this);
12    private int ticksFromStart = 0;
13
14    public DrawingPanel() {
15        setTitle("Drawing panel");
16        setSize(BACKGROUND_WIDTH, BACKGROUND_HEIGHT);
17        setVisible(true);
18        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
19        timer.start();
20    }
21
22    @Override
23    public void paint(Graphics g) {
24        Graphics2D g2d = (Graphics2D) g;
25
26        g2d.setColor(Color.WHITE);
27        g2d.fillRect(0, 0, BACKGROUND_WIDTH,
28                     BACKGROUND_HEIGHT);
29
30        g2d.setColor(Color.BLACK);
31        g2d.drawOval(ticksFromStart, BACKGROUND_HEIGHT /
32                     2, BACKGROUND_WIDTH / 10, BACKGROUND_WIDTH / 10)
33        ;
34        g2d.setColor(Color.GREEN);
35        g2d.fillOval(ticksFromStart, BACKGROUND_HEIGHT /
36                     2, BACKGROUND_WIDTH / 10, BACKGROUND_WIDTH / 10)
37        ;
38    }
39
40    @Override
41    public void actionPerformed(ActionEvent e) {
42        if (e.getSource() == timer) {
43            repaint();
44            ++ticksFromStart;
45        }
46    }
47}
```

Обсудим, что здесь написано. Помимо того, что класс уже наследуется от JFrame (6-я строка), также добавляются слова *implements ActionListener*. Здесь речь идет уже не о наследовании, а о реализации интерфейса — класса, который не может иметь своих экземпляров и нужен лишь для того, чтобы определить в коде возможное поведение. С точки зрения теории у интерфейса нет полей, нет реализации методов. В нашем случае он нужен для того, чтобы указать, что DrawingPanel будет иметь метод actionPerformed().

Причем, если в случае с наследованием от JFrame мы могли и не переопределять метод paint(), потому что он есть у родителя, то вот метод интерфейса определить обязаны, ведь у ActionListener никаких реализаций нет.

Сам метод нужен для того, чтобы ловить события (ActionEvent) от таймера, который будет посыпать их раз в секунду. Количество таких событий (тиков) будем записывать в переменную ticksFromStart, а после использовать при перерисовке изображения для обновления координат фигуры.

После запуска приложения, которое теперь пришло в полное соответствие с проектом из репозитория, можно увидеть, как зеленый круг отрисуется, а после начнет медленно двигаться вправо.

Полученная программа очень примитивна и имеет кучу проблем с точки зрения программирования. Сейчас весь код свален вместе, его сложно читать и еще сложнее вести дальнейшую разработку. Перед началом работы над собственной задачей посмотрите также проект:

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task1/SimpleDrawingApp2>

Второе приложение рисует фигурку потяжелее и выносит часть функционала в отдельный класс. Также полотно DrawingPanel теперь является не самим окошком, а виджетом на нем, что позволяет легко додавать кнопочки, меню и другие элементы в будущем. Код все еще далек от идеала. Например, не помешало бы вынести куда-нибудь константы, используемые при рисовании. О подобных вещах вам теперь предстоит задуматься самостоятельно. Здесь проведена первичная работа по улучшению, поэтому на этот проект желательно опираться при выполнении своей задачи.

## Условие задачи

Отталкиваясь от разобранных примеров, создайте приложение с осмысленной картинкой, используя готовую библиотеку. Важно, чтобы это был не абстрактный набор квадратиков, а что-то разумное и цельное. Необходимо включить в приложение большинство примитивов, таких как:

- ◊ Прямые и кривые линии;
- ◊ Прямоугольники, эллипсы (контуры и заполнения);
- ◊ Дуги;
- ◊ Текстовые надписи;
- ◊ др.

Не стоит забывать и про возможность рисовать различными цветами, стилями.

Код тоже должен быть осмысленным, ведь его написание — такой же творческий процесс, как и рисование. Красота внутри приложения повлияет на оценку.

Создайте грамотную архитектуру, не лепите все вызовы примитивов в кучу. Рисунок состоит из нескольких составляющих, которые следует разносить по смысловым блокам, методам. Используйте также условные операторы, операторы цикла, константы и все прочее, что позволит сделать код более читаемым.

Не стесняйтесь создавать отдельные классы для каких-то моделей с вашего холста: машин, домиков, солнышка и пр. Это не только улучшит читаемость, но и позволит легко использовать какую-то сущность несколько раз, с разными настройками.

Желательно, чтобы рисунок был не просто фигурой на белом холсте, но оказался как-то вписан в контекст, фон. Например, если делаете машину, добавьте дорогу, облачка, птиц и т.д.

Возможные варианты задач:

- ◊ Транспортные средства (автомобиль, самолёт, ж/д поезд, речное или морское судно);
- ◊ Пейзажи, натюрморты и прочие жанры живописи. Правда, и тут лучше избегать абстракционизма;
- ◊ Различные устройства, интерьеры, экстерьеры;
- ◊ Герои фильмов, мультфильмов, книг, игр;
- ◊ Любые другие осмысленные цензурные рисунки, удовлетворяющие требованиям законодательства РФ и другим нормативным документам.

Можно ограничиться и статическим изображением. Но с использованием разумной анимации или каким-то интерактивом: взаимодействием с помощью мыши, клавиатуры, оценка будет выше. Некоторые студенты умудряются на базе этого упражнения делать простенькие игры.

Это первое ознакомительное задание в курсе, так что не стоит ради него убиваться. Но, конечно, сложность выбранной вами темы повлияет на оценку. Если сомневаетесь, лучше обсудить свой вариант с преподавателем по практике.

В качестве примера прикрепляем некоторые из множества отличных работ, сдаваемых студентами в первом задании.

На рисунке 2 жабки могут следить глазками за мухой, управляемой курсором.



Рис. 2: Пример реализации первой задачи — жабки

Пример на рисунке 3 — это полноценная анимация движения машин на трассе. Траектория сложная, со множеством поворотов. Результат смотрится плавно и приятно глазу.

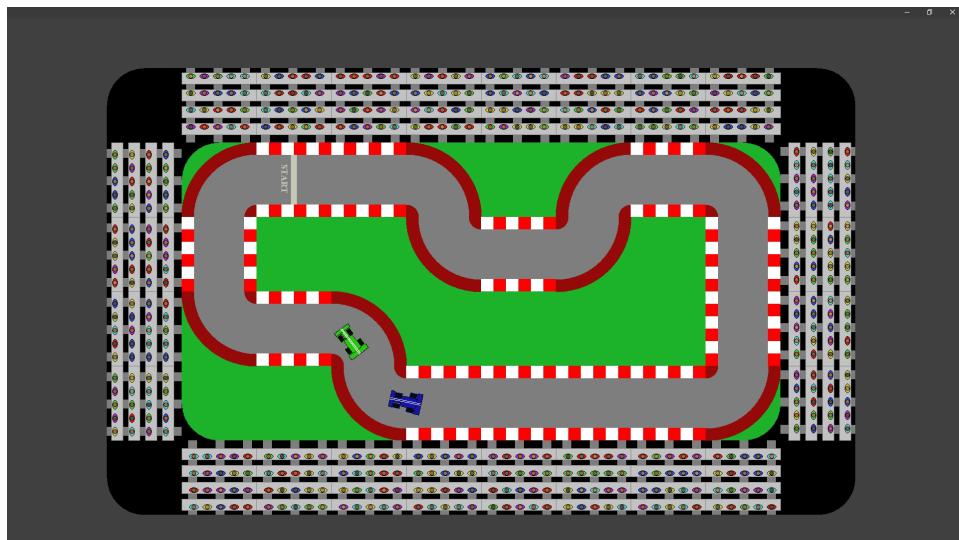


Рис. 3: Пример реализации первой задачи — гонки

А вот на рисунке 4 не динамическая сцена, а набор кадров. Но этот проект уникален тем, что реализует процедурную генерацию. В зависимости от нескольких настроек, можно получать случайные пейзажи.

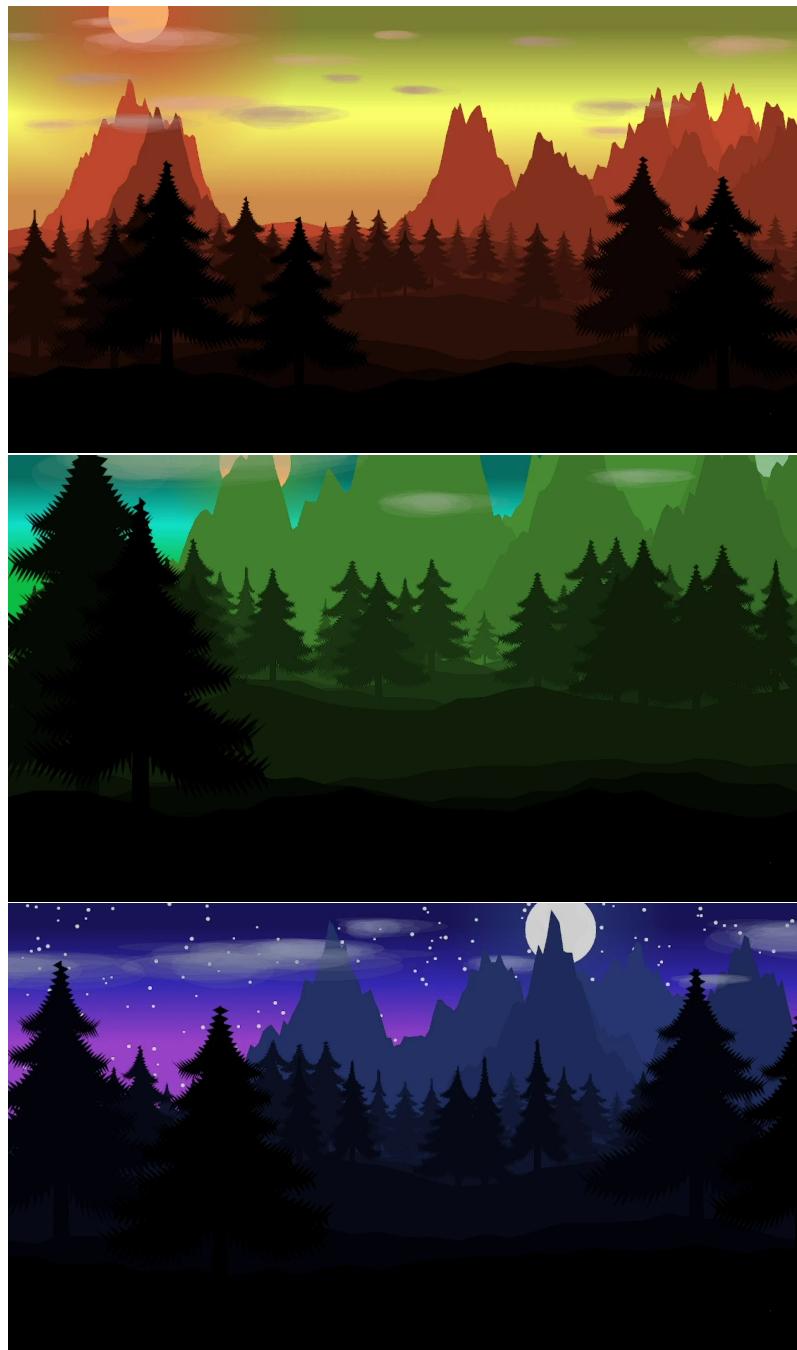


Рис. 4: Пример реализации первой задачи — процедурный лес

## 2 Растеризация, интерполяция и кривые

Второе задание также посвящено двумерной графике. Но теперь вместо того, чтобы вызывать готовые методы фреймворка, рассмотрим и реализуем некоторые базовые алгоритмы.

В этой главе упор будет сделан на отрисовку примитивов: отрезков, эллипсов, треугольников и т.д. Кривые, требующие больших математических выкладок, разобраны в “Приложение 3. Введение в теорию сплайн-функций”.

Вообще, процесс переноса фигуры, заданной набором формул, на матрицу пикселей называется растеризацией. Он схематично изображен на рисунке 5. Если двумерный объект хранится в виде математических формул, то можно говорить, что имеем дело с векторной графикой. Если в виде пикселей — с растровой.

Мы разбираем эти алгоритмы не только для того, чтобы понимать, что стоит за графическими движками, но и для того, чтобы иметь возможность модифицировать их для своих задач. Например, в финальном проекте курса вы не сможете вызывать библиотечный метод для растеризации треугольника, потому что для передачи текстуры модели и освещения необходимо иметь возможность указать свой собственный цвет для каждого из пикселей.

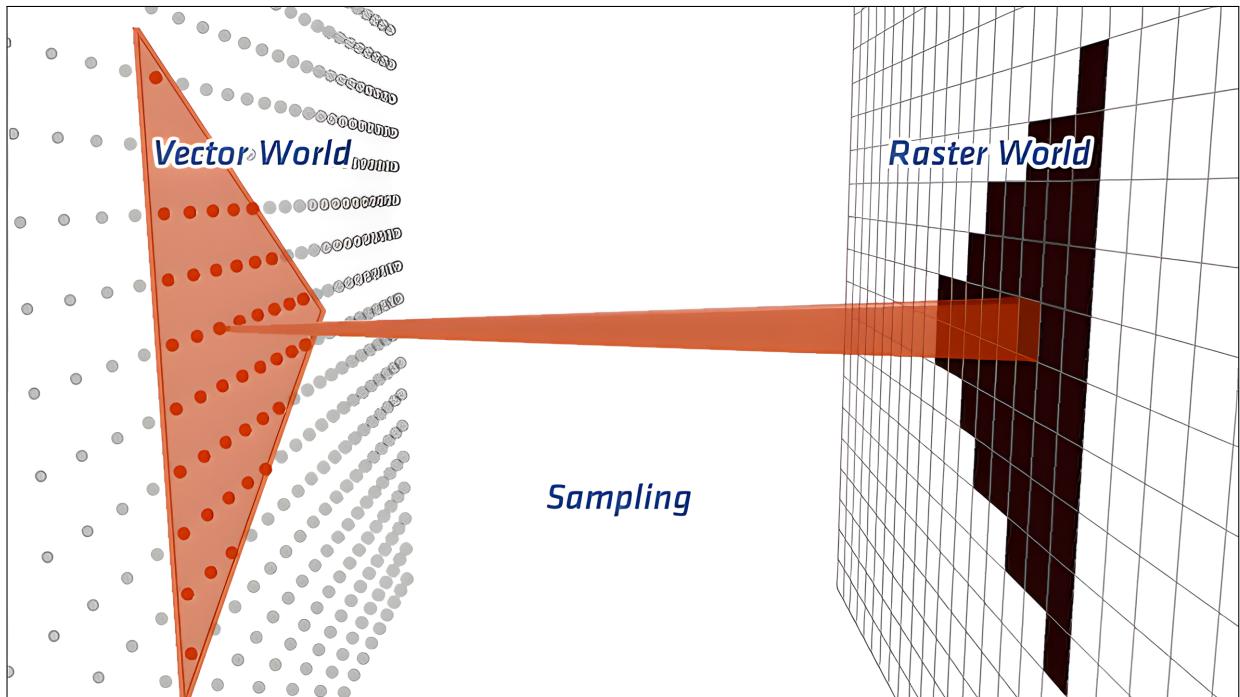


Рис. 5: Задача растеризации

## Алгоритмы отрисовки

В этом параграфе рассмотрим некоторые простейшие алгоритмы. При этом за небольшими исключениями здесь не будет приведён псевдокод для каждого из них.

Во-первых, описываемые задачи настолько хорошо изучены, что при желании можно легко найти даже их готовые реализации. И мы хотим, чтобы вы развивали в себе навыки самостоятельного поиска.

А во-вторых, вместо сухой подачи строк кода эта глава постарается изложить основные идеи, на которых базируются алгоритмы. Понимая их, вы сможете разработать и собственные методы для отрисовки более специфических фигур.

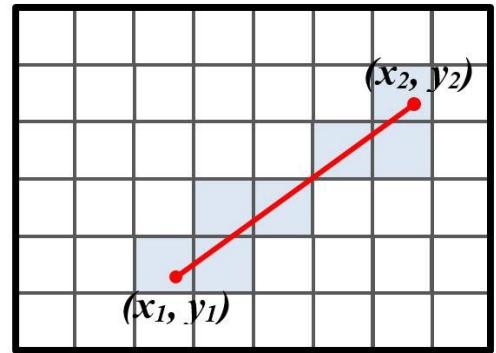


Рис. 6: Растеризация отрезка

### Отрезок прямой: DDA

Начнём с задачи растеризации отрезка (см. рисунок 6). Имеем две точки и линейную функцию между ними. Алгоритмы отрисовки задаются следующим вопросом: как и какие пиксели следует закрасить?

Исторически одним из первых был алгоритм **digital differential analyzer (DDA)**. Он настолько прост, что легко излагается на следующих строчках:

```
1 dx = (x2 - x1); dy = (y2 - y1);
2
3 if (abs(dx) >= abs(dy))
4     step = abs(dx);
5 else
6     step = abs(dy);
7
8 dx = dx / step; dy = dy / step;
9 x = x1; y = y1; i = 0;
10
11 while (i <= step) {
12     put_pixel(round(x), round(y));
13     x = x + dx;
14     y = y + dy;
15     i = i + 1;
16 }
```

Обратите внимание, что переменные  $dx$ ,  $dy$  должны быть вещественными. Они хранят сдвиг вдоль каждой из осей на конкретном шаге. Зависимость линейная, поэтому и коэффициенты получаются в результате деления меньшего на больший. Затем в цикле на каждом шаге делается округление  $x$  и  $y$  до целого с целью закрасить пиксель.

## Отрезок прямой: алгоритм Брезенхема

Описанный выше алгоритм сейчас почти нигде не используется. С практической точки зрения гораздо выгоднее предложенный в 1962-м году алгоритм **Брезенхема**. Чаще всего его описывают следующим образом:

```
1 int deltax = abs(x1 - x0);
2 int deltay = abs(y1 - y0);
3 int y = y0;
4 int diry = y1 - y0;
5
6 real error = 0;
7 real deltaerr = (deltay + 1) / (deltax + 1);
8
9 if (diry > 0) diry = 1;
10 if (diry < 0) diry = -1;
11
12 for x from x0 to x1 {
13     put_pixel(x, y);
14     error = error + deltaerr;
15
16     if (error >= 1.0)
17         y = y + diry;
18     error = error - 1.0;
19 }
```

В отличие от DDA здесь одна из координат (в данном случае  $x$ ) выбирается в качестве главной. В цикле по ней побочная координата  $y$  может как оставаться на той же самой строчке, так и переходить на следующую. За это отвечает аккумулирующая переменная  $error$ . Её значение в соответствии с формулой прямой накапливается на каждом шаге. И когда оно становится больше единицы, то координата  $y$  обновляется, и из  $error$  вычитается один пиксель.

Такой подход позволил сразу работать с целочисленными координатами. Но пока всё еще остаются вещественные переменные для работы с  $error$ . Слегка отредактировав формулу, можно получить вариант алгоритма, написанный исключительно на целых числах:

```

1 int deltax = abs(x1 - x0);
2 int deltay = abs(y1 - y0);
3 int y = y0;
4 int diry = y1 - y0;
5
6 int error = 0;
7 int deltaerr = (deltay + 1);
8
9 if (diry > 0) diry = 1;
10 if (diry < 0) diry = -1;
11
12 for x from x0 to x1 {
13     put_pixel(x, y);
14     error = error + deltaerr;
15
16     if (error >= (deltax + 1))
17         y = y + diry;
18     error = error - (deltax + 1);
19 }

```

Использование целочисленных переменных вместо чисел в плавающей точкой позволяет серьёзно оптимизировать код даже на современных компьютерах, если речь идёт об отрисовке большого количества фигур в доли секунды. Именно поэтому алгоритм Брезенхема на данный момент более предпочтителен.

Но будьте осторожны, реализуя его самостоятельно. У нас, как и в большинстве источников, псевдокод даётся в том виде, в котором он изначально был придуман — для случая, когда первая точка отрезка находится левее второй. Иногда также делается допущение, что она должна быть выше.

Для того, чтобы получить возможность рисовать любую линию, необходимо слегка модифицировать код: сделать дополнительные проверки и либо домножить некоторые переменные на  $-1$ , либо поменять местами точки. Такие правки мы возлагаем на плечи читателя.

## Отрезок прямой: алгоритм Ву

Закончим работу с отрезками упоминанием ещё одного важного для истории графики алгоритма **Ву**. Результат его работы отображен на рисунке 7.

Легко заметить главное отличие от предыдущих подходов. Раньше, когда линия проходила близко к двум пикселям, алгоритму предстояло решать, какой следует закрашивать, а какой — нет. Теперь же алгоритм Ву сразу рисует линию двойной толщины. При этом чем ближе прямая проходит к центру пикселя, тем ярче должен быть его цвет. За счёт этого достигается плавный переход (в англоязычной литературе подобная задача сглаживания носит название *antialiasing*).

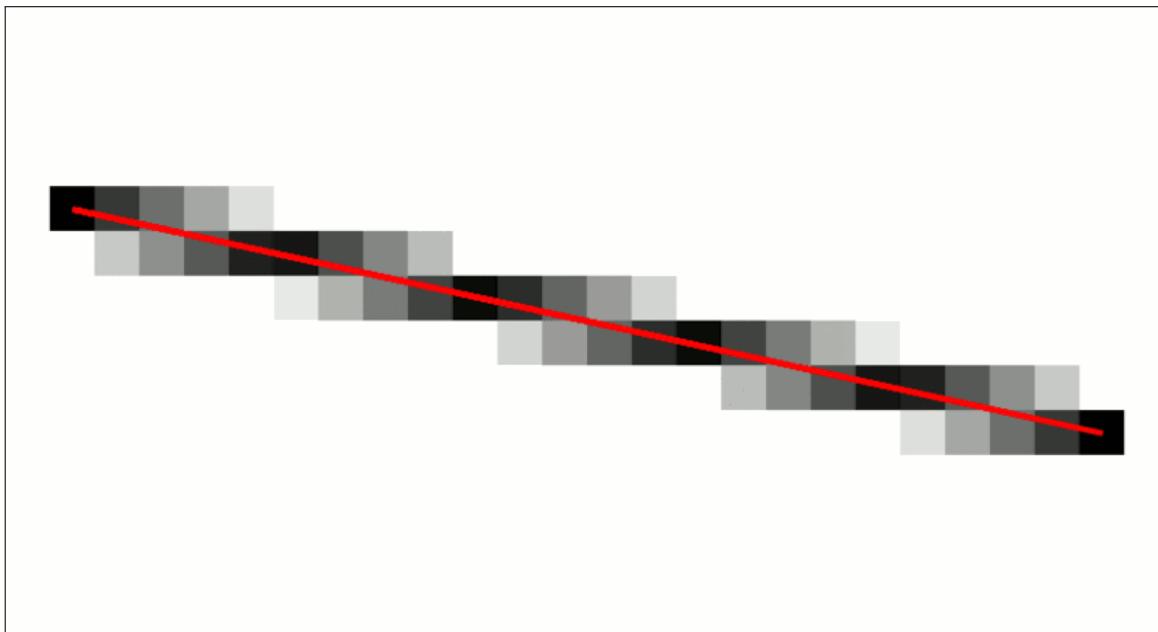


Рис. 7: Результат работы алгоритма Ву

Псевдокод для алгоритма займет прилично места, поэтому будет здесь опущен. Вы можете самостоятельно найти описание деталей алгоритма, оттолкнувшись, например, от:

[https://en.wikipedia.org/wiki/Xiaolin\\_Wu's\\_line\\_algorithm](https://en.wikipedia.org/wiki/Xiaolin_Wu's_line_algorithm)

## Окружность и эллипс

От отрезков прямых перейдем к кривым второго порядка. Работать будем с окружностями и эллипсами. Разумеется, первые являются частным случаем вторых. Однако для окружностей с учётом бесконечного количества осей симметрии есть более оптимальные алгоритмы.

Напомним уравнения окружности и эллипса

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (1)$$

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} = 1, \quad (2)$$

где  $(x_c, y_c)$  - центры фигур,  $r$  - радиус окружности,  $a$  и  $b$  - полуоси, то есть половины ширины и высоты эллипса.

Для отрисовки окружностей наиболее известен **метод средней точки**:

[https://en.wikipedia.org/wiki/Midpoint\\_circle\\_algorithm](https://en.wikipedia.org/wiki/Midpoint_circle_algorithm)

Его прелесть заключается в том, что на самом деле вычисляется только одна восьмая часть фигуры. В силу симметрии, если поменять знаки перед  $x$  и  $y$  (4 комбинации), а также сами координаты  $x$  и  $y$  местами (2 комбинации), выйдет одновременная отрисовка  $4 \cdot 2 = 8$  дуг окружности. Так, на рисунке 8 алгоритм идёт по зелёной дуге, а вместе с ней рисуются и все остальные.

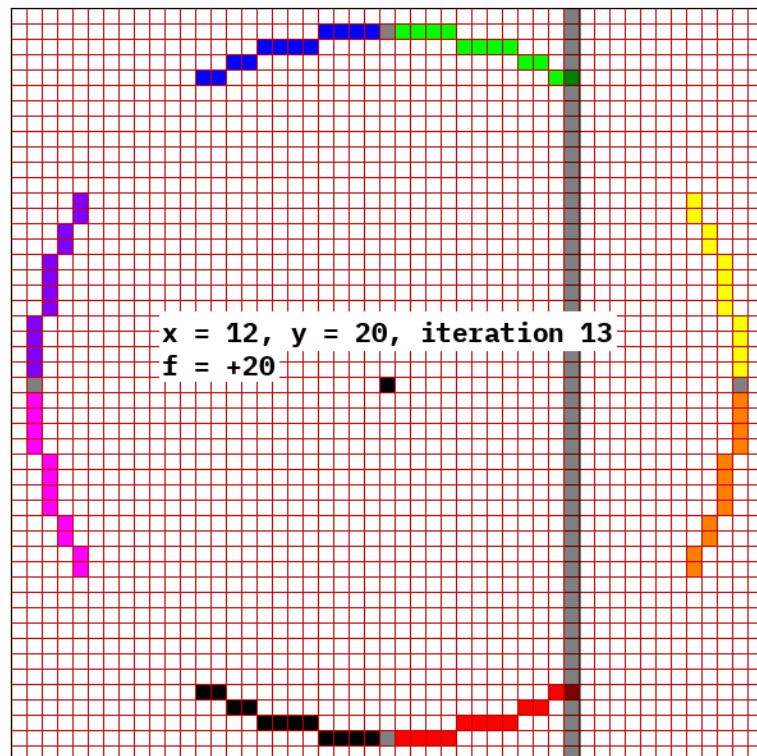


Рис. 8: Процесс работы метода средней точки

К сожалению, у эллипса всего две оси симметрии, поэтому такое разбиение реализовать гораздо сложнее. Но всё еще остается возможность зеркально отобразить одну четвёртую часть фигуры. Например, работа ведётся в первой четверти, а с помощью изменения знаков перед  $x$  и  $y$  рисуются и три другие.

Частный случай метода средней точки — алгоритм Брезенхема. Да, тот самый, который позволил нарисовать отрезок. Существуют его обобщения и для окружности с разбиением на 4 или 8 частей, и для эллипса с разбиением на 4 части, а также для гораздо более сложных фигур.

В качестве демонстрации ниже компактно описан алгоритм для отрисовки окружности с разбиением на 4 четверти. Переменные  $x_c$ ,  $y_c$  в примере отвечают за центр окружности, а  $r$  обозначает её радиус. Можно также заметить, что используется идея Брезенхема с аккумулирующей переменной  $error$ .

```
1 int x = -r, y = 0, error = 2-2*r;
2 do {
3     put_pixel(xc - x, yc + y);
4     put_pixel(xc - y, yc - x);
5     put_pixel(xc + x, yc - y);
6     put_pixel(xc + y, yc + x);
7
8     r = error;
9     if (r <= y) error += ++y*2+1;
10    if (r > x || error > y) error += ++x*2+1;
11 } while (x < 0);
```

Код для случая эллипса базируется на описанном выше, разве что изменяются формулы для перехода на следующую строчку. Но мы не станем приводить его здесь, дабы не отнимать удовольствия самостоятельного поиска информации.

## Заполнение эллипса

Теперь рассмотрим, как можно реализовать заполнение эллипса и окружности. Для начала напишем код для отрисовки одной из самых простых фигур — прямоугольника, чьи оси параллельны границам экрана. Если  $(x_0, y_0)$  — верхняя левая точка, а  $w$  и  $h$  — ширина и высота фигуры, то понадобятся следующие строчки:

```
1 for (x from x_0 to x_0 + w)
2     for (y from y_0 to y_0 + h)
3         put_pixel(x, y);
```

Предположим, что теперь в этот прямоугольник вписан эллипс. В таком случае его полуоси — это половины ширины  $w$  и высоты  $h$ .

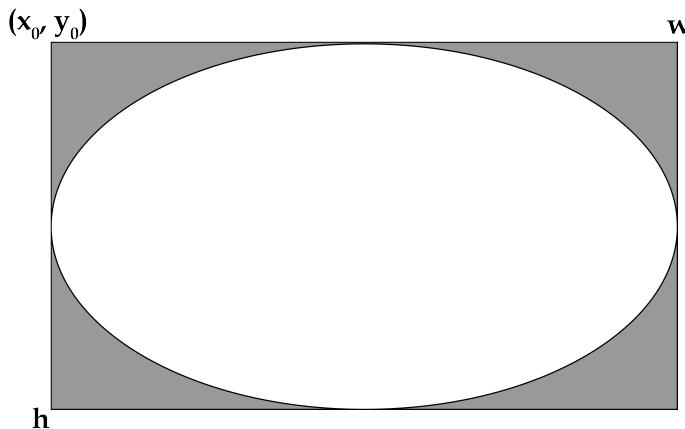


Рис. 9: Эллипс, вписанный в прямоугольник

Для того, чтобы закрасить только его, необходимо во вложенный цикл добавить проверку на то, что точка  $(x, y)$  лежит внутри фигуры. То есть теперь алгоритм всё еще бежит по всем пикселям прямоугольника, но рисует только те, что нам нужны:

```
1 for (x from x_0 to x_0 + w)
2     for (y from y_0 to y_0 + h)
3         if (x, y) inside ellipse:
4             put_pixel(x, y);
```

Опишем эту проверку. Центр эллипса можно легко найти, если добавить к левой верхней точке половину ширины и высоты фигуры:  $x_c = x_0 + \frac{w}{2}$ ,  $y_c = y_0 + \frac{h}{2}$ .

Точка будет лежать внутри (и на границе) эллипса, если выполняется:

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} \leq 1,$$

Аналогично для частного случая окружности формула выглядит следующим образом:

$$(x - x_c)^2 + (y - y_c)^2 \leq r^2,$$

Так мы получили простой алгоритм для заполнения эллипса. Описанная идея часто применяется и для других фигур, особенно очень сложных. Вместо вывода громоздких конструкций можно ограничить объект прямоугольником (часто используется название *bounding box*), а затем взять из него только те точки, что требуются по условию задачи. Единственная сложность здесь — уметь определять, находится пиксель внутри изначальной фигуры или нет.

Хоть предложенный подход и очень прост, у него есть серьезный недостаток — скорость работы алгоритма. Действительно, мы вынуждены бегать по ненужным для отрисовки пикселям, так ещё и вызывать у каждого проверку. Хочется иметь более оптимальное решение. И тут на сцену выходит другой часто применяемый алгоритм — **scanline**. Он также больше идея, чем конкретная реализация, и может существовать для самых разных фигур. Рассмотрим его на примере отрисовки эллипса.

Обычно в качестве главной оси выбирается  $y$ . Но теперь, двигаясь вдоль него, будем перебирать не все возможные  $x$ , а только те, что лежат внутри эллипса. Для этого подставим  $y$  на текущем шаге главного цикла в уравнение (2). Тогда становится легко получить две точки, лежащие на пересечении эллипса и данной строки:

$$x_{left}, x_{right} = x_c \pm \frac{a}{b} \sqrt{b^2 - (y - y_c)^2}$$

Перемещаясь от  $x_{left}$  к  $x_{right}$  теперь возможно закрасить только те пиксели, что лежат внутри фигуры:

```

1 for (y from y_0 to y_0 + h)
2     calculate x_left, x_right with y
3         for (x from x_left to x_right)
4             put_pixel(x, y);

```

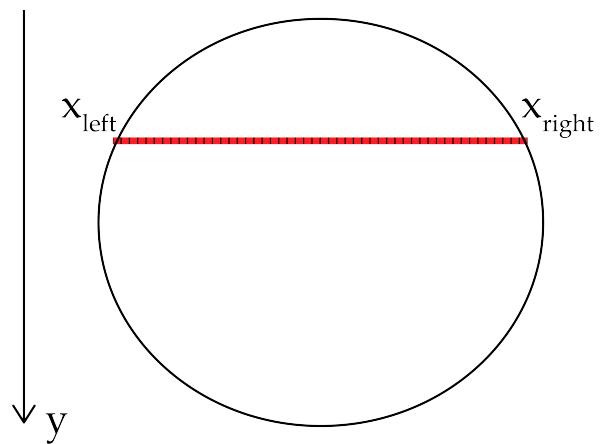


Рис. 10: Заполнение строки с помощью алгоритма *scanline*

## Дуга и сектор

Умев рисовать окружность и круг, перейдём к задачам растеризации дуги и сектора. В этом блоке не будут приведены конкретные алгоритмы. Вместо этого зададимся следующим вопросом: как, имея окружность и выходящие из её центра два вектора, определить, принадлежит точка сектору или нет?

Предположим, имеем окружность и центральный угол  $\angle AOB$ . Необходим механизм, который позволит показать, что точка  $P$  лежит внутри него.

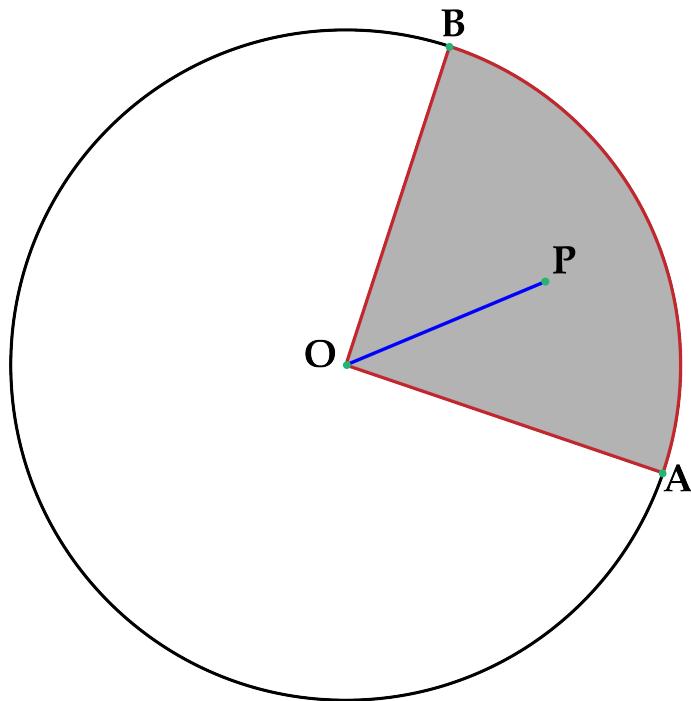


Рис. 11: Точка внутри сектора окружности

Для решения этой задачи неожиданно пригодится операция векторного произведения. Советуем заглянуть в “Приложение 2. Операции над векторами”.

Запишем два выражения:

$$\begin{aligned}\vec{M} &= \vec{OA} \times \vec{OP} \\ \vec{N} &= \vec{OP} \times \vec{OB}\end{aligned}$$

Множители выбираются не случайно, а в прямом порядке, если двигаться против часовой стрелки. Результирующие  $\vec{M}$  и  $\vec{N}$  в силу свойств векторного произведения будут направлены в одну сторону: в сторону читателя, перпендикулярно плоскости рисунка 11. То есть, если ввести координату  $z$ , раз уж мы совершаем переход к трёхмерному пространству, то  $M_z$  и  $N_z$  будут одного знака.

А теперь попробуйте мысленно или на бумаге вынести точку  $P$  за границы сектора. Вы увидите, что в одной из операций произведения множители начнут располагаться по часовой стрелке, а в другой останутся против. Это значит, что знаки у  $M_z$  и  $N_z$  начнут отличаться.

Таким образом, если  $M_z$  и  $N_z$  одного знака, то  $P$  лежит между векторами  $\vec{OA}$  и  $\vec{OB}$ . В противном случае — нет. Используя формулу (9), можно получить компактное выражение для решения задачи. Оставляем это на плечи читателя.

Итак, теперь, имея подобный инструмент, вы можете самостоятельно составить алгоритм для отрисовки дуги и сектора. Для этого следует взять за основу методы для растеризации окружности и круга соответственно, установить точки  $A$  и  $B$  (это можно сделать и отталкиваясь от углов). Проверяя каждый пиксель с помощью описанного выше условия, можно получить необходимый метод.

Есть и другие способы растеризации дуги и сектора. Предложенное решение, конечно, не будет оптимальным из-за проверки большого количества лишних пикселей. Аккуратно разбирая отрисовку круга, вы, например, можете попробовать перейти к идее scanline.

## Задача интерполяции

Говоря простым языком, *интерполяция* — это нахождение промежуточных значений. Задача эта очень общая и выходит далеко за рамки компьютерной графики. Она возникает, если у вас есть дискретный набор чисел и нужно по нему восстановить непрерывную функцию или конкретные точки внутри отрезка. Например, вы измерили высоты земельного участка с определённым шагом, а программа по ним строит гладкую функцию ландшафта. Некоторые примеры решения этой задачи приведены в “Приложение 3. Введение в теорию сплайн-функций”.

В наших заданиях интерес, в первую очередь, представляет интерполяция цвета, ведь это ключевая характеристика пикселей — то единственное, что мы можем задавать на экране. В конце этой главы вы попробуете реализовать механизм, который позволит задавать цвет лишь в определённых точках, а между ними совершать плавный переход. Тоже самое придётся делать и в финальном проекте курса, где при рендеринге трёхмерного объекта через передачу цветов будет интерполироваться текстура, освещение и т.д.

Пока нам будет достаточно *линейной интерполяции* — частного случая, в котором предполагается что искомая зависимость между двумя точками — линейная функция. Решим эту задачу для случая интерполяции цвета вдоль отрезка.

Пусть имеем отрезок с началом в точке  $(x_0, y_0)$  и концом в  $(x_1, y_1)$  (см. рисунок 12). Интерполировать будем некую функцию  $c = c(x, y)$ . В первой точке она равна значению  $c_0$ , а во второй —  $c_1$ . Мы должны восстановить её значение в произвольной  $(x, y)$ , то есть восстановить саму функцию  $c$ .

В качестве  $c$  здесь может выступать что угодно. Например, если имеем дело с трёхканальной картинкой — такой, где каждый пиксель описывается набором из трёх цветов  $(r, g, b)$ , то интерполировать эти значения можно будет, применяя выведенные ниже формулы для каждого из каналов (или для всех сразу, если реализованы простейшие операции над векторами).

В нашем случае предполагается, что зависимость  $c(x, y)$  — линейная функция. Это значит, что изменение функции  $(c - c_0)$  будет относиться к пройденной части отрезка также, как и все  $(c_1 - c_0)$  ко всей его длине:

$$\frac{c - c_0}{\text{dist}((x_0, y_0), (x, y))} = \frac{c_1 - c_0}{\text{dist}((x_0, y_0), (x_1, y_1))}$$

Вспомним, как считается расстояние между двумя точками, а затем выразим

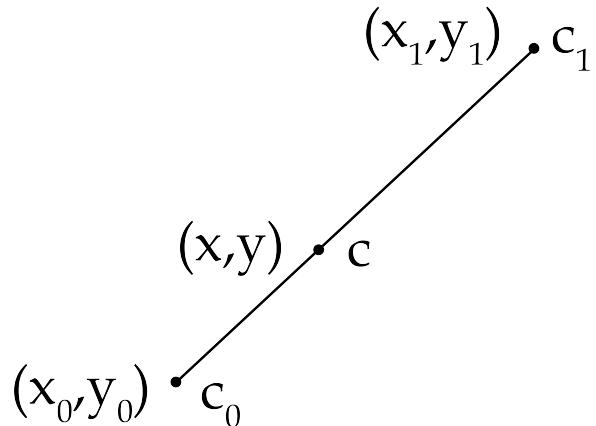


Рис. 12: Интерполяция вдоль отрезка

переменную  $c$ :

$$dist((x_0, y_0), (x_1, y_1)) = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

$$c(x, y) = c_0 + \frac{\sqrt{(x - x_0)^2 + (y - y_0)^2}}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}(c_1 - c_0)$$

Можно увидеть, что  $c$  действительно зависит от  $x$  и  $y$  линейно. Более того, для задачи интерполяции необходимо, чтобы искомая функция проходила через известные точки. Если подставить  $(x_0, y_0)$  и  $(x_1, y_1)$  в формулу, можно получить значения  $c_0$  и  $c_1$  соответственно. Это ещё раз подтверждает, что задача решена корректно.

В заданиях ниже можно встретить требования интерполяции цвета не только для отрезка, но и для эллипса, дуги окружности и пр. Формулы для этих случаев можно вывести самостоятельно, опираясь на приведённое решение. Главное — помнить, что при линейной зависимости отношение приращения функции к приращению аргумента пропорционально.

## Барицентрические координаты

Перед тем, как перейти к растеризации треугольника, необходимо ввести ещё один термин.

Барицентрические координаты — инструмент очень общий и определен для сколь угодно высоких размерностей. Однако нам на практике потребуется только частный случай с работой на плоскости.

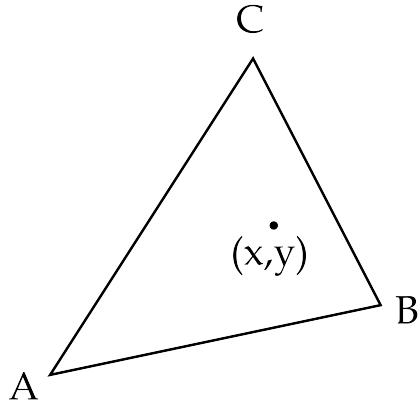


Рис. 13: Точка внутри треугольника

Предположим, имеются три точки  $A$ ,  $B$  и  $C$ . Часто представляют их как вершины треугольника. Тогда произвольную  $(x, y)$  можно выразить следующим образом:

$$(x, y) = \alpha A + \beta B + \gamma C, \quad (3)$$

где  $\alpha$ ,  $\beta$  и  $\gamma$  — так называемые барицентрические координаты.

Уравнение (3) также можно записать в виде системы:

$$\begin{cases} x = \alpha x_A + \beta x_B + \gamma x_C \\ y = \alpha y_A + \beta y_B + \gamma y_C \end{cases} \quad (4)$$

Может показаться абсурдным, что для задания точки на плоскости требуется не 2, а 3 координаты. Кроме того, даже если известны  $A$ ,  $B$ ,  $C$  и  $(x, y)$ , однозначно определить  $\alpha$ ,  $\beta$  и  $\gamma$  из системы (4) не представляется возможным. Обе эти проблемы решаются следующим дополнительным условием:

$$\alpha + \beta + \gamma = 1 \quad (5)$$

Теперь в момент растеризации треугольника (обсудим ниже) можно находить барицентрические координаты для каждого пикселя. В момент его отрисовки мы, очевидно, знаем  $(x, y)$ . Также по условию должны быть известны  $A$ ,  $B$  и  $C$ . Таким образом, из системы (4, 5) однозначно определяются барицентрические координаты. Осталось понять, как их можно применить.

Можно думать о барицентрической системе координат как об ещё одном виде линейной интерполяции. Если известны координаты вершин треугольника, то  $\alpha$ ,  $\beta$  и  $\gamma$  позволяют интерполировать их внутри фигуры. Например, в одном из крайних случаев при  $\alpha = 1$ , а  $\beta = \gamma = 0$ , получаем, что  $(x, y) = A$ . Возможно также применять барицентрические координаты и вне треугольника. В этом случае среди них появятся отрицательные значения. И с помощью этого, кстати, можно проверять, лежит точка внутри фигуры или нет.

Но интерполировать можно не только координаты, но и любые функции. В разделе выше мы обсудили, как вывести формулу для произвольной  $c$ , а после применить её для каждого из каналов  $r$ ,  $g$ ,  $b$ . В данном случае тоже, имея  $\alpha$ ,  $\beta$ ,  $\gamma$  и заданные значения цвета в точках, можно получить набор выражений:

$$\begin{cases} r = \alpha r_A + \beta r_B + \gamma r_C \\ g = \alpha g_A + \beta g_B + \gamma g_C \\ b = \alpha b_A + \beta b_B + \gamma b_C \end{cases} \quad (6)$$

Например, если в вершинах заданы красный, зелёный и синий цвета, можно получить следующую картинку:

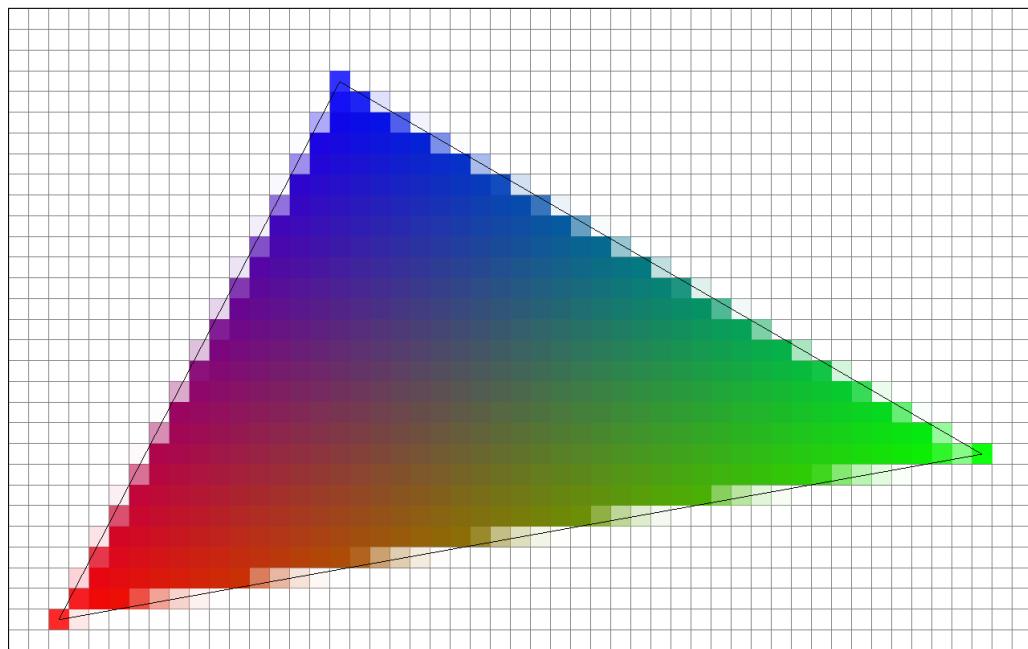


Рис. 14: Интерполяция цвета между вершинами треугольника

С помощью барицентрических координат можно интерполировать между вершинами далеко не только цвет, но и любые другие функции. То, что они инвариантны к аффинным преобразованиям, то есть то, что они корректно отображают пропорции внутри треугольника несмотря на некоторые простые его искажения — крайне выгодное свойство, которое мы будем использовать дальше по курсу.

## Растеризация треугольника

Разберём, наконец, как отрисовать треугольник. Эта задача особенно важна, поскольку без неё в финальном проекте невозможно будет визуализировать трёхмерную модель.

Подход к заливке треугольника будет примерно тот же, что мы разбирали выше для случая эллипса. Во-первых, всегда можно ограничить фигуру прямоугольником, а после, пробегаясь по нему, закрасить только те пиксели, что лежат внутри изначальной фигуры:

```
1 for (x from x_0 to x_0 + w)
2     for (y from y_0 to y_0 + h)
3         if (x, y) inside triangle:
4             put_pixel(x, y);
```

Для определения принадлежности треугольнику могут помочь барицентрические координаты. Или можно взять две пары векторов, образованных сторонами фигуры и проверить, что точка лежит между ними (решение через векторное произведение обсуждалось выше).

Но, разумеется, такая заливка не будет оптимальной. Поскольку в финальном задании вам предстоит рисовать в доли секунды несколько тысяч треугольников, очень важно реализовать наиболее быстрый вариант. Как и в случае с эллипсом желательно перейти к алгоритму **scanline**. Для лучшего восприятия прикреплен рисунок 15.

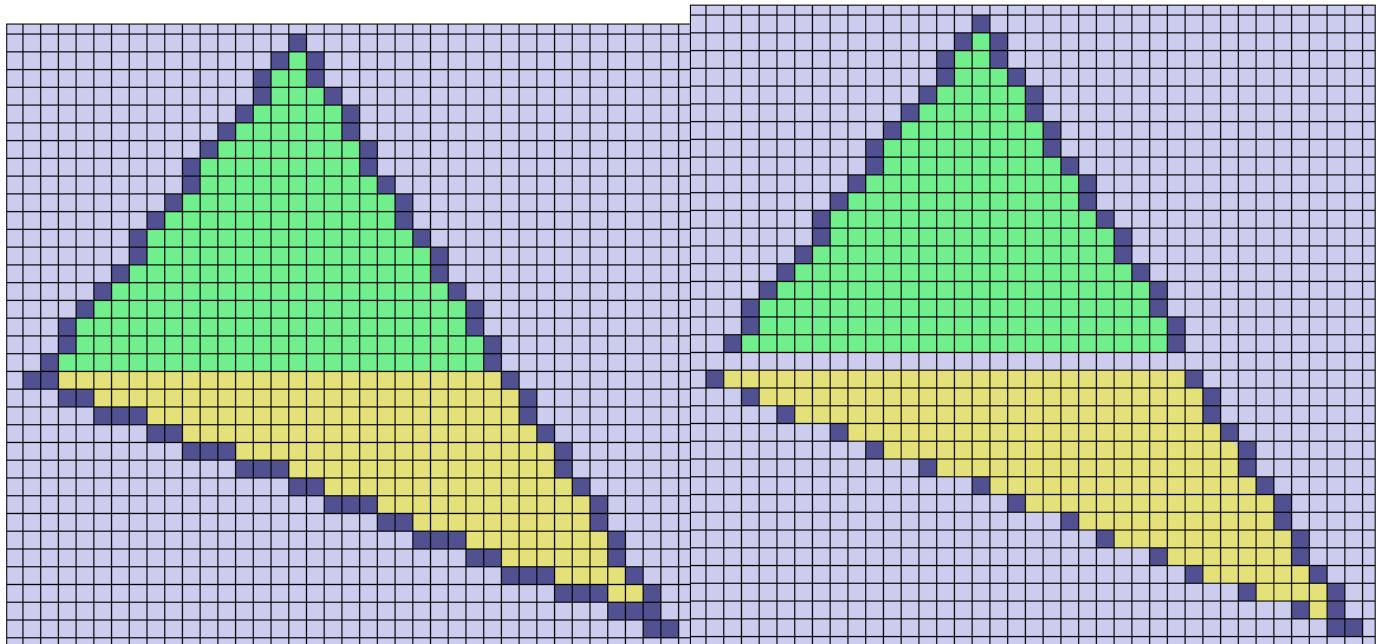


Рис. 15: Растеризация треугольника алгоритмом scanline

Эллипс описывался одним выражением, поэтому можно было вызывать алгоритм для всей фигуры целиком. Треугольник же состоит из трёх уравнений прямых, которые вы можете легко найти, зная вершины. Поэтому для перехода к scanline сперва нужно найти центральную по координате  $y$  точку и разделить горизонтальной линией треугольник на два других. Тогда для каждого из них слева и справа будут соответствующие уравнения прямых. Спускаясь по  $y$ , можно будет выражать из них  $x_{left}$  и  $x_{right}$ , а затем закрашивать всю строку между ними.

Схематично можно описать алгоритм следующим образом:

```
1 sort triangle vertices by y:  
2 (x_0, y_0) - top vertex  
3 (x_1, y_1) - middle vertex  
4 (x_2, y_2) - bottom vertex  
5  
6 calculate line equations  
7  
8 for (y from y_0 to y_1)  
9     calculate x_left, x_right with y from line equations  
10    for (x from x_left to x_right)  
11        put_pixel(x, y);  
12  
13 for (y from y_1 to y_2)  
14     calculate x_left, x_right with y from line equations  
15     for (x from x_left to x_right)  
16        put_pixel(x, y);
```

В теории всё просто, но на практике обычно реализация без багов получается далеко не сразу. Бывает, что написанный метод не работает для некоторых частных случаев треугольников (прямоугольный, со сторонами, которые параллельны осям и т.д.). Или иногда из-за округлений не закрашивается то место, где фигура делится на две части. Поэтому очень важно тщательно протестировать код.

## Вспомогательный проект

В заданиях на отрисовку может пригодиться следующий проект:

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task2/RasterizationFXApp>

Мы советуем оттолкнуться от его кода, чтобы не тратить время на освоение фреймворка JavaFX. Почти во всех задачах ниже не требуется качественный интерфейс. Нужно только грамотно реализовать алгоритм.

На всякий случай проговорим общую структуру оконного приложения. В файле *mainwindow.fxml* описаны виджеты, которые будут добавлены при запуске программы. В классе *RasterizationApplication* этот файл парсится, а затем настраивается внешний вид окошка. *RasterizationController* - это насадка на основной класс приложения, которая прописывает как именно виджетам необходимо себя вести. В данном случае мы добавляем при запуске программы вызов методов, которые будут рисовать на виджете canvas.

Эти методы статические и хранятся в классе *Rasterization*. В качестве примера взята растеризация прямоугольника с осями, параллельными сторонам экрана. Ещё раз обращаем внимание, что в задачах ниже на отрисовку примитивов (первый блок) вы можете использовать только библиотечный метод для установки значения конкретного пикселя:

```
1 pixelWriter.setColor(x, y, color);
```

Сейчас при запуске приложения метод вызывается дважды, в результате чего рисуются два прямоугольника. Вы можете рядом реализовать свои методы и точно также добавить их вызовы в *RasterizationController*.

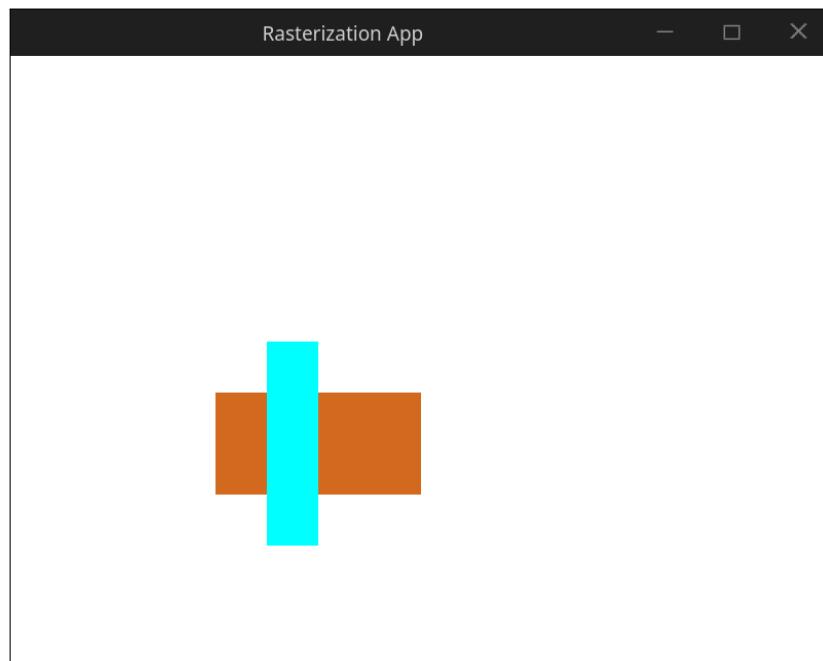


Рис. 16: Проект RasterizationFXApp

## Варианты заданий

**Важно:** начиная с этой задачи и дальше по курсу обязательным условием для выполнения будет использование Git, причём нужно уметь работать с ним через консольную оболочку. Помочь в этом сможет “Приложение 1. Работа с Git через консоль”.

Репозиторий рекомендуется делать приватным. В отдельных случаях преподаватель может принимать задачи через него или попросить при сдаче продемонстрировать работу с Git.

### Первый блок: алгоритмы растеризации.

В этом разделе представлены варианты, связанные с отрисовкой примитивов. В отличие от первой задачи курса, вы не можете использовать здесь готовые библиотечные методы. Единственная доступная опция — заливка цветом отдельного пикселя по его координатам.

Для ускорения работы рекомендуется воспользоваться вспомогательными проектами из репозитория.

Задачи отсортированы по возрастанию сложности. Первые три вообще следует брать только тем, кто отстаёт: тем, кто вынужден подтягивать базовые навыки программирования в данный момент.

Последняя задача с отрисовкой треугольника сложнее, чем кажется на первый взгляд. Её очень важно взять кому-то в группе и желательно взять сильному студенту. Код, который он напишет, будет использоваться остальными в следующих заданиях для растеризации полигонов.

При тестировании обязательно постройте все возможные варианты своей фигуры. Переставляйте координаты местами, сжимайте и растягивайте. Поворачивайте её, если это возможно. Очень частая ситуация, когда программа работает верно при одних входных параметрах, но ломается при других.

Для этих задач не нужен пользовательский интерфейс. Достаточно того, чтобы при запуске программы на экране по прописанным в коде точкам строились фигуры.

1. Реализовать метод (класс) для рисования отрезка алгоритмом DDA. Добавить возможность интерполяции цвета вдоль прямой: от одного конца отрезка до другого. На вход методу поступают координаты концов, цвет в этих точках и т.д.
2. Тоже самое, но с алгоритмом Брезенхэма.
3. Тоже самое, но с алгоритмом Ву.

4. Реализовать метод (класс) для рисования границ эллипса. На вход могут поступать координаты левого верхнего угла вместе с шириной и высотой фигуры. Или центр эллипса и размеры полуосей а и b. Возможны и другие варианты.
5. Реализовать метод (класс) для заполнения эллипса. Добавить возможность интерполяции цвета, например, при удалении от центра фигуры. На вход могут поступать координаты левого верхнего угла вместе с шириной и высотой фигуры. Или центр эллипса и размеры полуосей а и b. Возможны и другие варианты.
6. Реализовать метод (класс) для рисования дуги окружности. Добавить возможность интерполяции цвета от одного конца кривой до другого.
7. Реализовать метод (класс) для заполнения сектора окружности. Добавить возможность интерполяции цвета, например, при удалении от центра круга.
- 8\*. Реализовать метод (класс) для заполнения треугольника. Добавить возможность интерполяции цвета с использованием барицентрических координат. На вход методу поступают координаты вершин треугольника, цвет в этих точках и т.д.

## **Второй блок: кривые и сплайны.**

В выполнении этих задач поможет “Приложение 3. Введение в теорию сплайн-функций” и описанные внутри вспомогательные проекты. Программа должна быть холстом, на котором пользователь сможет с помощью кликов мыши создавать точки. По ним в реальном времени должна строиться кривая. В коде она является ломаной линией, но отступ между точками берётся таким маленьким, что человек не замечает разбиения. Масштабирование, отрисовка координатных осей не требуются.

Важно, что строящаяся кривая не обязана быть функцией, то есть одному  $x$  в ней могут соответствовать несколько  $y$ . Для того, чтобы это было возможно, в некоторых вариантах придётся применить трюк, описанный во вспомогательном проекте, где заводятся две кривые: для осей  $x$  и  $y$ .

При желании в конце можно добавить возможность перетаскивать точки так, чтобы положение кривой обновлялось в реальном времени.

9. Реализовать интерполяционный полином Лагранжа.
10. Реализовать кубический сплайн.
11. Реализовать кривые Безье.

## Третий блок: произвольные задачи.

В этом блоке несвязанные напрямую с лекциями задания на двумерную графику. За желание взять что-то масштабное придётся расплачиваться повышенной сложностью и большим количеством правок.

12. Реализовать программу для построения графиков произвольных функций  $f(x)$ . Пользователь вбивает функцию в текстовое поле. Ваш собственный интерпретатор разбирает строку, которая может содержать все основные операции, разные форматы чисел, элементарные функции, скобки. По полученному в коде выражению строится график. Этот график можно ещё и масштабировать, перемещать с помощью мышки.
13. Программа позволяет строить цветные фракталы. Для того, чтобы архитектура была универсальной, нужно реализовать минимум два множества: например, Мандельброта и треугольник Серпинского. Пользователь может переключаться между ними, перетаскивать картинку с помощью мышки и до бесконечности масштабировать на колёсико.
14. Написать конвертер произвольной цветной картинки в таблицу ASCII символов. Приложение может быть консольным. На вход поступает картинка, параметры. На выходе то же изображение, но составленное из символов. В качестве параметров следует передавать уровень сжатия: сколько пикселей превращается в один символ, а также цветное или монохромное изображение ожидается на выходе. Необходимо добиться лучшей передачи формы, самостоятельно разобраться, к каким трюкам пришли люди. Для этой цели следует в первую очередь отладить до совершенства монохромный режим. В конце же работы надо добавить режим батча: возможность работы с несколькими изображениями сразу. Обработать небольшую секвенцию кадров (видео).
15. Любое достаточно сложное задание, предложенное преподавателем.

### 3 Трёхмерное пространство

#### Структура трёхмерных объектов

С этой главы начинаем погружение в трёхмерную графику. Основной в нашем курсе, как и везде, будет графика полигональная, т.е. такая, где объекты описываются набором полигонов.

Модель — это, в первую очередь, полигональная сетка. Поэтому на английском языке часто используется термин *mesh*. Все полигоны — многоугольники (чаще треугольники и четырёхугольники, но иногда встречается и большее количество вершин).



Рис. 17: Модель и её полигональная сетка

Очевидно, что чем больше полигонов используется, тем более сложную геометрию они могут описать. Низкополигональные (low poly) модели применяются там, где не хватает вычислительных ресурсов, или выбран определённый угловатый стиль визуализации. А для достижения реализма, напротив, нужны высокополигональные (high poly) модели. Но и тут остро стоит вопрос, какой уровень детализации сетки нужно использовать, чтобы зритель или игрок восхищался картинкой, но приложение могло отрисовывать объекты с необходимой скоростью.

О полигональной сетке можно думать как о графе. Поэтому, кстати, многие алгоритмы на них нашли своё применение и в трёхмерной графике. Геометрия задаётся вершинами, где каждая описывается тремя координатами  $v(x, y, z)$ . Между некоторыми из вершин проводятся рёбра. А между рёбрами, лежащими в одной плоскости, образуются многоугольники — полигоны. Перечисленных объектов хватает, чтобы задать фигуру в трёхмерном пространстве.

Но, конечно, одной геометрии недостаточно. Объект будет лучше восприниматься, если на каждом из полигонов будет какое-то изображение. Например, кубик можно будет превратить в модельку ящика для компьютерной игры, если каждая из его граней будет как-то раскрашена.

За это отвечает текстура — изображение (*jpg*, *png* и т.д.), которое обычно хранится где-то рядом с основным файлом модели. В процессе рендеринга, т.е. во время отрисовки объекта эта картинка накладывается на объект, как бы его оборачивая. Дальше по курсу будет разобран механизм такой операции. Но пока скажем, что для этого необходимы текстурные вершины, которые задаются не в трёхмерном пространстве, а на плоскости текстуры —  $vt(u, v)$ . Из-за часто используемых обозначений  $u$  и  $v$  их также могут называть *uv*-координатами, а само изображение — UV Map.

Привязка текстурных координат к модели выполняется не во время задания её трехмерных вершин, а при создании полигона. Вершина может участвовать в разных гранях объекта, где каждый из полигонов окрашен по-своему. Поэтому одной вершине могут соответствовать несколько текстурных координат.

Часто стараются делать текстуру так, чтобы близкие полигоны и на ней располагались рядом. Так по ней можно будет понять, к какой модели она относится. Но из-за того, что мы разворачиваем поверхность трёхмерного объекта на плоскость,



Рис. 18: Трёхмерная модель и её текстура

пропорции на текстуре вовсе не обязаны совпадать с реальными, как это, например, происходит и при переходе от глобуса к карте. Описываемый эффект можно проследить, например, на рисунке 18. Кстати, эта модель хранится в основном в репозитории курса и может использоваться вами для тестов.

Наконец, последняя характеристика модели, которая потребуется нам позже — нормали. Они используются для тех ситуаций, где нужно следить, под каким углом на полигон попал луч. Пожалуй, самая распространённая из них — это задача расчёта освещения на сцене.

Вот только в модели обычно хранят не нормали к полигонам, а нормали к вершинам —  $vn(x, y, z)$ . Сейчас для вас это может показаться абсурдным, ведь у трёхмерной точки не может быть однозначной нормали. Но дальше по курсу станет понятно, для чего выполняется этот трюк.

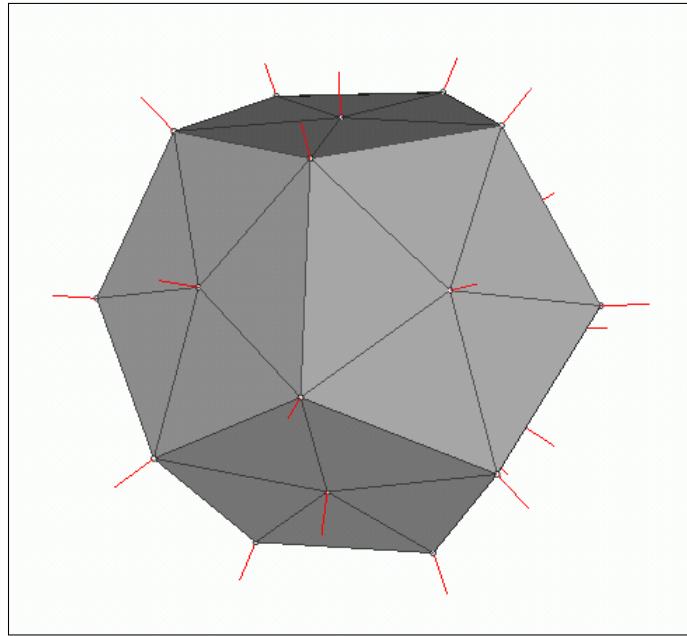


Рис. 19: Нормали вершин

Для того, чтобы перейти от нормалей вершин к нормалям полигонов, надо для каждого полигона взять среднее арифметическое всех нормалей вершин, что его составляют. Верно и обратное. Чтобы вычислить нормаль вершины, нужно сложить нормали полигонов, в которых она участвует, а затем поделить на их количество. При этом длины векторов нормалей обычно никак не используют, важно лишь их направление. Поэтому, чтобы с ними было удобнее работать, вектора обычно нормализуют.

Те, кто уже работал с трёхмерными моделями, наверное, слышали про карты нормалей. Благодаря вот таким заданным в вершинах векторам, они и создаются.

## Задача триангуляции

Полигоны могут быть треугольниками, четырёхугольниками и так далее. Стого говоря, нет никакого ограничения на количество вершин, хотя на практике стараются работать с простыми объектами.

Но есть алгоритмы, которые предполагают, что все полигоны модели — треугольники. Так, например, при написании рендера в финальном задании курса вам потребуется вызывать метод их отрисовки, который реализует кто-то из студентов во второй задаче. При этом нет смысла разрабатывать алгоритм для произвольного многоугольника, т.к. любой из них можно разбить на треугольники. Чуть позже станут понятны и другие причины работать только с тремя вершинами.

*Триангуляцией* в нашей предметной области называется процедура превращения трёхмерной модели в такую, где все полигоны являются треугольниками. Часто в программе хранится как оригинальная модель, так и её версия после триангуляции. Это добавляет гибкость: позволяет работать как с оригинальной сеткой, так и быстро переходить к вызову определённых методов.

Разумеется, вершины, текстурные вершины и нормали никак не изменяются при триангуляции (за исключением некоторых алгоритмов). В простых реализациях единственное, что затрагивается — это полигоны.

Существует большое количество подходов для решения данной задачи. Часто стараются избегать слишком острых или тупых углов. Например, можете самостоятельно разобрать и реализовать *триангуляцию Делоне*. С применением таких алгоритмов можно получить визуально приятную полигональную сетку. Однако в наших задачах форма полигонов после триангуляции не будет важна, мы их даже не увидим. Важно только, чтобы они были треугольниками. Поэтому для реализации предлагается самый примитивный алгоритм.

Возьмём произвольный  $n$ -угольник с индексами вершин от 0 до  $n-1$ . Соединим нулевую вершину со второй, затем с третьей и так далее до  $n-2$ . Таким образом получим  $n-2$  треугольника.

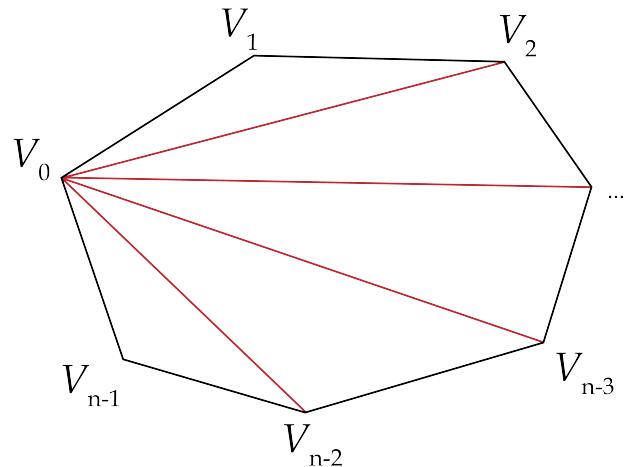


Рис. 20: Триангуляция полигона

## Расчёт нормалей

Нормали как и текстурные вершины могут быть в файле модели не заданы. Более того, возможна ситуация, что они заданы неверно, а это способно привести к некорректной работе алгоритмов. Поэтому в общем случае правилом хорошего тона является самостоятельное вычисление нормалей при загрузке модели в программу.

Отталкиваясь от геометрии объекта, проще всего вычислить нормали к полигонам. Как обсуждалось выше, от них можно легко переходить к нормалям вершин и обратно.

Уже в который раз при решении задачи поможет векторное произведение (см. “Приложение 2. Операции над векторами”). Для удобства предположим, что полигон является треугольником, хотя здесь это необязательно. Попарно соединив вершины, получим два вектора, лежащих в нужной плоскости. Векторное произведение даст результат, перпендикулярный им.

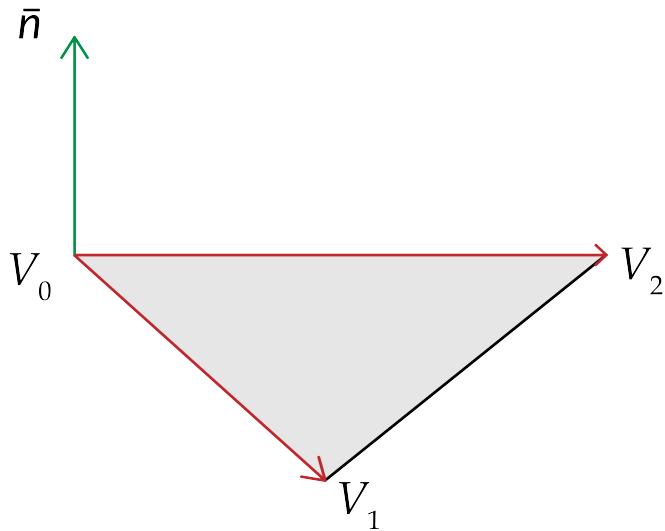


Рис. 21: Вычисление нормали к полигону

Все три вектора составляют правую тройку, однозначно определяя направление. Но чтобы не оказалось так, что нормали бьют внутрь модели, а не наружу, важно учитывать порядок обхода. К счастью, почти любая серьёзная программа подстраивается под общепринятые стандарты, когда прикрепляет вершины к полигонам. Поэтому, если алгоритм заработает правильно для одной модели, для других он, скорее всего, тоже будет пригоден.

И важно не забывать всегда нормализовать результат. Также лучше сразу переходить к нормалям вершин. Именно они будут использоваться нами далее.

## Некоторые важные правила разработки

Задачи в рамках курса постепенно усложняются и, в связи с этим, хочется обсудить некоторые важные принципы разработки. Многократно проверено, что стремление к их соблюдению значительно экономит время, уменьшает вероятность ошибок и делает конечный код лучше.

Мы попытались сформулировать основные идеи, опираясь на свой и чужой опыт. Программы люди пишут аж с середины XX века, поэтому и попыток собрать в одном месте подобные правила было уже очень много. И если начнете их разбирать, быстро увидите, что все они говорят примерно об одном и том же.

Стоит обратить внимание, что подобные принципы носят фундаментальный характер. Для вас лично или в вашем коллективе, конечно, могут действовать свои уникальные договоренности, начиная от стиля кода и заканчивая организацией работы. Но то, что мы обсудим ниже, следует пытаться соблюдать всегда, за исключением тех случаев, когда внешние обстоятельства вносят серьезные корректизы.

### 1. Код следует разбивать на модули, обеспечивая слабую связность их друг с другом.

Под словом “модуль” здесь и ниже будем понимать любую единицу кода: методы, классы, библиотеки, приложения и так далее.

С тем, что большой сегмент желательно разбить на несколько поменьше, вряд ли кто-то будет серьёзно спорить, ведь этому учат с первых курсов программирования. Такое разбиение улучшает читаемость и даёт большую гибкость.

Но, нарезая программу на куски, неопытные разработчики часто забывают обеспечивать слабую связность, то есть получают ситуацию, когда модули друг от друга сильно зависят. Это приводит к тому, что от такого разбиения нет почти никакого толка. Собирая, изменяя или вызывая одну часть кода, мы обязаны тоже самое делать и с другой.

А ведь гораздо проще вести разработку, когда модули независимы в рамках своей деятельности. Да, конечно, более высокоуровневые обязаны вызывать тех, кто уровнем пониже. Но, за исключением некоторых шаблонов, следует при этом стремиться к древовидной структуре, а не графу. Некоторые языки программирования даже не позволяют собирать код с циклическими зависимостями.

Пожалуй, одним из самых ярких выражений данного принципа является микросервисная архитектура. Она была создана для крупных сервисных приложений, где особенно важно быстро и незаметно изменять, пересобирать, перезагружать отдельные компоненты системы. Ощущимый прирост в скорости и надежности наблюдается здесь, если делать модули маленькими и незави-

симыми. При этом те же идеи применимы и для десктопных, мобильных и прочих приложений.

*P.s.* Вообще, термин “связность” часто встречается в теории программирования, но надо уточнять, что вы имеете в виду. Бывает, что под ним понимают разные вещи. Например, в терминологии шаблонов “GRASP” связность — это мера взаимодействия компонентов *внутри* модуля, а то, о чём писалось выше, называется зацеплением. Поэтому в таких терминах стоит стремиться к сильной связности и слабому зацеплению.

## 2. Модули должны принимать и выдавать только то, с чем они работают.

Этот пункт тесно связан с предыдущим, но настолько важен, что его следует обсудить отдельно.

Хоть мы и стремимся к независимости отдельных участков программы, они должны что-то друг о друге знать. Иначе из них не выйдет составить цельное приложение. Но тут возникает вопрос: а какую конкретно информацию они имеют право использовать?

Правилом хорошего тона является создание такой архитектуры, где модули принимают только необходимое для их работы. Например, если функция взаимодействует только с некоторыми полями класса, не надо передавать туда весь объект. Так ваш код будет тесно связан именно с этим классом, вызвать его в другом месте (например, в тестах, о которых мы поговорим позже) вы не сможете. В случае серьёзного изменения класса придётся переделывать и функцию. И, наверное, самое опасное — легко можно допустить ошибку, сделав что-то с теми данными, которые вы не должны были трогать.

Одна из формулировок данного принципа носит название закона Деметры (LoD). Советуем дополнительно разобрать его самостоятельно. В этом законе, например, постулируется, что если у объекта А есть объект В, а у объекта В — объект С, то А не может каким-то образом дать прямой доступ к С. Так, если класс А хранит В, а у В есть `Method()`, то строка `a.b.Method()` нарушит закон. Правильным решением здесь будет вызов `a.Method()`, внутри которого происходит вызов `b.Method()`.

Разумеется, такой подход влечёт за собой создание большого количества обёрток и делегатов. Чтобы как-то бороться с этим, применяют шаблон *внедрения зависимости*. В такой парадигме класс вместо того, чтобы хранить какой-то сервис в качестве поля, даёт возможность передать его себе в методы как аргумент. Доступ к этому сервису больше не оказывается закрыт за кучей прослоек. Так еще и передавать в класс можно разные сервисы, заставляя его работать по-разному. Такой приём очень полюбили при написании тестов.

### 3. Ветвление — необходимое зло.

Вот такая картина не может вызывать ничего кроме ужаса:

```
1 if condition1:  
2     if condition2:  
3         if condition3:  
4             if condition4:  
5                 if condition5:
```

И это здесь ещё опущены фигурные скобки.

Тем не менее, надо признать, что в некоторых ситуациях без условных операторов не обойтись. Первое, что хочется напомнить — так это то, что всегда есть возможность записать логическое выражение в другом виде. Даже если не получилось как-то упростить, можно от кода следующего вида

```
1 if condition:  
2     do something()
```

перейти к такому, где условный оператор будет отвечать за выход из блока и не позволит выполнить код ниже:

```
1 if NOT condition:  
2     return, break, continue, ...  
3 do something()
```

Это не даст сделать ветвление слишком объёмным.

Но ещё лучше будет по возможности избегать любых *if*-ов. Например, если есть сущность с очень разным поведением, почти всегда не стоит городить внутри класса сложную логику для учёта всех условий. Лучше описать базовый класс (абстрактный или интерфейс), а затем наследовать от него и переопределять частные случаи. Такой полиморфизм всегда лучше, чем пачка условных операторов. У вас будет чистый, разнесённый по блокам, читаемый код. Изменяя его, добавляя или удаляя что-то, вам не придётся лезть в один и тот же базовый класс. Вероятность ошибок значительно уменьшится.

Это может быть важно и для оптимизации кода. Мы сейчас, конечно, не говорим о случаях, когда требуется выполнить маленькую дешевую проверку — тут ООП способен даже навредить. Но если какой-то объект может вести себя очень по-разному, а переключения его режимов — штука медленная, гораздо выгоднее будет релизовать разные сущности и скармливать их алгоритму в зависимости от потребностей.

Может быть незаметно относительно более крупных участков кода, но будьте готовы к тому, что *if*-ы могут вас тормозить. Особенно это актуально для параллельного программирования.

#### **4. ООП и шаблоны — важный элемент программирования. Но с ними не стоит перебарщивать.**

Знать принципы ООП, следить за модификаторами доступа, понимать шаблоны (стандартные решения часто встречающихся задач) крайне необходимо. Как и в других примерах, это позволяет экономить время, писать читаемый расширяемый код, уменьшать вероятность ошибок.

Но хочется акцентировать внимание на том, что почти всегда разработка — это поиск золотой середины между идеальным образом и доступными ресурсами. При проектировании инженерного сооружения не всегда ведут расчёты на случай ядерной войны, вместо этого описывают конструкцию и материалы с учётом необходимых нагрузок. Это позволяет сэкономить и уложиться в сроки. Так и при написании программы, если для решения задачи, на которую надо было потратить день, вы будете целую неделю выстраивать себе архитектуру “на будущее”, то это будущее для вашего кода может никогда и не наступить. В таких ситуациях лучше показать что-то в срок, но с умеренным потенциалом для расширения, а после при необходимости переписать некоторые моменты.

#### **5. “Сделай — Сделай правильно — Сделай быстро”.**

Вообще, при решении любой серьёзной задачи редко что-то делается сразу на чистовик. Почти всегда нужен черновой вариант, чтобы понять, а как оно вообще будет выглядеть. При написании программы также советуем не изобретать сразу идеальный инструмент, а сформировать в голове его прототип и начать работу. Может так статья, что ваши задумки окажутся бесполезными, а без эмпирической проверки любые мысленные эксперименты непригодны на практике. Поэтому начать работу нужно со “сделай”. Пусть пока с ошибками, плохой архитектурой, медленной работой, но какой-то результат уже должен быть.

Только получив прототип и проверив часть идей, можно переходить к “сделай правильно”. На этом этапе имеет смысл улучшать архитектуру, вычищать баги. Необходимо стремиться к такому состоянию кода, чтобы им можно было гордиться. Чтобы каждая функция или класс могли быть вашей визитной карточкой.

Конечно, и на первых двух этапах вы подходили к программированию разумно и не тормозили программу умышленно. Но только после написания почти идеального варианта можно переходить к “сделай быстро”. Серьёзная преждевременная оптимизация замедлит разработку и ухудшит финальный вариант. Проводить её нужно, имея корректную рабочую программу, хотя бы потому, что так профилирование покажет реальные узкие (замедляющие приложение) места, а не те проблемы, что вы себе вообразили. И стоит быть готовым к то-

му, что читаемость кода при его оптимизации снизится, в том числе поэтому желательно сперва грамотно выстроить архитектуру.

Описанная триада сама по себе является этапом разработки. При написании крупного модуля её структура может слегка размываться. Например, часть модуля уже может быть в конечном состоянии, другая может работать правильно и ждать своей оптимизации, а третья — создаваться в данный момент в черновом варианте.

## **6. Любая работа — это движение маленькими шагками.**

Наверное, уже очевидно, что разработка состоит из последовательных итераций. Это могут быть этапы, описанные в предыдущем пункте, или более глобальные компоненты вроде постоянно сменяющих друг друга написания кода и его тестирования.

При этом много раз проверено, что чем меньшими шагами вы идёте, тем быстрее движетесь. Большие по объёму этапы сложно держать в голове, на них легко допустить ошибки и застрять. И психологически они тоже изматывают.

Но особенно ярко это раскрывается при работе в команде. Если возьмёте себе крупную задачу и будете работать над ней в отдельной ветке до тех пор, пока не закончите, вы рискуете потерять связь с коллективом. Но даже если будете следить за обновлениями, за годы вашей работы репозиторий с кодом коллег может измениться до неузнаваемости. И у вас после уйдут месяцы на то, чтобы вставить свой модуль к другим.

Дабы избегать таких ситуаций, желательно разбивать задачу на маленькие сегменты, часто вливать друг в друга ветки. Если познакомитесь с парадигмами организации командной работы вроде git flow, trunk-based development и др., то заметите, что везде обсуждается необходимость делать коммиты как можно меньше. Такие небольшие структурные единицы проще воспринимать и поддерживать.

## **7. Тестирование должно быть не прихотью, а обязанностью.**

Только тестирование кода может доказать его работоспособность. Оно должно выполняться не по желанию программиста, а по строгой необходимости.

Конечно, на крупных проектах нанимают отдельных специалистов — тестировщиков, цель которых создавать и запускать тесты. Но хороший разработчик должен сам знать, как протестировать его программу.

Существует несколько подходов, позволяющих проводить тестирование быстро (чуть ли не по нажатию одной кнопки) и качественно, с учетом специфики кода. В следующем параграфе подробнее распишем некоторые ключевые идеи.

# О тестах и их важности

Итак, вы добавили что-то в программу. Как убедиться, что код рабочий?

Конечно, можно (и нужно) запустить приложение и сделать ряд действий, чтобы вызвать необходимый модуль. Но может быть так, что при одних входных данных он выдаёт корректный результат, а на других программа падает. Хотелось бы иметь механизм, который позволяет быстро проверить все случаи.

Более того, когда правки вносятся только в небольшую часть приложения, не хочется каждый раз для отладки пересобирать, запускать и тестируировать всю программу целиком. Это очень сильно замедлит любую разработку.

Обе эти проблемы решает написание так называемых **юнит-тестов**. Их цель — проверить работоспособность конкретного модуля (поэтому часто также говорят о модульном тестировании). Действительно, при проектировании ракеты не производятся же запуски каждый раз, когда нужно проверить двигатель, обшивку или корпус. Вместо этого все эти компоненты сперва тестируются отдельно. Так и нам в целях экономии ресурсов в первую очередь следует проверять те небольшие модули, что мы создаём.

Общая структура юнит-теста выглядит так:

```
1 testFoo() {  
2     values = ...  
3     result = foo(values)  
4     expected = ...  
5     result == expected?  
6 }
```

При тестировании некого модуля указываем входные параметры (*values*) и ожидаемое на выходе значение (*expected*). Результат работы модуля сравниваем с тем, что хотим получить. Если они совпадают, тест прошёл. Если нет — значит, мы нашли ошибку в коде.

Нет необходимости писать свои обёртки для подобных конструкций. Любой уважающий себя язык программирования имеет несколько популярных фреймворков для юнит-тестирования, а среди разработки наглядно отображают результат работы. В вспомогательном проекте к этой главе есть небольшой пример такого использования.

Обсудим еще некоторые важные достоинства юнит-тестов. Во-первых, для того, чтобы была возможность протестировать отдельные части программы, да ещё и с разными входными условиями, приложение придётся сразу писать с грамотной архитектурой. Таким образом, тесты дополнитель но заставляют код быть читаемым, расширяемым, универсальным.

Во-вторых, юнит-тесты можно рассматривать как приложение к документации. Их изучение поможет постороннему человеку лучше понять, как должен работать модуль.

В-третьих, сама постановка задачи придумывания тестов заставляет программиста размышлять над тем, какие же входные данные могут к нему прийти. А уже одно только это уменьшает вероятность ошибок.

Наконец, в-четвертых, хотя это ещё не все преимущества, тесты упрощают жизнь при изменении модуля, рефакторинге. Иногда приходится вернуться к коду спустя годы и что-то в нём подправить. В таких ситуациях особенно важно убедиться, что вы ничего не сломали. Если действовать аккуратно и после каждой небольшой правки запускать тесты, ошибка в старом поведении не должна будет появиться.

То, что модули работают корректно по отдельности, ещё не значит, что и приложение будет выполняться без ошибок. Отдельные части программы вполне могут вызываться неправильно или конфликтовать друг с другом. Для того, чтобы отслеживать такие ситуации, пишутся так называемые **интеграционные тесты**. Их цель — убедиться, что модули работают вместе правильно.

Отдельный вопрос — а что вообще должно быть в тестах? В идеальном случае покрытие кода ими должно быть стопроцентным. Лучше сделать плохой тест, чем никакой вообще. Проверять при этом следует как типовые случаи (те с которыми будет работать пользователь), так и крайние варианты: нулевые аргументы, выходы за допустимые диапазоны значений и т.д.

Когда писать тесты — это уже зависит от того, как вам удобнее. Некоторые покрывают ими код в самом конце, когда довели его до идеального состояния. Кто-то старается писать их в процессе, тем самым ускоряя чистку от ошибок. Есть также парадигма разработки через тестирование — это когда тесты создаются в самом начале, а уже после под них выстраивается модуль. Тут у каждого из подходов есть свои достоинства и недостатки.

## Чтение Obj-формата

Теперь, когда вы знакомы со структурой трёхмерной полигональной модели, настало время обсудить, как они хранятся на диске. Существует большое количество форматов для самых разных задач (FBX, STL и так далее). В рамках данного курса мы будем использовать один из самых простых — формат OBJ.

Это текстовые файлы, то есть вы можете понять структуру модели, открыв их в любом текстовом редакторе. В данном формате каждая из строк файла добавляет в модель какой-то новый компонент.

Например, так могут выглядеть строки, описывающие вершины:

```
1 v 1.5 2.8 0.1
2 v 0.4 1.2 4.0
3 v 2.2 -0.4 -0.3
4 ...
```

Сначала идёт символ *v*, обозначающий vertex, то есть вершину. А затем через пробел указываются три её координаты.

Текстурные вершины записываются следующим образом:

```
1 vt 0.3 0.7
2 vt 0.5 0.25
3 ...
```

Тут в начале строки стоит *vt*, а координат будет всего две, потому что задаётся точка на плоскости.

Наконец, нормали к вершинам будут задаваться строками следующего вида:

```
1 vn 0.66 0.66 -0.33
```

Напомним, что в модели могут быть не указаны текстурные координаты и нормали. Также в файле нет какого-то строгого порядка, что за чем задавать. Вполне могут быть подобные ситуации:

```
1 v ...
2 vt ...
3 ...
4 vn ...
5 v ...
6 ...
7 vt ...
8 vt ...
```

Но по мере того, как встречаются элементы одного и того же типа, им присваиваются номера, начиная с единицы. Первый раз встретилась строка с *v* — это будет первая вершина. Далее пойдут вторая, третья и так далее. Аналогично своя собственная нумерация будет у текстурных координат и нормалей.

Эти индексы могут использоваться при описании полигонов. В отличие от объектов выше, их можно задать разными способами. Например, эта строка указывает, что в модели есть полигон, состоящий из первой, третьей и пятой вершин:

```
1 f 1 3 5
```

Разумеется, их количество может быть произвольным (но не меньше трёх):

```
1 f 1 3 5 7  
2 f 10 20 12 22 40 42
```

Буква *f* обозначает face — так тоже иногда называют полигоны.

Если у модели есть текстура, то её с помощью текстурных вершин следует привязать к объекту. Напомним, что делается это в момент описания полигонов. Стока в таком случае должна выглядеть следующим образом:

```
1 f 1/2 3/4 5/6
```

В данном примере полигон состоит из трёх вершин (указаны их индексы в файле). И к этим точкам треугольника прикреплены вторая, четвертая и шестая текстурные координаты соответственно.

Если в файле хранятся нормали, можно аналогично через ещё одну косую черту привязать их к полигону по индексам:

```
1 f 1/2/1 3/4/5 5/6/10
```

А если есть вершины и нормали, но не указаны текстурные координаты, то строка примет следующий вид:

```
1 f 1//1 3//5 5//10
```

В целом разобранных случаев достаточно для выполнения заданий курса. Конечно, структура модели несколько сложнее, а в Obj-файле можно встретить и другие строки. Но для наших целей они не понадобятся. Тем не менее всё же советуем разобрать этот формат чуть более подробно. Он даст лучшее понимание тех объектов, с которыми мы будем работать:

[https://en.wikipedia.org/wiki/Wavefront\\_.obj\\_file](https://en.wikipedia.org/wiki/Wavefront_.obj_file)

В основном репозитории курса хранятся модели для тестов. Вы можете как использовать их, так и скачать или сделать свои собственные:

<https://github.com/kv1tr4vn/CGVSU/tree/main/3DModels>



Посмотрим, как можно загрузить модель в программу. Для этих целей был написан вспомогательный проект:

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task3/ObjReaderInitial>

Класс модели в нём содержит следующие поля:

```
1 public ArrayList<Vector3f> vertices;
2 public ArrayList<Vector2f> textureVertices;
3 public ArrayList<Vector3f> normals;
4 public ArrayList<Polygon> polygons;
```

Как и обсуждалось выше, модель — это набор массивов из четырёх элементов. Вершины и нормали задаются трёхмерными векторами. Текстурные координаты — двумерными. Полигон — это отдельный класс, задаваемый следующими полями:

```
1 private ArrayList<Integer> vertexIndices;
2 private ArrayList<Integer> textureVertexIndices;
3 private ArrayList<Integer> normalIndices;
```

Здесь указываются индексы вершин, текстурных координат и нормалей в том порядке, в котором они привязывались в Obj-файле. Только для удобства при чтении из них вычитается единица, чтобы в программе нумерация шла с нуля. В третьей и четвертой задачах вы имеете право отталкиваться от этого кода при работе с моделями.

Чтение формата реализовано в классе *ObjReader*. Сейчас не будем подробно расписывать его структуру. При разборе кода обратите внимание на следующие детали:

1. Основной метод чтения файла разбит на группу вспомогательных функций, позволяющих не только улучшить читаемость кода, но и подготовить его для удобного тестирования.
2. Заложена основа для обработки ошибок. В Obj-файле модель может быть задана некорректно. Например, у полигона могут быть указаны только две вершины. Чтобы программа не зависала и не падала, важно учитывать, что пользователь может давать ей ошибочные данные. Хорошее приложение — то, которое способно всё это предусмотреть.
3. Начата работа над тестированием модуля. Отдельный класс *ObjReaderTest* покрывает проверками часть методов основного кода. Используется готовый фреймворк, а среда разработки позволяет хранить и запускать тесты в удобном формате.

В соответствии с той структурой, о которой писалось выше, самый простой юнит-тест выглядит следующим образом:

```
1  @Test
2  public void testParseVertex01() {
3      ArrayList<String> wordsInLineWithoutToken = new
4          ArrayList<>(Arrays.asList("1.01", "1.02", "1.03"
5              ));
6      Vector3f result = ObjReader.parseVertex(
7          wordsInLineWithoutToken, 5);
8      Vector3f expectedResult = new Vector3f(1.01f, 1.02
9          f, 1.03f);
10     Assertions.assertTrue(result.equals(expectedResult
11         ));
12 }
```

В теле метода задаются входные данные и ожидаемое значение. На последней строчке результат работы сравнивается с тем, что мы хотели получить. Для данного примера тест проходит, поэтому можно сказать, что на таких данных тестируемый *ObjReader.parseVertex()* работает.

## Аффинные преобразования

Сама по себе трёхмерная модель уже может представлять интерес. Но большего практического применения возможно достичь, если научиться её деформировать. Более того, в ряде задач важен не один конкретный объект, а целая сцена, состоящая из множества моделей. В связи с этим их нужно как-то размещать по всему объёму.

Для того, чтобы трансформировать модель, нужно изменять её вершины, ведь именно в них заложена геометрия объекта. Поэтому задачу можно свести к следующей: дан трёхмерный вектор  $\vec{v}(x, y, z)$ . Необходимо на основе неких операций получить новый  $\vec{v}'(x', y', z')$ .

С точки зрения нашей практики наиболее важными будут аффинные преобразования. Это операции, которые можно записать в следующем виде:

$$\vec{v}' = M\vec{v} + \vec{t},$$

где  $M$  — обратимая матрица, а  $\vec{t}$  — вектор той же размерности, что и  $\vec{v}$ .

Раз уж приходится вводить новый термин, обсудим некоторые взаимосвязи между часто используемыми понятиями. До этого ещё со школы вы много работали с линейными преобразованиями, но, скорее всего, не задавались вопросом: а что делает оператор линейным?

Так вот преобразование называется линейным, если для него выполняются следующие равенства:

$$\begin{aligned} L(\vec{x} + \vec{y}) &= L(\vec{x}) + L(\vec{y}) \\ L(\alpha\vec{x}) &= \alpha L(\vec{x}) \end{aligned}$$

Их ещё называют свойствами линейности. Проверим, выполняются ли они для аффинного преобразования:

$$\begin{aligned} \left( A(\vec{x} + \vec{y}) = M(\vec{x} + \vec{y}) + \vec{t} \right) &\neq \left( A(\vec{x}) + A(\vec{y}) = M\vec{x} + \vec{t} + M\vec{y} + \vec{t} = M(\vec{x} + \vec{y}) + 2\vec{t} \right) \\ \left( A(\alpha\vec{x}) = \alpha M\vec{x} + \vec{t} \right) &\neq \left( \alpha A(\vec{x}) = \alpha M\vec{x} + \alpha\vec{t} \right) \end{aligned}$$

Как видно, аффинное преобразование не является линейным, этому мешает слагаемое  $\vec{t}$ . То есть линейный оператор является частным случаем аффинного при  $t = 0$ . Такое утверждение приводится здесь, чтобы вы понимали: не всё, что было допустимо в случае линейности, может быть использовано и для преобразований, о которых дальше пойдёт речь.

По возможности, будем записывать операции в виде матриц (матричных операторов). Воспринимать запись вида  $A\vec{x}$  можно как применение преобразования  $A$  к вектору  $\vec{x}$ .

В коде также часто стараются задавать операции матрицами. Например, потому, что перемножив их между собой, можно получить в качестве произведения матрицу, описывающую за раз все эти преобразования в нужном порядке. Так, если к вектору  $\vec{x}$  сначала применяется  $A$ , а потом  $B$ , то в силу ассоциативности:

$$BA\vec{x} = B(A\vec{x}) = (BA)\vec{x} = C\vec{x},$$

где  $C$  задаёт последовательное применение сначала  $A$ , а потом  $B$ .

Здесь и ниже предполагается, что  $\vec{x}$  — вектор-столбец. Все основные преобразования будут записаны для этого случая. Напомним, что для того, чтобы перейти к векторам-строками и оставить описание той же самой операции, придётся транспонировать матрицы и менять порядок перемножения на обратный. Подробнее такой переход разобран в “Приложение 2. Операции над векторами”.

Можно привести много примеров аффинных преобразований. Но для целей нашего курса понадобятся только три часто используемых: масштабирование, поворот и параллельный перенос.

## Масштабирование

Масштабирование — самая простая операция. На английском языке её называют *scale*, поэтому и латинская буква, описывающая соответствующую матрицу, будет  $S$ .

Если задан вектор  $\vec{v}(x, y, z)$ , то задача масштабирования — получить  $\vec{v}'(x', y', z')$  такой, что  $x' = s_x x$ ,  $y' = s_y y$ ,  $z' = s_z z$ .

Здесь  $s_x$ ,  $s_y$  и  $s_z$  — некоторые коэффициенты. Применяя такую операцию к каждой вершине модели можно:

1. Растирнуть её вдоль оси, если коэффициент для неё больше единицы.
2. Оставить без изменения при  $s = 1$ .
3. Сжать модель вдоль оси к началу координат, если  $0 \leq s < 1$ .
4. Отразить объект относительно начала координат, если  $s$  отрицательный.

Записать масштабирование можно в виде следующей диагональной матрицы:

$$S\vec{v} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \end{pmatrix}$$

Легко заметить, что масштабирование является линейной операцией.

## Поворот

В самом простом варианте операцию можно описать так: мир и все его объекты совершают поворот на заданный угол вокруг заданной оси. Какие будут координаты у конкретной точки относительно старой системы координат?

Эта задача гораздо сложнее для восприятия, поэтому для начала рассмотрим её в двумерном пространстве.

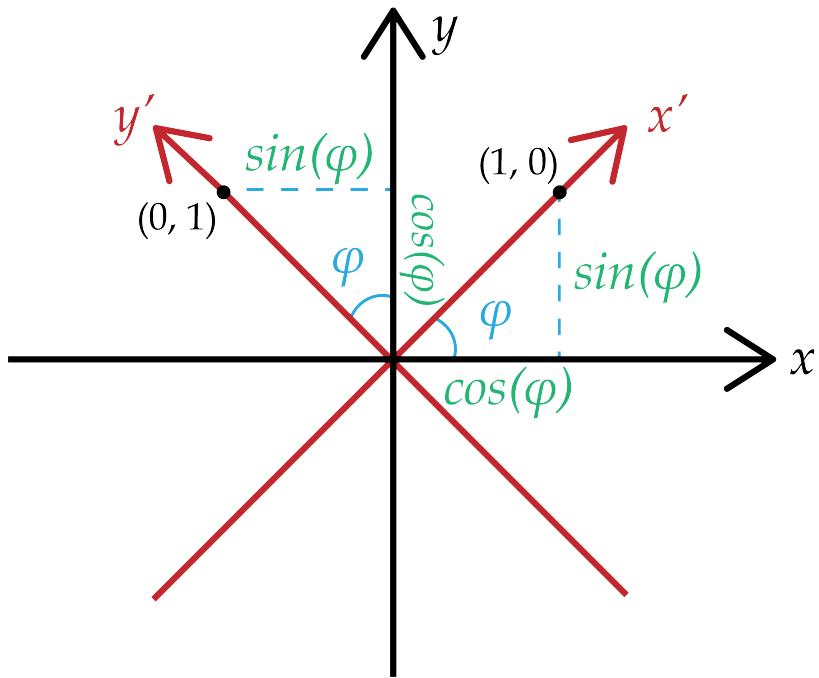


Рис. 22: Поворот на плоскости

Предположим, что система координат совершает поворот против часовой стрелки на угол  $\phi$  (см. рисунок 22). Точки  $(1, 0)$  и  $(0, 1)$  также перемещаются и приобретают координаты относительно старой системы  $(\cos(\phi), \sin(\phi))$  и  $(-\sin(\phi), \cos(\phi))$  соответственно. А координаты  $(1, 1)$ , например, при повороте на  $0.25\pi$  перейдут в  $(0, \sqrt{2})$ .

В общем случае, проверяя любую другую точку, вы всё время будете получать одну и ту же закономерность:

$$\begin{aligned} x' &= x \cos(\phi) - y \sin(\phi), \\ y' &= x \sin(\phi) + y \cos(\phi), \end{aligned}$$

где  $(x', y')$  — новые координаты.

Поворот на английском носит название *rotation*. Поэтому соответствующая матрица обозначается  $R$ . Для двумерного случая можно записать:

$$R_2 \vec{v} = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Но чаще угол  $\phi$  направляют в другую сторону, по часовой стрелке. Тогда аргумент следует взять с отрицательным знаком. Косинус — чётная функция, а синус — нечётная. Поэтому матрица поворота принимает вид:

$$R'_2 = \begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix}$$

Обе формы записи имеют право на существование. Нужно только сперва согласовать, в какую сторону знак угла положителен.

Теперь распространим это решение на трёхмерный случай. Допустим, поворот осуществляется вокруг оси  $z$  на тот же угол  $\phi$ . Тогда верно следующее:

$$\begin{cases} x' = x \cos(\phi) + y \sin(\phi) \\ y' = -x \sin(\phi) + y \cos(\phi) \\ z' = z \end{cases}$$

С помощью матрицы данную операцию можно описать так:

$$R_z \vec{v} = \begin{pmatrix} \cos(\phi) & \sin(\phi) & 0 \\ -\sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Аналогично для поворота вокруг  $y$  на угол  $\psi$ :

$$R_y = \begin{pmatrix} \cos(\psi) & 0 & \sin(\psi) \\ 0 & 1 & 0 \\ -\sin(\psi) & 0 & \cos(\psi) \end{pmatrix}$$

И для поворота вокруг  $x$  на  $\theta$ :

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

Перемножив  $R_x$ ,  $R_y$  и  $R_z$  в нужном порядке, можно получить универсальную матрицу поворота сразу вокруг каждой из осей. Это предлагается сделать читателю самостоятельно.

Обратите внимание, что поворот — все еще линейное преобразование.

*P.s.* Способ, который мы используем для реализации поворота, носит название углов Эйлера. Это хорошая точка входа в трёхмерную графику, однако в системах более сложных, чем та, что вы создадите в финальном проекте, у них могут открыться недостатки. Любопытствующим студентам рекомендуем разобрать алгоритм поворота, основанный на кватернионах.

## Параллельный перенос (смещение)

Параллельным переносом (translation) называется следующая операция:

$$\vec{v}' = \vec{v} + \vec{t}$$

Она уже не является линейной, и это накладывает свои особенности. Так, для работы с трёхмерными векторами нет возможности задавать её матрицей 3 на 3. А работать с ними очень хотелось бы, тем более, что для всего остального механизма отлажен. Поэтому был придуман следующий трюк.

На время выполнения операции совершается переход в четырёхмерное пространство, а после вектор возвращают обратно.

Для начала получим  $\vec{v}_4$ , дописав ему еще одну координату  $w$ :

$$\vec{v}_4 = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Добавляемая  $w$  носит название однородной координаты и по умолчанию должна быть равна единице. Операцию смещения тогда можно записать так:

$$T_4 \vec{v}_4 = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x + t_x w \\ y + t_y w \\ z + t_z w \\ w \end{pmatrix}$$

Очевидно, что при  $w = 1$  получаем необходимое смещение  $\vec{v}$  на  $\vec{t}$ . Чуть позже познакомимся с другими операциями, где также будет использоваться однородная координата. Там будут возникать ситуации, когда она отлична от единицы. Чтобы приводить вектор в порядок, придётся делить результат на его однородную координату.

Но пока  $w$  всегда равна одному. Для того, чтобы иметь возможность объединить операции масштабирования, поворота и переноса, придётся и матрицы для первых двух перекинуть в четырёхмерное пространство. Так как они тоже не должны влиять на однородную координату, правило перехода принимает следующий вид:

$$M_4 = \begin{pmatrix} M_3 & 0 \\ 0 & 1 \end{pmatrix}$$

Например, для матрицы масштабирования:

$$S_4 = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Итак, получаем следующий алгоритм:

1. Для всех вершин модели перевести  $\vec{v}$  в  $\vec{v}_4$ , дописав однородную координату  $w = 1$ .
2. На основе 9 параметров:  $s_x, s_y, s_z, \phi, \psi, \theta, t_x, t_y$  и  $t_z$  составить матрицы  $S, R$  и  $T$ .
3. Умножить  $\vec{v}'_4 = TRS\vec{v}_4$ .
4. Достать из  $\vec{v}'_4$  трёхмерный  $\vec{v}'$ .

Для векторов-строк, очевидно, умножение будет выглядеть так:

$$\vec{v}'_4^\top = \vec{v}_4^\top S^\top R^\top T^\top$$

Поэтому не смущайтесь, если в другой литературе будете встречать не те матрицы, что выведены выше, а их транспонированные версии. Это лишь другая форма записи.

Но обратите внимание, что важен порядок операций. Сначала идёт масштабирование, потом поворот и после перенос. Например, если поменять местами последние две, то из-за того, что поворот осуществляется относительно центра координат, смещение будет в нём учтено. В итоге вместо того, чтобы повернуться вокруг своей оси, модель может улететь за границы экрана.

## Варианты заданий

**Важно:** как и в предыдущем задании, обязательным условием для выполнения задачи является грамотное использование Git.

Кроме того, теперь требуется, чтобы все ключевые места программы были покрыты юнит-тестами, рассматривающими поведение кода как в рядовых, так и в крайних случаях. Детский пример: если в методе происходит деление на число, то следует проверить и то, что операция реализована верно, и то, что присутствует обработка деления на ноль.

Почти все задания ниже составлены таким образом, что написанный вами код будет использоваться позже в рамках финальной задачи курса. Вызывать ваши методы будут и другие студенты. Поэтому относитесь к своей работе нужно особенно ответственно.

1. Отталкиваясь от примера, доработать и вычистить от багов модуль ObjReader так, чтобы в нём была обработка ошибок на почти все возможные случаи. Задача, на самом деле, сложнее, чем кажется. Код для чтения входных файлов писать сложно, потому что необходимо предусмотреть огромное количество ситуаций с неверными данными со стороны пользователя. И форматов, разновидностей файлов, как правило, приходится поддерживать несколько. В текущей реализации ObjReader, например, пока нет ни намёка на обработку случая, когда в модели нет вершин/полигонов. Или когда к полигону прикреплена сотая вершина в то время, как в файле их всего две. Ещё нынешний ObjReader спокойно пропустит модель, у которой к некоторым полигонам привязана текстура, а к некоторым — нет, а если дальше при разработке не закладываться на такой случай, пользователь получит падение программы во время отрисовки. И это всё только верхушка айсберга. А ведь есть ещё и вечная проблема с кириллицей и пробелами в путях. Чтобы учесть всё это, нужно помимо юнит-тестов загрузить большое количество самых разных файлов.

Помимо возможных ошибок, надо учесть, что и сами Obj-файлы в разных приложениях сохраняются по-разному. Заложиться под всё сейчас невозможно, но, как минимум, изучите самостоятельно и добавьте обработку относительных (отрицательных) индексов.

2. Желательно выполнить эту задачу как можно быстрее, потому что ваш код может быть полезен другим студентам уже в этом задании.

Написать модуль ObjWriter, предназначенный для сохранения модели в формат Obj. Помимо юнит-тестов, в качестве проверки корректности работы можно загрузить модель с помощью ObjReader, а после выгрузить обратно в текстовый формат и сравнить файлы. Несмотря на то, что код придётся писать с нуля, эта задача проще с той точки зрения, что в ней не придётся или почти не придётся заниматься обработкой ошибок. За исключением некоторых базовых

проверок считаем, что модели, приходящие для сохранения из нашего кода, точно адекватные.

3. Написать модуль для триангуляции. Взять для реализации можно самый примитивный алгоритм, просто соединяющий вершины. Но желающие могут по-пробовать и что-то посложнее. Разработать для хранения модели, прошёдшей операцию, отдельный класс, с наследованием его от базового, так как часть полей и методов у них должна быть общей. Помимо юнит-тестов, для проверки можно загрузить модель в программу с помощью ObjReader, обработать, а после выгрузить через ObjWriter (код попросить у товарищай). Посмотреть на результат в стороннем софте.
4. Написать программу для вычисления нормалей модели. На вход с помощью ObjReader поступает модель. Но вместо того, чтобы брать нормали из файла (если они вообще там есть), теперь проводятся самостоятельные вычисления на основе векторного произведения и координат вершин. Результат, помимо юнит-тестов, можно сверить с файлами, где нормали посчитаны и сохранены с помощью стороннего софта.
5. Написать программу для удаления вершин. Пользователь загружает модель, выбирает номера вершин, которые следует удалить и получает результат. Сложность в том, что при удалении вершины полигоны, к которым она прикреплена, следует также удалить. Иначе модель можно считать некорректной. Если получится взять ObjWriter у товарищай, будет полезно загрузить модель в программу, обработать и выгрузить, чтобы посмотреть на неё в стороннем софте.
6. Написать программу для удаления полигонов. Пользователь загружает модель, выбирает номера полигонов, которые следует удалить и получает результат. Сложность в том, что в модели могут возникнуть свободные вершины — такие, которые не прикреплены ни к какому полигону. Следует добавить возможность пользователю выбрать, стоит их удалить или оставить. Если получится взять ObjWriter у товарищай, будет полезно загрузить модель в программу, обработать и выгрузить, чтобы посмотреть на неё в стороннем софте.
7. Написать библиотеку для работы с линейной алгеброй. Результат следует оформить в виде отдельного модуля (пакета). Назвать можно, например, Math. Реализовать несколько классов для работы:
  - Вектора размерности 2, 3, 4 (это отдельные классы). Для них должны быть:
    - Операции сложения и вычитания
    - Умножения и деления на скаляр

- Вычисления длины
- Нормализации
- Скалярного произведения
- Векторного произведения (Для вектора размерности 3)
- Матрицы размерности 3x3, 4x4 (это отдельные классы). Для них должны быть:
  - Операции сложения и вычитания
  - Умножения на соответствующий вектор 3 или 4. Здесь и дальше по курсу работаем с векторами-столбцами
  - Перемножения матриц
  - Транспонирования
  - Быстрого задания единичной и нулевой матрицы (через конструктор или метод)

Особенно важно покрыть весь код тестами, чтобы нажатием пары кнопок можно было доказать, что он рабочий. Стоит также добавить обработку ошибок (например, деление на ноль).

Желающие могут добавить (возможно, частично) для существующих классов пункты из следующего списка - поощряется дополнительными баллами:

- Вычисление определителя матриц
- Вычисление обратной матрицы
- Решение систем методом Гаусса

8. Написать программу для аффинных преобразований модели. Пользователь загружает модель с помощью ObjReader, а после может:

- Переместить модель на произвольный вектор ( $x, y, z$ ),
- Повернуть вокруг каждой из осей на определённый угол,
- Масштабировать вдоль каждой из осей, растянуть или сжать.

Каждая из операция в коде должна задаваться матрицей. Для работы с ними можно либо взять код у студента, разрабатывающего библиотеку для линейной алгебры (предыдущий вариант), либо воспользоваться пакетом javax.vecmath.

Желательно также взять у кого-нибудь ObjWriter, чтобы можно было созраниТЬ результат преобразований и посмотреть на него в стороннем софте.

# 4 Софтверный рендер

## Графический конвейер

В финальном проекте курса подходим к задаче визуализации модели. Вообще, получение изображения объекта по его описанию называется *рендерингом*. В случае трёхмерных моделей речь идёт о *3d-рендеринге*.

В нашем случае рендер будет *программным* (software), то есть код должен выполняться без использования видеокарты. Такая задача не теряет свою актуальность и среди разработчиков крупного ПО, а в рамках данного курса она позволит по-настоящему прочувствовать все алгоритмы, что стоят за отрисовкой.

За отображением трёхмерной модели на плоскости экрана стоит несколько переходов из одной системы координат в другую. Все они вместе называются графическим конвейером, а точнее его первым и необходимым этапом. Как и в главе выше, за каждый переход будет отвечать своя матрица. Умножая их поочереди на трёхмерные вершины моделей, будем получать в итоге соответствующие координаты экрана.

Ниже по порядку разобраны все необходимые преобразования координат.

### Из локальной системы координат в мировую

Модели на сцене могут располагаться в разных точках, с разными углами поворота и масштабом. Однако хранить эту информацию в файле с геометрией неправильно: так, например, можно потерять возможность использовать один и тот же объект в нескольких местах. Поэтому для гибкости работы вершины моделей записывают в файлах так, чтобы центр фигуры был около начала координат.

В связи с этим говорят, что модель задана в своей **локальной** системе координат. Сцена же состоит из многих объектов, разнесённых по разным точкам. Система координат, отвечающая за взаимное расположение моделей на ней, называется **мировой**.

Переход из локальной в мировую системы заключается в том, чтобы взять все вершины конкретной модели, а затем масштабировать ( $S$ ), повернуть ( $R$ ) и переместить ( $T$ ) их. Такие преобразования подробно разбирались в предыдущей главе. Перемножив соответствующие матрицы, можно получить матричный оператор, совершающий необходимый переход.

$$M = TRS$$

Напомним, что здесь и ниже предполагается работа с векторами-столбцами.

Матрица  $M$  уникальна для каждого объекта, поэтому и называется **матрицей модели** (*model matrix*).

## Из мировой системы координат в камеру

Переходим к описанию того, что необходимо для отображения модели на экране. Самый важный объект на этом этапе — камера. Она описывается своей позицией в мировых координатах (иногда соответствующую переменную обозначают  $e\vec{y}e$ ) и целевой точкой (*target*), куда она смотрит. Зная эти два вектора, можно получить **систему координат камеры**. Делается это следующим образом:

1. Получим вектор  $\vec{z} = \vec{target} - e\vec{y}e$ , отвечающий за направление взгляда камеры.
2. Возьмем произвольный  $\vec{up}$ , примерно задающий направление оси вверх. Например, он может иметь мировые координаты  $(0, 1, 0)$ .
3. Найдем  $\vec{x}$  как  $\vec{x} = \vec{up} \times \vec{z}$ . Результат, очевидно, будет перпендикулярен обоим векторам.
4. Временный  $\vec{up}$  брался для вычисления  $\vec{x}$ . Теперь получим правильный  $\vec{y} = \vec{z} \times \vec{x}$ .
5. Нормализуем  $\vec{x}$ ,  $\vec{y}$  и  $\vec{z}$ . В итоге имеем прямоугольную систему координат камеры, записанную в мировых координатах сцены.

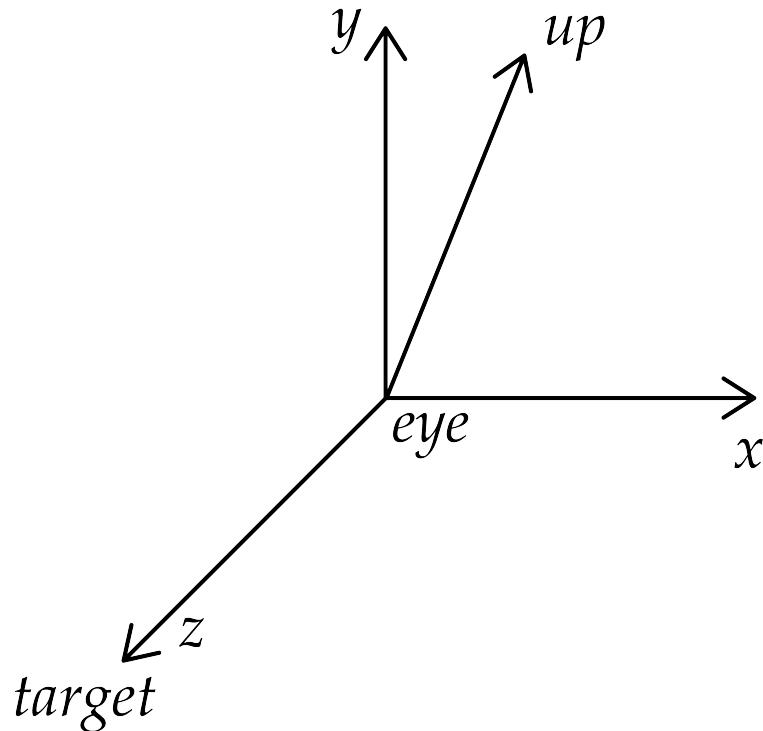


Рис. 23: Вычисление системы координат камеры

Основная задача на этом этапе — перевести все объекты сцены в систему координат камеры. Сделать это можно в два этапа:

1. Сопоставить начало мировой системы координат с центром камеры.
2. Спроецировать все мировые координаты моделей на полученные выше  $\vec{x}$ ,  $\vec{y}$  и  $\vec{z}$  камеры.

Для первого этапа можно использовать матрицу перемещения:

$$T_v = \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Второй этап основан на операции скалярного произведения, позволяющей дать проекцию одного вектора на другой. Соответствующая матрица будет выглядеть так:

$$P_v = \begin{pmatrix} x_x & x_y & x_z & 0 \\ y_x & y_y & y_z & 0 \\ z_x & z_y & z_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где, например,  $x_x$ ,  $x_y$  и  $x_z$  — мировые координаты вектора  $\vec{x}$  камеры.

Теперь вы можете самостоятельно вычислить произведение матриц и найти:

$$V = P_v T_v$$

Полученная **видовая** матрица (*view matrix*) отвечает за переход в систему координат камеры. Соответствующую операцию в библиотеках иногда называют *look at*.

## Из системы координат камеры в плоскость проецирования

Только находясь в системе координат камеры, можно спроектировать трёхмерные объекты на плоскость экрана (напомним, что мы вынуждены работать с четырёхмерными векторами). Соответствующая матрица носит название **проекционной** (*projection matrix* или *clip matrix*).

Для этих целей потребуются следующие характеристики, обычно также хранящиеся в объекте камеры:

1. fov (field of view) — поле зрения, показывающее, какое угловое пространство видно с этой камеры. В нашем случае для удобства fov задаётся как половина от всего угла.
2. ar (aspect ratio) — соотношение сторон. Это обычно отношение либо ширины к высоте экрана, либо наоборот.
3. f (far plane) — так называемая дальняя плоскость, задающая максимум по глубине для точек, которые мы хотим видеть на проекции.
4. n (near plane) — ближняя плоскость, аналогично задающая минимальное  $z$  для точек, которые должны быть на экране.

В зависимости от итоговой системы координат в других источниках могут быть слегка отличающиеся матрицы. Мы будем решать задачу, изображенную на рисунке 24:

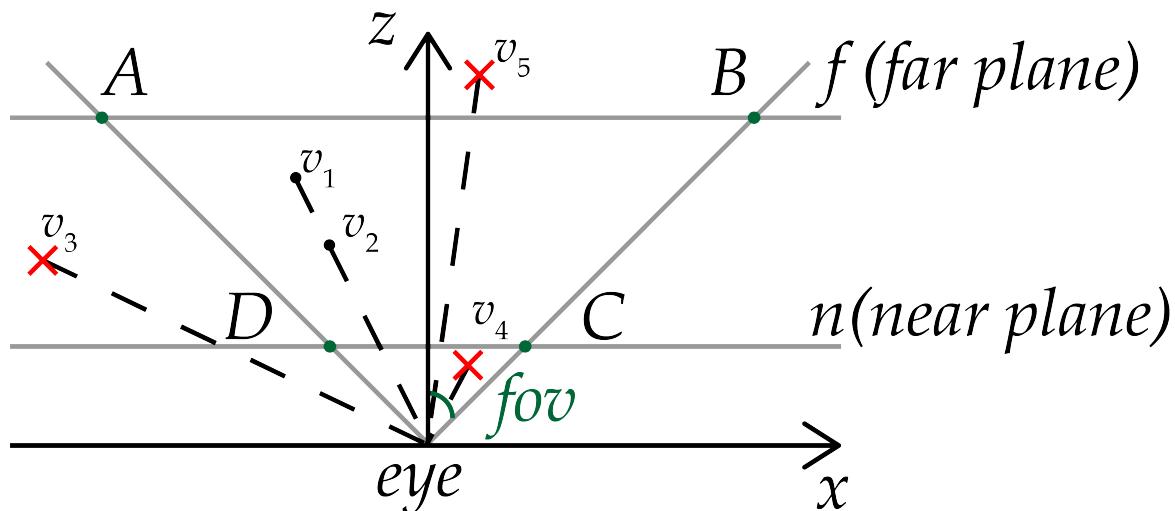


Рис. 24: Переход в однородное пространство отсечения

При получении проекции имеем дело с усечённой пирамидой (*view frustum*). В случае двух осей  $x$  и  $z$  должны отрисовать трапецию  $ABCD$ . На экране будут отображены только лежащие внутри неё точки, то есть только такие, что находятся внутри угла  $2 * \text{fov}$  и имеют координату  $z$  такую, что  $n \leq z \leq f$ . Остальные объекты должны быть отсечены. Причём, работая с проекционной геометрией, важно помнить, что, например, точки  $v_1$  и  $v_2$  будут в итоге иметь одни и те же координаты.

Для случая осей  $y$  и  $z$  рисунок такой же, но надо учитывать соотношение сторон.

Систему координат, в которую будет совершен переход, называют также однородным пространством отсечения. Поэтому четвёртая координата  $w$ , впервые введённая в прошлой главе, носит название однородной. Если до этого этапа она всегда была равна единице, то теперь с помощью её нормализации будет происходить проекция.

Плоскость проекции, то есть экран, на который проецируются точки, будет находиться на глубине  $z = 1$ . В отличие от предыдущих этапов одного применения матрицы для перехода недостаточно. Нужно будет ещё нормировать пространство.

Матрица перехода дополнительно сохранит координаты  $z$  в  $w'$ . Нормализация будет проводится с помощью деления результата на  $w' = z$ . В том, что переход осуществляется именно так, находит отражение тот факт, что координаты точек  $v_1$  и  $v_2$  на рисунке 24 получают одинаковые значения после проекции. Благодаря нормализации пространства условие принадлежности точки усечённой пирамиде можно будет записать следующим образом:

$$\begin{aligned} -1 &\leq x' \leq 1 \\ -1 &\leq y' \leq 1 \\ -1 &\leq z' \leq 1 \end{aligned}$$

Вместо постепенного вывода сначала выпишем финальную матрицу, а после обсудим её вид:

$$P = \begin{pmatrix} \operatorname{tg}(fov)^{-1} & 0 & 0 & 0 \\ 0 & \frac{\operatorname{tg}(fov)^{-1}}{ar} & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{n-f} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Очевидна последняя строчка. Как было написано выше, в координату  $w'$  в итоге будет сохранена  $z$ . Для удобства восприятия запишем результат в виде системы:

$$\begin{cases} x'z = \frac{x}{\operatorname{tg}(fov)} \\ y'z = \frac{y}{ar \cdot \operatorname{tg}(fov)} \\ z'z = \frac{f+n}{f-n} \cdot z + \frac{2fn}{n-f} \cdot w \\ w'z = z \end{cases}$$

Умножения на  $z$  в левых частях уравнений означают, что для получения проекции недостаточно применить матрицу. Нужно будет ещё и нормализовать координаты с помощью деления на  $w' = z$ .

Логично выглядит, что изменение  $x$  зависит только от самой себя. Рассмотрим рисунок:

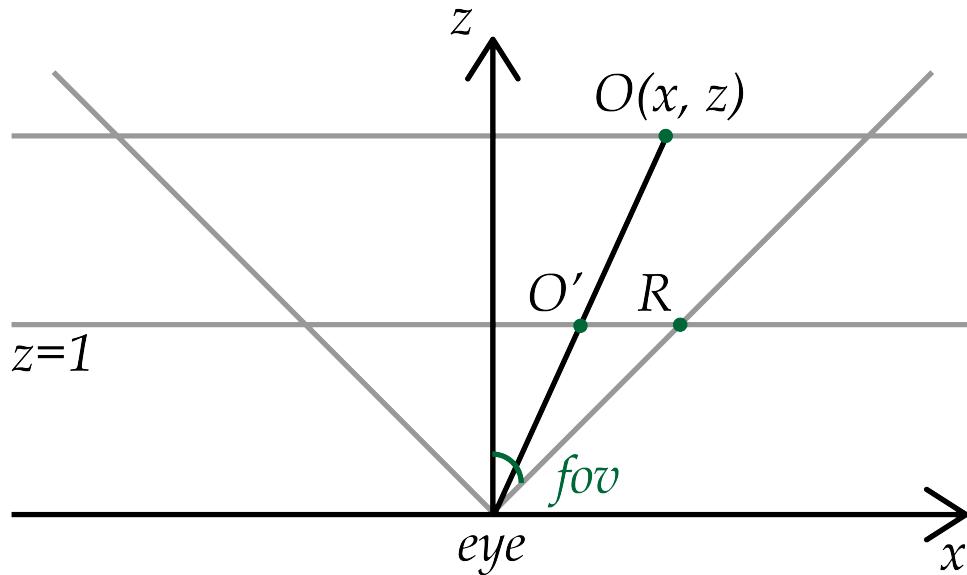


Рис. 25: Получение однородной координаты  $x'$

Точка  $O'(x', z')$  является проекцией  $O(x, z)$ . При этом в силу подобия треугольников:

$$\frac{x'}{1} = \frac{x}{z}$$

Отсюда

$$x'z = x$$

Вот только пространство, в которое совершается переход, находится в другом масштабе. Так, у точки  $R$  на рисунке координата  $x = \operatorname{tg}(fov)$ , а  $x'$  должна быть равна единице. Поэтому дополнительно необходимо поделить результат на  $\operatorname{tg}(fov)$ . Таким образом получаем выражение для  $x'$ .

Для  $y'$  рассуждения аналогичные. Только здесь результат делится ещё и на величину соотношения сторон, так как при проекции надо учесть, что экран не обязательно квадратный.

А вот для  $z'$  выражение несколько сложнее. Координаты  $x$  и  $y$  в нём по понятным причинам не участвуют. Чтобы найти элементы матрицы, с учётом  $w = 1$  составим уравнение:

$$z'z = A \cdot z + B \cdot w = A \cdot z + B$$

При переходе требуется, чтобы в точке  $z = n$  координата  $z'$  равнялась  $-1$ , а при  $z = f$  она была равна одному. Это приводит к уравнениям:

$$\begin{cases} -n = A \cdot n + B \\ f = A \cdot f + B \end{cases}$$

Решив систему, вы получите оставшиеся элементы матрицы проекции.

## Получение координат на экране

Итак, алгоритм перехода от локальных координат моделей в однородные координаты экрана выглядит так:

1. Перевести вершины моделей в четырёхмерные вектора при помощи  $w = 1$ .
2. Составить матрицы  $M$ ,  $V$ ,  $P$  и получить  $\vec{v}' = (PVM)\vec{v}$ .
3. Нормализовать  $\vec{v}'$  при помощи деления на  $w'$ .
4. Позже при отрисовке можно отбирать только те точки, для которых верно:

$$-1 \leq x' \leq 1$$

$$-1 \leq y' \leq 1$$

$$-1 \leq z' \leq 1$$

Вот только полученные однородные координаты ещё не позволяют отрисовать что-то на экране. Они хранятся в нормализованном диапазоне, чтобы после вычислений была возможность растянуть изображение под любой размер.

Для завершения обработки координат необходимо сделать переход от однородного пространства к стандартным координатам виджета, приложения или экрана (см. рисунок 26).

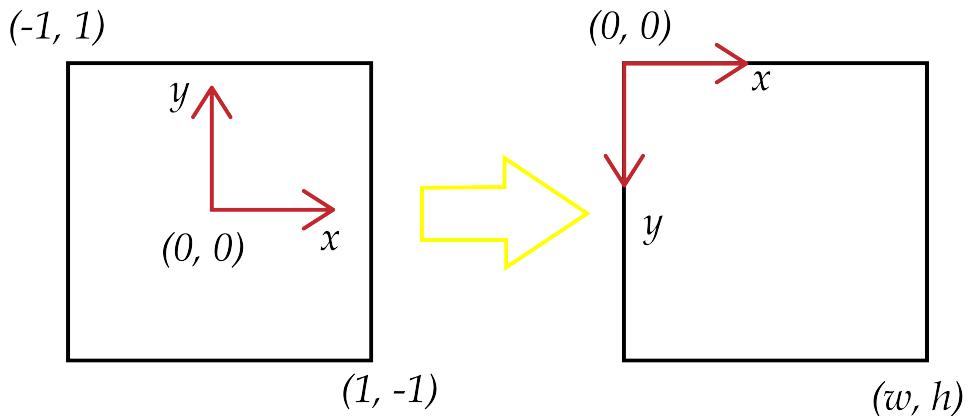


Рис. 26: Переход от однородных к экранным координатам

Сделать это можно с помощью следующих выражений:

$$\begin{cases} x' = \frac{w-1}{2} \cdot x + \frac{w-1}{2} \\ y' = \frac{1-h}{2} \cdot y + \frac{h-1}{2}, \end{cases}$$

где  $w$  — ширина экрана, а  $h$  — его высота.

При необходимости этот переход также можно записать с помощью умножения на матрицу.

## Наложение текстуры

Описываемые выше преобразования позволяют превратить трёхмерные вершины модели в двумерные координаты экрана. Если ещё и провести триангуляцию полигонов, отрисовка сведётся к разобранной во второй главе растеризации набора треугольников.

Иногда модель действительно заливают сплошным цветом. Но в ряде задач на неё необходимо надеть текстуру. Делается это в момент отрисовки полигонов.

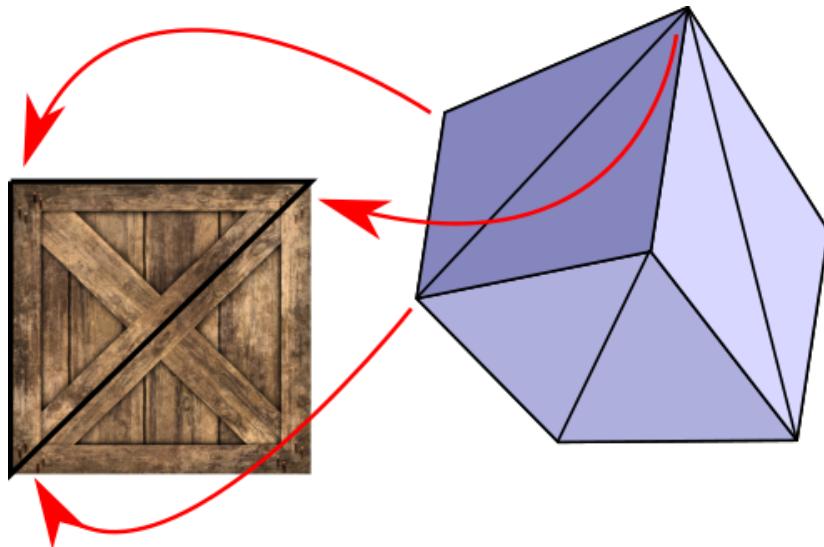


Рис. 27: Оборачивание полигона текстурой

У треугольников есть не только координаты вершин, но и текстурные координаты. Точно также, как во второй главе предлагалось интерполировать цвет между вершинами, теперь можно на основе барицентрических координат получать текстурную вершину для конкретного пикселя:

$$\vec{vt}_{pixel} = \alpha \vec{vt}_A + \beta \vec{vt}_B + \gamma \vec{vt}_C$$

То есть, слегка модифицируя алгоритм, возможно в момент закраски пикселя получать его координаты  $\alpha$ ,  $\beta$  и  $\gamma$ , а после на основе текстурных координат вершин вычислять  $uv$ -координаты для текущей точки. Цвет для закрашивания пикселя теперь можно брать с текстуры, растянув нормализованные  $\vec{vt}_{pixel}$  под размер изображения. Картинку, конечно, следует заранее загрузить в оперативную память, чтобы не тормозить программу постоянными обращениями к жёсткому диску.

Особо внимательные студенты после применения алгоритма с использованием барицентрических координат, рассчитанных уже после проекции, могут заметить неприятные артефакты при поворотах модели. Читателю предлагается найти решение этой проблемы самостоятельно.

## Простейшая модель освещения

В самом простом варианте для того, чтобы картинка выглядела более реалистичной, достаточно добавить тени. Их расчёт основан на операции скалярного произведения.

Яркость цвета в точке зависит от того, под каким углом на неё падает луч света. Если тот приходит почти перпендикулярно, то и объект яркий. В ином случае цвет следует слегка затемнить.

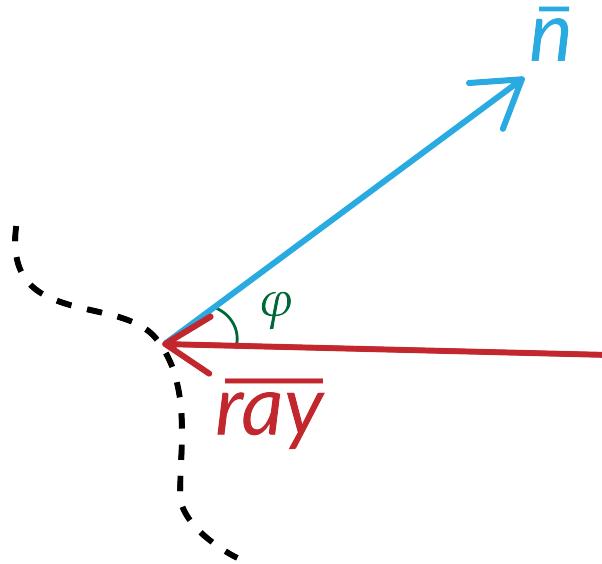


Рис. 28: Нормаль и падающий луч в точке

Для того, чтобы была возможность вычислить цвет в точке, нужно знать координаты нормали и луча света. Так как от этих векторов используются только направления, величины обычно нормализуют. Тогда становится возможным получить нормированный коэффициент:

$$l = -\vec{n} \cdot \vec{ray}$$

Если  $l$  отрицательный, это значит, что луч приходит изнутри модели. В таком случае коэффициент можно обнулить.

В итоге получаем универсальный коэффициент  $l$  в диапазоне от 0 до 1 такой, что его можно умножить на текущий цвет  $rgb$  в точке, дабы обновить тот с учётом освещения. Правда, если это сделать в лоб, переход может показаться слишком резким и неестественным. Это связано с тем, что в реальном мире независимо от конкретного источника света почти всегда есть какое-то фоновое освещение. Поэтому в нашей модели можно не доводить до ситуации, когда цвет близок к абсолютно чёрному и выбрать следующий формат:

$$\vec{rgb}' = \vec{rgb}(1 - k) + \vec{rgb} * k * l,$$

где  $k$  — тоже какой-то коэффициент от 0 до 1, настраиваемый разработчиком. Он показывает, какая доля от цвета отдается для настройки теней. Часть  $(1 - k)$ , напротив, отвечает за фоновую неизменяемую яркость.

Осталось ответить на вопрос, как же посчитать нормаль в каждой точке модели. Если брать её одну на весь полигон, то после работы алгоритма будут видны переходы между гранями. Чтобы этого избежать, реализуют так называемое **сглаживание нормалей**. В том числе ради него в модели хранятся нормали не для полигонов, а для вершин.

Опять же, с помощью барицентрических координат в момент закрашивания пикселя на треугольнике, можно на основе нормалей вершин получить нормаль в точке:

$$vn_{pixel}^{\rightarrow} = \alpha v\vec{n}_A + \beta v\vec{n}_B + \gamma v\vec{n}_C$$

А затем применить её для вычисления яркости.

В случаях, когда модель может менять своё положение или совершать повороты, необходимо переводить нормали из локальной системы координат в мировую. Иначе можно получить странную ситуацию, когда движение на сцене не влияет на освещение.

*P.s.* А как в реальном мире образуются блики на предметах? И что следует добавить в описанный алгоритм для того, чтобы их было возможно отрисовать.

## Алгоритм Z-буфера

Отображение трёхмерного пространства на плоскости приводит к тому, что далёкие и близкие объекты могут оказаться на проекции рядом. Более того, если реализовать что-то неправильно, то первые смогут перекрывать собой вторые. Например, затылок человека может вылезти на лоб. Поэтому необходимо добавить некий дополнительный алгоритм обработки.

Вообще, пока после всех матричных преобразований мы активно использовали только координаты  $x'$  и  $y'$ . Настало время поработать и с  $z'$ , которая хранит информацию о том, насколько далеко точки располагаются от камеры. Да, мы с помощью неё уже можем отсечь те объекты, что оказались вне усеченной пирамиды ( $-1 \leq z' \leq 1$ ). Но теперь ставится задача ещё и как-то сортировать точки внутри неё.

Для этих целей разработан алгоритм Z-буфера. Вместе с экраном, на котором идёт отрисовка, вводится вещественная матрица того же размера, по умолчанию заполненная очень большими значениями. Далее алгоритм похож на поиск минимума в массиве. В момент закрашивания точки смотрим в буфер. Если значение по соответствующим координатам больше  $z'$ , значит пиксель нужно отрисовать и обновить буферную матрицу. А если в ней уже стоит отметка о более близкой точке, то пиксель рисовать не стоит. Схематично это можно изобразить так:

```
1 if (z_buffer(x, y) <= z)
2     continue;
3
4 put_pixel(x, y);
5 z_buffer(x, y) = z;
```

Осталось ответить на вопрос, как же получить координату  $z'$  для каждой точки на полигоне. Ведь описанные выше матричные преобразования выполнялись только для вершин модели.

Ответ тот же, что и раньше — барицентрическая система координат. Получив для точки на треугольнике  $\alpha$ ,  $\beta$  и  $\gamma$ , вы сможете интерполировать и координату глубины:

$$z'_{pixel} = \alpha z'_A + \beta z'_B + \gamma z'_C$$

## Вспомогательный проект

Чтобы немножко упростить жизнь, предлагается начать работу с уже работающего, но ещё далеко не идеального проекта:

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task4/Simple3DViewer>

Это оконное приложение на JavaFX, позволяющее загрузить модель, посмотреть на её полигональную сетку и даже немножко покрутить камеру.

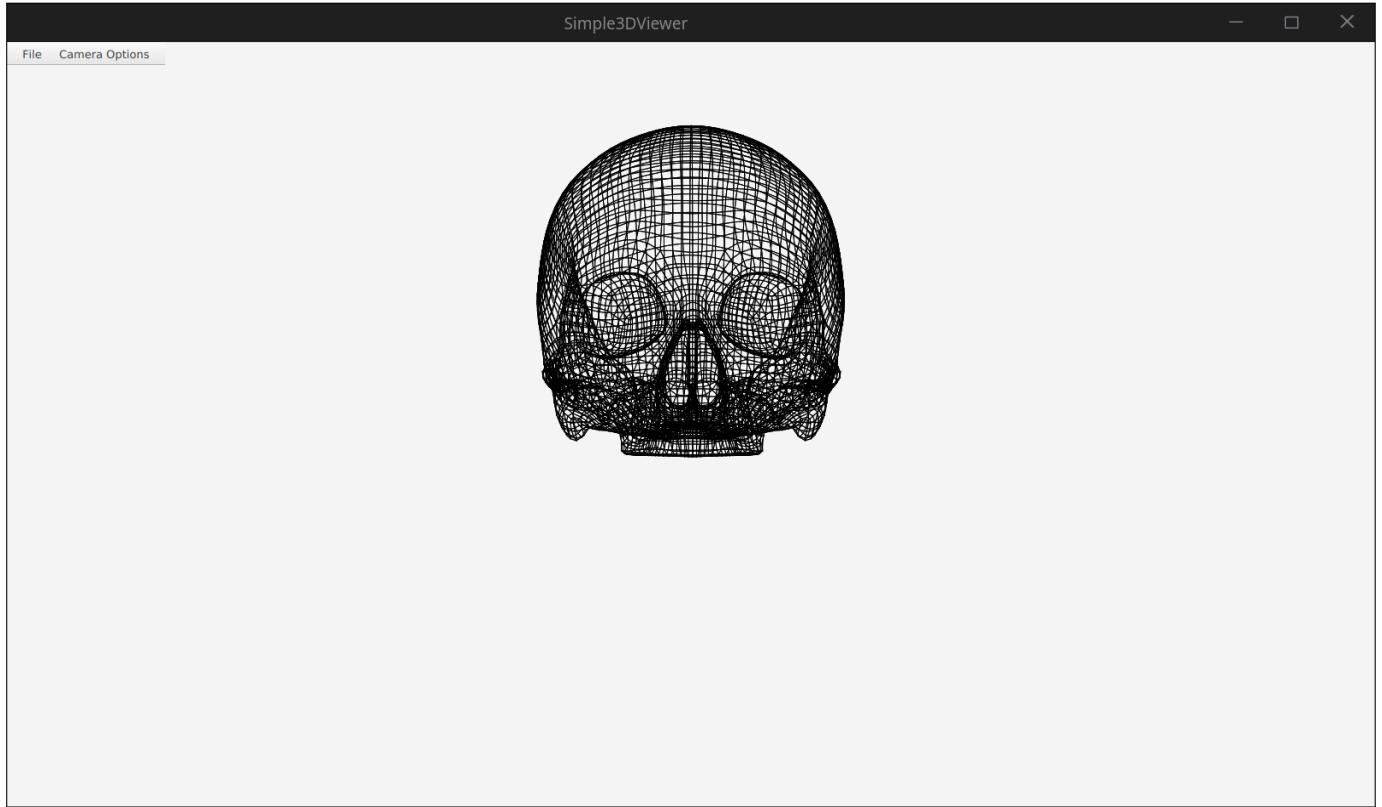


Рис. 29: Проект Simple3DViewer

Бегло пробежимся по структуре проекта, чтобы дальше было проще разобрать его самостоятельно.

1. Папки math, model и objreader полностью перекочевали из проекта для предыдущей главы. Это заготовки под библиотеку для работы с математикой, классы хранения модели и её чтения из файла.
2. Новые объекты лежат в пакете render\_engine. Это, например, класс камеры, хранящий поля, необходимые для создания видовой и проекционной матрицы.
3. Рядом есть класс GraphicConveyor, в котором статическими методами написаны создания некоторых матриц. Есть lookAt() и perspective() для перехода в пространство камеры и проекции соответственно. Матрица модели, возвращаемая методом rotateScaleTranslate(), пока единичная. Это потому, что классические аффинные преобразования над нею пока не реализованы.

Все матрицы заданы для случая векторов-строк, то есть представлены не в том виде, что были описаны выше. Используется библиотека `javax.vecmath.*`. Поскольку в ней ведётся работа с векторами-столбцами, добавлен костыль `multiplyMatrix4ByVector3()`, делающий умножение матрицы на вектор. Конечно, в вашем финальном варианте такой гадости быть не должно.

4. В классе `RenderEngine` происходит отрисовка модели. Необходимые матрицы перемножаются и вычисляются новые координаты вершин. Пока добавлен самый примитивный способ отрисовки, соединяющий точки полигона линиями.
5. В `Simple3DViewer` и `GuiController` стандартно описаны виджеты приложения и взаимодействие с ними. Даже на таком простом участке кода уже допущены некоторые ошибки проектирования программы.

Ваше четвёртое задание будет отталкиваться от предложенного проекта. Оно заключается в улучшении приложения, добавлении новых возможностей.

Подробности описаны ниже, но глобально идея следующая: объединить почти всё, что успели обсудить ранее: алгоритмы, математику, программирование и подходы к разработке. Постарайтесь подойти к задаче серьезно и выполнить её так, чтобы вашим проектом можно было гордиться.

## Варианты заданий

Это финальное задание курса и на него уйдет больше всего времени. Задача выполняется в группах максимум из трёх человек, но каждый отвечает и оценивается только по своим пунктам. Если нет возможности сформировать группу из трёх человек, возможно взяться за задачу вдвоём и реализовать её две трети. В совсем крайних случаях отстающий студент может взять себе только одну часть задачи и сдавать проект в одиночку. Однако это всё нежелательные ситуации, потому что одной из целей задания является тренировка работы в команде.

Не забывайте, что необходимым условием для разработки здесь является Git. Постарайтесь организовать работу грамотно: с разведением программистов по веткам, частыми коммитами и слияниями (или даже pull request'ами).

Текущие задачи опираются на код, который вы писали в предыдущих заданиях. Предполагается, что здесь будет использоваться результат труда чужих студентов. Но в первую очередь берите код, который писали сами или который писали другие участники вашего финального проекта. Это позволит быстрее отлаживать проблемы, и в случае багов человек, которому следует настучать по шапке, будет в шаговой доступности. Но при этом вряд ли вы втроём реализовали все задачи курса, так что контактировать с другими рабочими группами придётся. На лекциях мы организуем таблицы для обмена.

Продолжайте там, где это возможно, покрывать код юнит-тестами. Если берёте чужой модуль, желательно перетащить себе и его тесты.

Берите за основу описанный выше вспомогательный проект. И разберитесь, что будут делать ваши коллеги. Хорошая слаженная организация работы — это уже половина выполненной задачи.

## **Задания для первого человека.**

Основная сфера деятельности этого студента - наведение внешней красоты и достижения удобства работы с софтом. Программу встречают по одежке, так что та должна быть приличной.

1. *Загрузка и чтение моделей.* Вместо сырого ObjReader должна быть качественная протестированная реализация: ваша или другого студента, если вы ее не делали в третьей задаче. А также должна быть возможность сохранения модели с помощью ObjWriter по кнопке в меню.
2. *Сцена.* Сейчас в программе только одна модель. Нужно добавить работу с несколькими. При этом должна остаться возможность перемещать, вращать и как-то изменять (см. работу другого студента) только одну из них. Продумайте, как можно пользователю выбирать, какая/какие (множественный вариант сложнее в реализации, но интереснее) из моделей сейчас активные, т.е выбраны для применения разных трансформаций, сохранения и т.д. В этом пункте очень важен выбор грамотной архитектуры внутри приложения. Можно обсудить её с товарищами.
3. *Удаление части модели.* На основе кода других студентов добавьте возможность удалять вершины и полигоны внутри программы. Интерфейс продумайте самостоятельно.
4. *Интерфейс.* Эти пункты, а также то, что выполняют другие члены команды, обязывает развивать интерфейс программы. В вашу зону ответственности входит написание его таким, чтобы с ним было удобно работать. А еще хорошо бы продумать стиль приложения: добавить цвета, оформление. При желании можно сделать динамическое переключение тем: например, "светлая/темная".
5. *Обработка ошибок.* Пока это касается только ObjReader, но вы можете продумать и другие случаи. Модуль выплёвывает exception'ы, если приходит некорректный файл. Эти исключения пока никак не используются. А нужно выводить окошко с ошибкой, чтобы пользователь мог обдумать свое поведение и щелкнуть "ок", а не получить зависание или падение программы.
6. *Деплой.* Провести финальную работу по деплою приложения. При последней сдаче его следует запускать не из среды разработки, а как самостоятельное портативное приложение.

## Задания для второго человека.

Основное занятие этого студента - математическая база для графического движка. На нём реализация всех матричных преобразований.

1. *Модуль для математики.* В программе в качестве библиотеки для работы с линейной алгеброй используется `javax.vecmath`. А нужно использовать свой модуль, либо то, что реализовал ваш однокурсник, если у вас не было такого варианта в третьей задаче. То есть нужно переделать всю работу с математикой на наш собственный пакет `Math`.
2. *Вектора-столбцы.* На данный момент во всех матричных преобразованиях в коде предполагается, что мы работаем с векторами-строками. Необходимо переделать методы так, чтобы векторы были столбцами. Помните, что вам придётся транспонировать матрицы и поменять порядок их перемножения.
3. *Аффинные преобразования.* В программе реализована только часть графического конвейера. Нет перегонки из локальных координат в мировые координаты сцены. Вам нужно реализовать её, то есть добавить аффинные преобразования: масштабирование, вращение, перенос. Можете использовать наработки студентов из предыдущей задачи. И не забудьте про тесты, без них визуально может быть сложно отследить баги.
4. *Трансформация модели.* После реализации всего конвейера, нужно добавить в меню настройку модели. Необходима возможность масштабировать ее вдоль каждой из осей, вокруг каждой из осей поворачивать и перемещать. При сохранении модели (см. работу другого студента) следует выбирать, учитывать трансформации модели или нет. То есть нужна возможность сохранить как исходную модель, так и модель после преобразований. Посоветуйтесь с человеком, отвечающим за интерфейс, он может выделить вам место под нужные кнопки.
5. *Управление камерой.* Сейчас взаимодействие с камерой не очень удобное, используется только клавиатура. Но можно переделать его, добавив в систему мышь. За основу можете взять управление из компьютерной игры или приложения для работы с трехмерной графикой. Здесь хорошо бы продумать горячие клавиши и заодно упростить управление моделями.

## Задания для третьего человека.

Третий студент занимается режимами отрисовки: добавлением текстуры, освещения. Ему будет больше всего от того, что код выполняется медленно.

1. *Подготовка к отрисовке.* Добавить возможность триангуляции модели и вычисления её нормалей после загрузки. Можно использовать код однокурсников из третьей задачи. Нормали следует пересчитывать даже если те сохранены в файле, потому что мы не можем им доверять.
2. *Растеризация полигонов.* Добавить заполнение полигонов одним цветом, используя растеризацию треугольников, написанную студентом во втором задании. Учтите, что вам потребуется алгоритм Z-буфера, чтобы прирендере задние полигоны не вылезали на передние.
3. *Текстура и освещение.* Используя уже написанный метод, можно легко дополнить программу наложением текстуры (её как картинку следует подгружать в меню). А после следует реализовать простейшую модель освещения. Источник освещения можно привязывать к текущей камере. Используя сглаживание нормалей, вы сможете мягко опоясывать объекты светом.
4. *Режимы отрисовки.* Когда все описанные выше пункты будут готовы, надо добавить в программу возможность переключения между режимами. Чтобы легко перебирать все возможные случаи, предлагается оформление в виде галочек (ваши названия могут отличаться):

- Рисовать полигональную сетку
- Использовать текстуру
- Использовать освещение

Так, например, если не выбрана ни одна галочка, модель окрашивается статическим цветом, выбор которого, кстати, можно вынести в меню. В этом режиме для оптимизации можно вызывать не наш метод отрисовки треугольника, а библиотечный. Если добавляется галочка *использовать освещение*, цвет по модели становится то ярче, то темнее. При добавлении *использовать текстуру* он подменяется на те значения пикселей, что были на текстуре. А галочка *рисовать полигональную сетку* нарисует поверх всего контуры полигонов. Тут необязательно, но тоже было бы хорошо добавить Z-буффер (для этого придется вставить самостоятельную растеризацию прямых).

5. *Несколько камер.* Нужно добавить поддержку нескольких камер в сцене. Задание похоже на добавление нескольких моделей, так что имеет смысл обсудить архитектуру приложения с однокурсниками. Должна быть возможность создавать, удалять камеры, а также переключаться между ними. Сами камеры тоже могут быть видны на сцене в виде трехмерных моделей.

Если подходить к проекту творчески, у вас получится своё уникальное приложение. Рисунок ниже показывает фрагмент того, как это выглядело у одной из команд прошлых лет. По странному стечению обстоятельств он также иллюстрирует эмоциональное состояние тех студентов, которые приступают к выполнению работ в последние сроки.

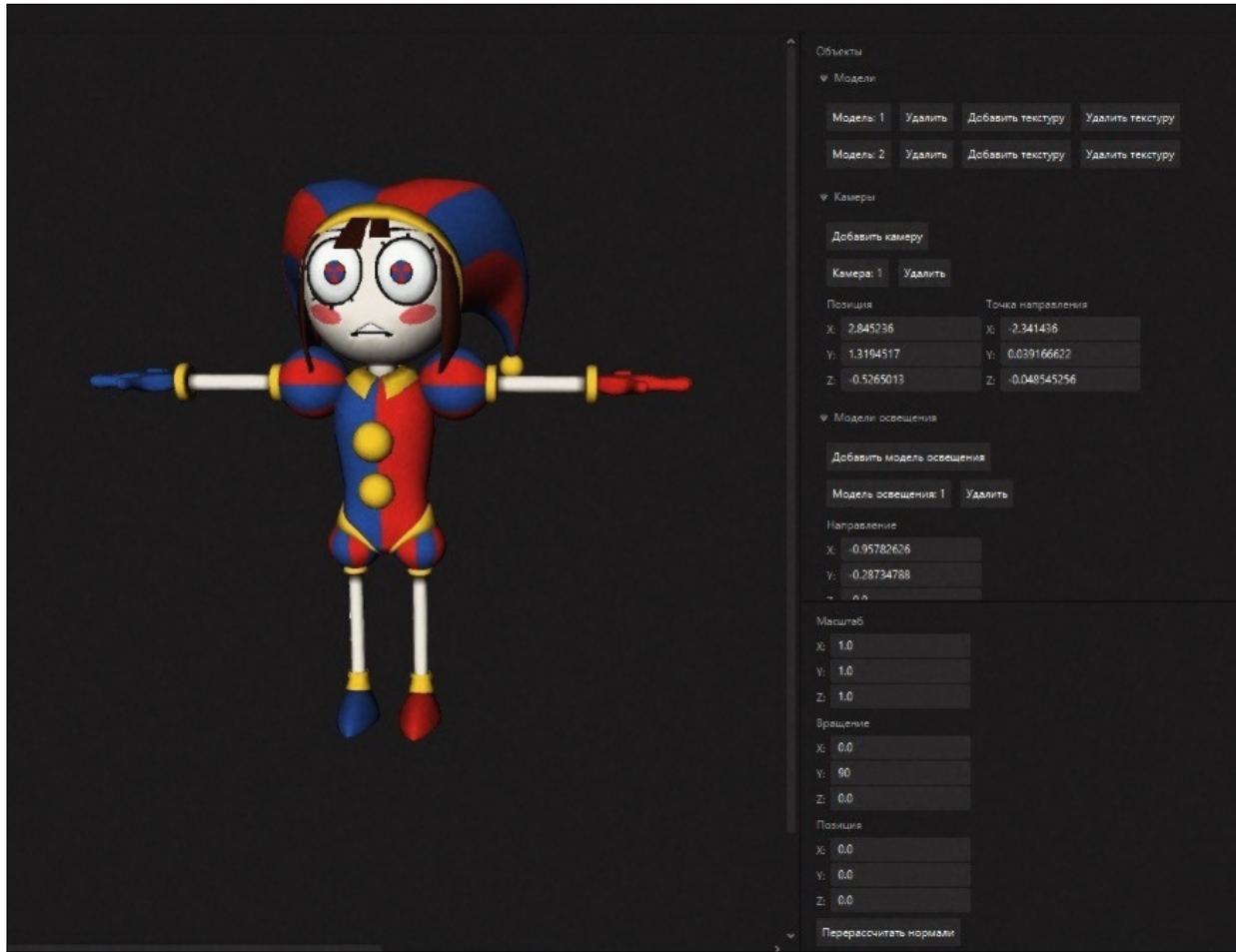


Рис. 30: Кусочек финального проекта одной из команд

## 5 Быстрее, выше, сильнее!

Пятая задача не является обязательной для сдачи курса. Студенты вполне могут получить оценку *отлично*, справившись со всеми предыдущими заданиями, контрольными работами и прочими тестами.

Но иногда находятся те, кому мало данного материала. Действительно, курс носит вводный характер, может захотеться более глубого погружения или хотя бы возможности применить полученные знания при разработке какого-то более серьёзного проекта.

Для таких желающих как раз это задание. Чтобы получить его, необходимо закрыть всё, что нужно для сдачи курса, и обсудить требования с лектором. Именно им принимается решение о выдаче задачи, а также её последующее сопровождение и оценка.

Выполнять задание также можно в группах. Сдать проект можно даже после завершения семестра.

### Возможные задачи

1. Оптимизация проекта, созданного в 4-й задаче. Более осознанная разработка, тестирование, профилирование. Возможно использование других языков программирования и OpenGL.
2. Реализация на базе проекта собственной 3D игры. Задача может превратиться в разработку собственного игрового движка.
3. Написание своей игры на базе готового движка (Unreal Engine, Unity, Godot и пр.)
4. Реализация алгоритма, предложенного лектором, для задач компьютерной графики. Самостоятельное чтение книг, статей и прочих источников — для тех, кто хочет прикоснуться к научной среде.
5. Разработка приложений с использованием готовых моделей машинного обучения. Самостоятельное их обучение.
6. Любая другая задача, связанная с компьютерной графикой и одобренная преподавателем по теории.

# Приложение 1. Работа с Git через консоль

Текст ниже — попытка сэкономить время, проводимое в формулировке ответов на однотипные вопросы.

Так выходит, что большинству новичков сложно разобраться во всех возможностях, которые предоставляет Git. Несмотря на относительную простоту технологии и наличие большого количества документаций, когда перед студентом, почти не работавшим с системами контроля версий, ставится задача создать коммит или залить на сервер свои изменения, он, как правило, впадает в ступор.

При этом есть и другой класс возможных ситуаций: когда студент узнает, как коммитить через IDE или какой-то из сотни GUI-клиентов и начинает без разбора жать на одну и ту же кнопочку. Коммиты сыплются в репозиторий надежно как швейцарские часы: зачастую с одним и тем же названием по умолчанию, изменениями, которые тянуть не следовало, конфликтами и странными переплетениями веток. Пользователь рад и доволен, ведь он думает, что умеет работать с Git. Но стоит ему серьезно накосячить или столкнуться с задачей, для которой готовой кнопки нет, начинаются проблемы.

Мы ничего не имеем против оконных приложений и систем, встроенных в среды разработки. Это по-своему удобные инструменты, часто упрощающие жизнь программиста. Но речь о том, что их стоит использовать, понимая, какие операции на самом деле выполняются. К тому же знание того, как работать с Git через консоль, сделает вас универсальным бойцом, непривязанным к конкретному приложению.

Несмотря на кучу статей и обучающих видео, знакомство с любой технологией становится проще, если рядом садится знакомый человек и рассказывает, что и как делать. Этот текст ставит себе целью частично заменить такого человека и попытаться за раз перечислить команды, которые на практике закроют подавляющее большинство возможных ситуаций.

## Что такое Git?

Для более полной картины приведем описание Git для тех, кто уж совсем не успел с ним познакомиться. Официальный сайт дает следующее определение:

*Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.*

Итак, основное здесь то, что Git - система контроля версий. Это инструмент, позволяющий не просто хранить свои данные, но вместе с ними держать в памяти их историю изменений, откатываться к предыдущим версиям, организовывать работу в команде, в том числе одновременную и над общими модулями; отслеживать, кто и когда внес соответствующую правку, и много чего еще. При наличии сервиса для хостинга (GitHub, GitLab и пр.) Git также упрощает работу по хра-

нению своих файлов в облаке, а не на локальной машине. В той или иной мере мы коснемся всех перечисленных возможностей.

## Куда вбивать команды?

Раз уж собирались работать через консоль, для начала нужно её найти. Для Windows можно скачать себе Git BASH с официального сайта:

<https://git-scm.com/downloads>

Это минималистичный bash-эмулятор для работы с Git через терминал, крайне удобная вещь.

В свежих версиях Windows Git вообще может быть предустановлен в стандартной консоли.

Если есть желание не вылезать из уютной IDE, можно вспомнить, что в большинство современных сред встроен терминал, где Git уже должен быть. Да и почти в любую другую консоль он ставится соответствующей командой.

На том же сайте можно найти инструкцию и для других ОС. Например, в популярные дистрибутивы Linux Debian/Ubuntu Git ставится так:

```
sudo apt-get install git
```

А для macOS поможет Homebrew:

```
brew install git
```

## С чего начать работу с репозиторием?

Этап с авторизацией сейчас пропустим. Во-первых, когда она потребуется, Git сам напишет, что ему нужно сделать. Обычно ничего сложного, в крайнем случае можно забить его сообщения в поиск по сети. Во-вторых, эта часть с новыми обновлениями периодически меняется, нет смысла делать инструкцию на несколько лет вперед. На момент написания текста, например, при работе через Git BASH с GitHub-репозиторием приложение само подсоединится к браузеру, чтобы использовать существующий логин. А под линуксом в последний раз Git попросил указать свои логин и почту:

```
*** Please tell me who you are.  
Run  
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"
```

Репозиторий — это и есть ваши данные плюс история их изменений. Отличить репозиторий от обычной директории можно, например, по присутствию внутри папки .git и некоторых других специальных файлов (проводники, как правило, по умолчанию их скрывают).

Превратить директорию в Git-репозиторий можно через:

```
git init
```

Но при работе в команде и с использованием какого-нибудь веб-сервиса для хостинга, например, GitHub, можно создать репозиторий сразу на сайте, а после клонировать его к каждому из участников проекта.

Так, с использованием HTTPS, GitHub предлагает:

```
git clone https://github.com/account/repository_name.git
```

Есть также и другие способы, например, через SSH. Обо всем этом есть информация на GitHub-странице репозитория (всплывающий блок по зеленой кнопке *Code* на момент написания текста). Но обратите внимание, что если скачать оттуда код zip-архивом, вы не сможете создавать и отправлять на сервер свои коммиты, влиять новые изменения. Дело в том, что так архив качается без тех самых метаданных (папки .git и пр.), которые делают директорию репозиторием.

## Как делать коммиты?

Это все была прелюдия, теперь же переходим к самому интересному. Разберемся, как делать коммиты — блоки, фиксирующие минимальные изменения в репозитории.

Предположим, в ходе работы были созданы или изменены два файла: **header.h** и **main.cpp**.

Поставим себе задачу внести в следующий коммит изменения в первом файле, но при этом пока не трогать **main.cpp**. Правилом хорошего тона в общем случае будет сперва вызвать команду:

```
git status
```

Она покажет, что успело измениться в репозитории со времени последнего коммита. В нашем случае вывод будет примерно такой:

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be  
committed)  
header.h  
main.cpp
```

```
nothing added to commit but untracked files present (use "  
git add" to track)
```

Вывод сообщает, что в репозитории были изменены два файла, но ни один из них не прикреплен к будущему коммиту. Сделать это можно с помощью команды:

```
git add filename
```

Вместо **filename** в нашем случае будет **header.h** - название файла относительно текущей рабочей директории. Если нужно добавить все файлы внутри папки, можно указать в **filename** путь к ней, тогда рекурсивно добавится всё её содержимое. Например, находясь в корне репозитория, можно написать:

```
git add .
```

Это прикрепит к следующему коммиту вообще все, что было изменено.

А для нашей задачи после двух команд можно получить следующий результат:

```
git add header.h  
git status
```

...

```
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
  new file:   header.h
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be  
  committed)  
  main.cpp
```

Вывод говорит, что в репозитории со времени последнего коммита изменены два файла. Причем файл **header.h** будет прикреплен к следующему коммиту.

Сделать коммит можно с помощью:

```
git commit -m "commit name"
```

Здесь **-m** — аргумент, показывающий, что следующая строка — это название коммита. Например, в нашем случае там можно написать “Change header.h”, но вообще желательно делать названия как можно более информативными, чтобы позже вы или другие члены команды могли найти в списке коммитов то, что нужно.

Совершим коммит и посмотрим, что теперь скажет уже известная команда:

```
git status
```

...

```
Untracked files:  
(use "git add <file>..." to include in what will be  
  committed)  
  main.cpp
```

Отлично, теперь нам сообщается, что со времени последнего коммита, который был сделан только что, в репозитории только один измененный файл. Так мы закоммитили **header.h**.

По большому счету, разобранные команды: *git add*, прикрепляющая изменения к коммитам, *git commit*, совершающая коммиты, а также *git push*, отправляющая коммиты на сервер (о ней речь пойдет ниже) — это около 80% работы рядового программиста.

Если ставится задача создать коммиты со всеми изменениями, что есть в репозитории на данный момент, можно вместо связки команд в корневой директории:

```
git add .
git commit -m "commit name"
```

использовать более короткую равноценную форму записи:

```
git commit -am "commit name"
```

Но будьте, пожалуйста, осторожны со слепыми коммитами всего подряд. Ситуация, когда в коммит попадают лишние изменения (или не попадают нужные), крайне неприятная, встречается и у опытных пользователей. Поэтому неслучайно в примере выше делался акцент на *git status*, с помощью которой можно отслеживать, что вообще происходит.

## Как залить коммиты на сервер?

Для этих целей необходима уже упомянутая *git push*. Предположим, мы работаем в ветке *main*, сделали какое-то количество коммитов. Ставится задача залить их на GitHub, в ту версию ветки, что хранится на сервере. Тогда команда будет выглядеть следующим образом:

```
git push origin main
```

Вместо *main* в общем случае может быть любое название. Про ветки и работу с ними подробнее расскажем ниже.

Ключевое слово *origin* — это флаг, помечающий, что ветка хранится на сервере, ведь именно туда мы отправляем коммиты. Для Git понятие *branch* — локальное, а вот *origin branch* — уже удаленное.

В отдельных ситуациях может быть достаточно и

```
git push
```

Возможно, Git попросит пароль или GitHub-токен. Как и в случае с авторизацией, этот момент не будем описывать по причине: легко ищется и периодически меняется.

Вот, собственно, и все. Но поговорим о возможных проблемах.

Достаточно рядовая ситуация, когда за время вашей работы ветка на сервере успела обновиться. Поскольку коммиты хранятся в стеке строго друг за другом, *git push* не сможет встроить изменения и попросит сперва обновить локальную ветку, привести ее в согласованное с сервером состояние. Тут поможет команда *git pull*. О ней также поговорим ниже.

Еще один возможный случай, как правило, являющийся признаком неправильной организации работы с Git — это когда локальный стек коммитов противоречит стеку удаленной ветки. Тогда *git push* не пройдет и предложит вызвать себя с дополнительным флагом.

```
git push origin branchname -f
```

Да, флаг *-f* или *-force* позволит выполнить *git push* с силой и так решить подобную задачу. Если подобная ситуация произошла, Git, документация и ответы на форумах всё объяснят, но хочется акцентировать внимание, что *-force* перезапишет ваши и чужие коммиты на сервере, что может быть не очень безопасно. Так что решить эту проблему желательно крайне аккуратно, возможно, с привлечением более опытного товарища. Также есть чуть более безопасный флаг *-force-with-lease*.

## Как работать с ветками?

Поговорим о ветках.

Это инструмент для организации работы при наличии нескольких разработчиков или просто нескольких задач. Если с точки зрения конкретной ветки репозиторий выглядит как стек коммитов, то на уровень выше это дерево, а если точнее — граф. Корень графа — самый первый коммит самой первой ветки, а дальше в одну или несколько сторон идут ветви, на которые как на шампуры насажены коммиты. Ветки могут создаваться, удаляться, влиять друг в друга и снова расходится.

Картинка ниже может объяснить, о чем идет речь:

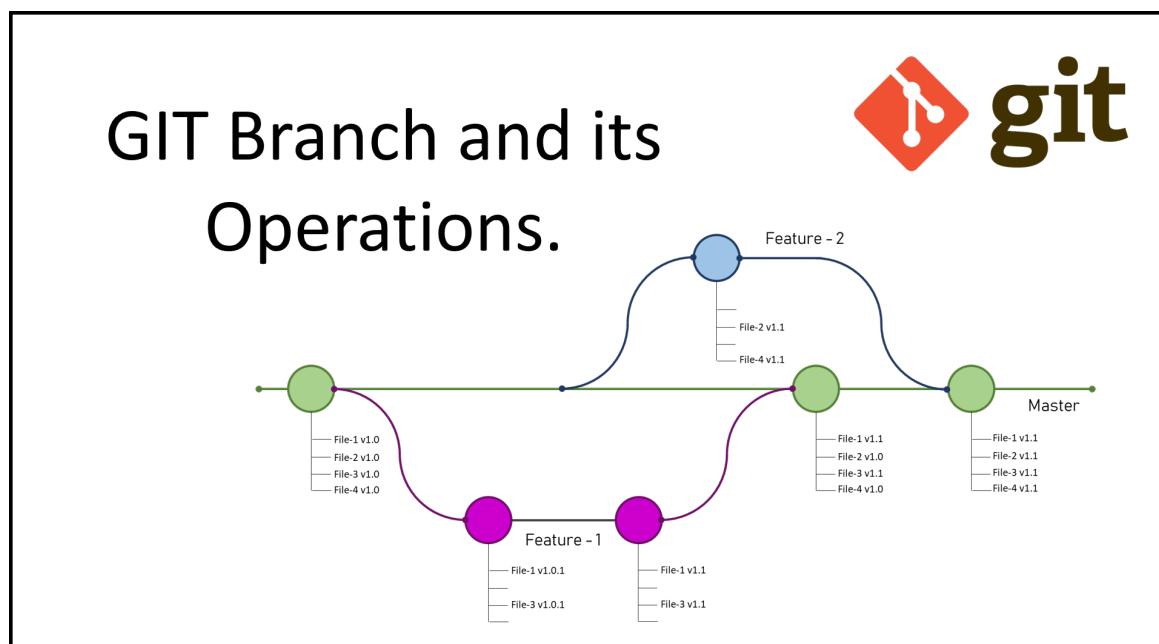


Рис. 31: Работа с ветками Git

Когда разработчику необходимо поработать над новой задачей и при этом никому и ничему не мешать, он может отпочковаться от той ветки, где находится сейчас и уйти в новое место. Делается это, например, с помощью команды:

```
git checkout -b "NewBranch"
```

```
...
Switched to a new branch 'NewBranch'
```

Флаг **-b** означает, что мы создаем новую ветку. 'NewBranch' — ее уникальное название, но как и в случае с коммитами, лучше давать максимально конкретные, понятные и читаемые имена.

Кстати, помимо того, что некоторые оболочки по умолчанию пишут название текущей ветки, посмотреть, где ты находишься, можно еще и с помощью уже знакомой *git status*:

```
git status
```

```
...
On branch NewBranch
nothing to commit, working tree clean
```

Если, поработав немного, нужно сменить ветку, это можно сделать с помощью:

```
git checkout branchname
```

Вместо **branchname** следует написать ветку назначения. Например, если там будет `main`, то вывод команды будет следующим:

```
git checkout main
```

```
...
Switched to branch 'main'
```

И последняя ключевая операция при работе с ветками — *git merge* — слияние одной в другую. Например, программист внес необходимые правки и, завершая работу, хочет добавить свои изменения ко всем. Одно из возможных решений — находясь в той ветке, куда необходимо влить коммиты, выполнить команду:

```
git merge branchname
```

Здесь **branchname** — название той ветки, которую мысливаем.

Описанных команд хватит для выполнения практически всех задач. Можно было бы добавить еще, например, про удаление ненужных веток, но как раз вот такие вещи лично по нашему мнению удобнее и безопаснее делать через веб-интерфейсы вроде <https://github.com> или какие-нибудь оконные приложения, где граф веток наглядно представлен.

Как и раньше, здесь возможны ситуации, когда какую-то из команд не получится выполнить. И как и раньше, Git напишет что-нибудь, что с помощью подсказок в интернете позволит разобраться в проблеме.

Например, может не получиться перейти в другую ветку, потому что в текущей есть несохраненные изменения. Их нужно закоммитить, удалить или сохранить во временное хранилище (`stash`, рассмотрим ниже).

Или при слиянии одной ветки в другую Git может не разрешить изменения от разных разработчиков, те будут противоречить друг другу. Тогда он сообщит о конфликтах и укажет файлы, которые придется привести в порядок ручками.

## Как влить себе чужие коммиты?

Очень частая ситуация — за время вашей работы другие разработчики успели залить на сервер кучу своих правок. Ставится задача влить их изменения себе для возможности дальнейшей работы. Сделать это можно, например, сперва обновив локальный репозиторий с помощью:

```
git fetch
```

А после совершив `git merge` необходимой ветки.

Но есть команда, которая выполнит обе эти операции за раз — `git pull`. Находясь в локальной ветке назначения, выполните:

```
git pull origin branchname
```

Эта операция обновит локальный репозиторий, чтобы тот соответствовал удаленному, а в текущую ветку вольёт изменения из **branchname**. Обратите внимание, поскольку здесь происходит `git merge`, в качестве удаленной ветки можно указать любую, а не только ту, в чьей локальной версии мы сейчас находимся.

В отдельных ситуациях достаточно просто

```
git pull
```

## Как сохранить свои изменения, но не делать коммит?

Может произойти так, что во время работы срочно придётся перейти в другую ветку на небольшой промежуток времени, а после вернуться обратно. Изменения будут мешать друг другу, к тому же возможно, что Git и не позволит уйти с ветки из-за несохранённых файлов. Поэтому нужно будет что-то сделать со своими правками, чтобы не потерять их.

Конечно, есть вариант закоммитить. Но что, если работа еще не закончена, или проект в коммите вообще не будет собираться? Ничего смертельного, конечно, но ведь некрасиво. Еще над названием коммита думать, а тут работы порой на час.

Есть вариант сделать временный коммит, а после отменить его и перезаписать по-человечески. Но тут больше возни, к тому же в спешке можно случайно затереть что-то не то.

Для подобных случаев у Git есть механизм временного хранилища — тайничка `stash`. Это свой собственный стек изменений, никак не привязанный к конкретной ветке — он глобален для репозитория. И работает `stash` по принципу стека — положить наверх/взять сверху.

Так, в уже описанной ситуации можно поступить следующим образом. Перед уходом из ветки вызвать команду, которая занесёт в стек ваши изменения:

```
git stash
```

Не забывайте пользоваться `git status`, чтобы постоянно все контролировать — например, некоторые файлы могут не добавиться. После можно перейти в другую ветку, выполнить работу и, наконец, вернуться назад. Чтобы достать свои правки из стека, следует выполнить команду:

```
git stash pop
```

Механизм нас ни разу не подводил, но если внутри вас тоже параноик, для больших и существенных правок, возможно, лучше все же выбрать вариант с коммитом.

А еще с помощью `stash` можно перенести изменения из одной ветки в другую.

## Как откатить коммиты и несохраненные правки?

Сперва познакомимся с достаточно мощной командой. Если `git status` позволяет узнать обстановку относительно правок, внесенных со времени последнего коммита, то вот сами последние коммиты можно быстро посмотреть с помощью:

```
git log
```

Ее вывод будет выглядеть примерно так:

```
commit 1f6f5a46f533c092273ea523cd334a6a8fdb8ece (HEAD ->
  main, NewBranch)
Author: kv1tr4vn <kv1tr4vn@gmail.com>
Date:   Mon Aug 29 09:43:32 2022 +0300
```

Change header.h

```
commit 80c758fc8466740168dd9d66b342ae7d7831fab3 (origin/
  main, origin/HEAD)
Merge: 2f52463 99e94a0
Author: kv1tr4vn <kv1tr4vn@gmail.com>
Date:   Tue Aug 28 15:45:18 2022 +0300
```

```
The best commit ever!  
  
commit 99e94a047f2d0891a921660f6d4757be89ead45d  
...
```

Это стек коммитов для текущей ветки. Можно увидеть хэш коммита, автора, дату и сообщение, которое мы указывали в кавычках после *git commit -m*. А еще обратите внимание, что коммиты показывают, какой является последним для какой ветки. Например, последний коммит был создан локально и еще не запущен на GitHub. Он совпадает с последним коммитом в ветке NewBranch. Для нас он HEAD — голова стека, а для сервера, куда он еще не залит, HEAD — наш предпоследний коммит.

Итак, HEAD — короткое название для коммита, являющегося головой стека. Для указания почти всех остальных коммитов в общем случае придется использовать хэш (длинная комбинация символов после слова *commit* в выводе команды выше).

Рассмотрим несколько возможных ситуаций.

## Как отменить добавление файлов к коммиту?

Если изменения в файлах уже прикреплены к следующему коммиту, но нужно отменить действие, можно воспользоваться командой *git reset*.

Например, имеем:

```
git status  
  
...  
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
  new file:   file.txt
```

После вызова команды получим:

```
git reset HEAD  
git status  
  
...  
Untracked files:  
(use "git add <file>..." to include in what will be  
  committed)  
  file.txt
```

Если теперь совершить коммит, этот файл не прикрепится к нему.

В данном случае можно было написать и просто *git reset*. Если же нужно выцепить какой-то конкретный файл, можно написать:

```
git reset HEAD file.txt
```

## Как откатить изменения до состояния последнего коммита?

Достаточно частая ситуация. Разработчик попробовал внести какую-то правку, в надежде, что та решит его проблему. Оказалось, что дорожка неверная, а в коде теперь каша. Но можно откатиться до последнего коммита, где все еще было нормально. Для этого уже известной команде *git reset* потребуется добавить флаг:

```
git reset --hard HEAD
```

А вот для одного конкретного файла *git reset -hard* не существует из-за особенностей реализации этой команды. Но можно воспользоваться:

```
git checkout HEAD -- file.txt
```

## Как откатить изменения до состояния любого коммита?

Да также как и до этого. Но если не использовать HEAD, то в общем случае придется скопировать и вставить хэш. Например, команда ниже откатит правки до состояния указанного коммита, а все изменения, включая те, что были закоммичены после, сохранит для возможности дальнейшей работы.

```
git reset 80c758fc8466740168dd9d6ae7d7831fab3  
git status
```

...

```
Untracked files:  
(use "git add <file>..." to include in what will be  
committed)  
file.txt  
header.h
```

А версия команды с флагом *-hard* сотрет вообще все, что было после указанного коммита. Правда, иногда необходимо перевести файлы из статуса **untracked files**, чтобы их перезаписало. Например, с помощью *git add*:

```
git add .  
git reset --hard 80c758fc8466740168dd9d6ae7d7831fab3  
git status
```

...

```
nothing to commit, working tree clean
```

Учтите, что если вы таким образом удалите и перезапишите свои коммиты локально, а на сервере останутся старые версии, придется заливать их с флагом, о котором упоминалось выше:

```
git push origin branchname -f
```

## Как запретить Git видеть некоторые файлы?

С этой проблемой сталкиваются многие новички, так что не будет лишним добавить.

Часто ставится задача запретить Git отслеживать изменения в каких-то файлах. Например, среди разработки забивают рабочие папки своими данными, уникальными для конкретного разработчика. Директории, куда собирается проект, могут весить порядочно и содержать много мусора. Такое в общем случае коммитить не надо. К тому же у вас может быть собственное желание не коммитить работу в какой-то из папок. Например, она нужна для личных тестов.

Для этих целей в репозитории можно добавить фильтры в файле *.gitignore*. В нем строка за строкой выписываются в обобщенной форме имена файлов и директорий, которые нужно игнорировать. Файлик можно создавать для каждой папки, но по умолчанию лучше держать в корне репозитория. Вот один из возможных минимальных примеров его содержимого:

```
cmake-build*/
.idea/
```

В сети достаточно информации, которая поможет заполнить его конкретно под ваш случай. Проверить себя можно с помощью *git status*. Как только файлы перестали отслеживаться, хоть изменения в них есть, значит, все работает.

## Заключение

В тексте мы постарались кратко познакомить вас с тем, с чем предстоит работать. В использовании Git нет ничего сложного. Если к уже описанному материалу добавить еще немножко: некоторые альтернативные подходы, сокращения, примерное понимание того, как реализованы операции, а еще команды, которые вызываешь раз в жизни из-за какой-то поломки, то будет практически все, что знаем и мы.

Для того, чтобы освоить Git на вполне рабочем уровне, может хватить одного вечера. Для того, чтобы стать опытным пользователем - недели тесной работы.

Здесь мы попытались упростить материал, чтобы текст был легкой точкой входа, а не подробной документацией, коих итак много. К нему следует относиться как к еще одному вспомогательному источнику, но держать под рукой:

<https://git-scm.com/doc> и <https://stackoverflow.com>

## Приложение 2. Операции над векторами

Повторим некоторые базовые понятия линейной алгебры. Для задач курса потребуется знание только самых простых разделов математики. Но знание это должно быть безукоризненным: не на уровне заученного определения, а с полноценным осмыслением происходящего.

### Скалярное произведение

Скалярным произведением двух векторов называется операция

$$c = \vec{a} \cdot \vec{b}$$

такая, что

$$c = ab \cos(\phi) \quad (7)$$

Здесь  $a$  и  $b$  — длины векторов, а  $\phi$  — угол между ними. Скалярное произведение потому и называется так, что на выходе скаляр, то есть единственное число, а не вектор.

Очевидно, что

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$$

Для декартовой системы координат (случай трёхмерного пространства) верна также формула:

$$c = a_x b_x + a_y b_y + a_z b_z \quad (8)$$

Верно это также и для других размерностей. А сумма поэлементных произведений используется во многих обобщениях скалярного произведения.

Из (7) и (8) можно получить формулу для нахождения угла между векторами:

$$\phi = \arccos\left(\frac{a_x b_x + a_y b_y + a_z b_z}{ab}\right)$$

Однако использовать скалярное произведение можно и без прямой привязки к координатам. Для начала стоит напомнить, что  $b \cos(\phi)$  в формуле (7) — проекция вектора  $\vec{b}$  на  $\vec{a}$ . Аналогично,  $a \cos(\phi)$  — проекция  $\vec{a}$  на  $\vec{b}$ . Чем меньше угол между векторами, тем она больше.

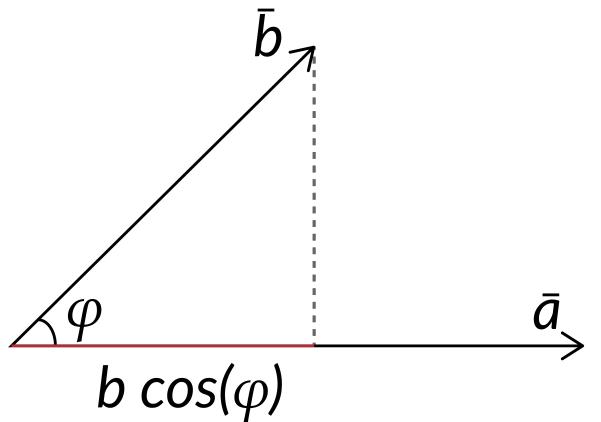


Рис. 32: Проекция вектора

Рассмотрим теперь, как будет изменяться скалярное произведение при разных углах. На рисунке (33) длины векторов  $\vec{b}$ ,  $\vec{c}$ ,  $\vec{d}$ ,  $\vec{e}$  и  $\vec{f}$  равны. Тогда

$$\vec{a} \cdot \vec{f} < \vec{a} \cdot \vec{e} < \vec{a} \cdot \vec{d} = 0 < \vec{a} \cdot \vec{c} < \vec{a} \cdot \vec{b}$$

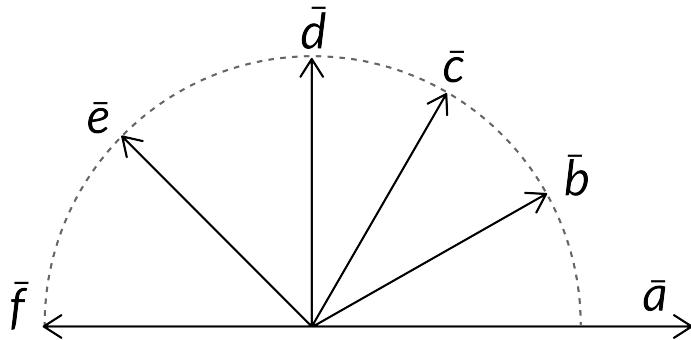


Рис. 33: Взаимное расположение векторов

Получается, что с помощью скалярного произведения можно оценить, насколько близко друг к другу направлены вектора. Это свойство выходит за рамки линейной алгебры и используется в некоторых обобщениях скалярного произведения для нахождения меры близости величин. В общем случае из (7) вытекает, что:

$$-ab \leq \vec{a} \cdot \vec{b} \leq ab$$

Левое равенство достигается, если вектора направлены в противоположные стороны, правое — если они сонаправлены. Для ортогональных векторов скалярное произведение равно нулю. Иногда для удобства величины сперва нормируют, то есть делят на свои длины. Тогда для нормализованных  $\vec{a}' = \frac{\vec{a}}{a}$  и  $\vec{b}' = \frac{\vec{b}}{b}$  с учётом

$$-1 \leq \vec{a}' \cdot \vec{b}' \leq 1$$

можно получить ёмкий коэффициент, отвечающий за взаимное расположение векторов.

В англоязычной литературе скалярное произведение называют *dot product* (от *dot* - точка).

## Векторное произведение

Векторное произведение обозначается, например, следующим образом:

$$\vec{c} = \vec{a} \times \vec{b}$$

Поскольку и на выходе тоже имеем вектор, придётся задать как его длину, так и направление. Исчерпывающее определение приведено ниже:

1.  $c = ab \sin(\phi)$
2.  $\vec{c} \perp \vec{a}, \vec{b}$
3.  $\vec{a}, \vec{b}, \vec{c}$  составляют правую тройку (в правом базисе)

Первый пункт отвечает за длину  $\vec{c}$ . Часто говорят, что векторное произведение позволяет найти площадь параллелограмма со сторонами  $\vec{a}$  и  $\vec{b}$ , т.к.  $b \sin(\phi)$  — высота, проведенная к  $a$ .

Второй пункт начинает задавать направление результирующего вектора. Он должен быть перпендикулярен как  $\vec{a}$ , так и  $\vec{b}$ . Но остается неоднозначность, поскольку перпендикулярным к двум другим векторам можно быть в две противоположные стороны. Отсюда добавляется третий пункт, что  $\vec{a}, \vec{b}$  и  $\vec{c}$  должны являться правой тройкой.

Это значит, что если расположить ладонь правой руки вдоль вектора  $\vec{a}$  так, чтобы была возможность загнуть пальцы в сторону  $\vec{b}$  по меньшему углу, то большой палец укажет направление  $\vec{c}$ .

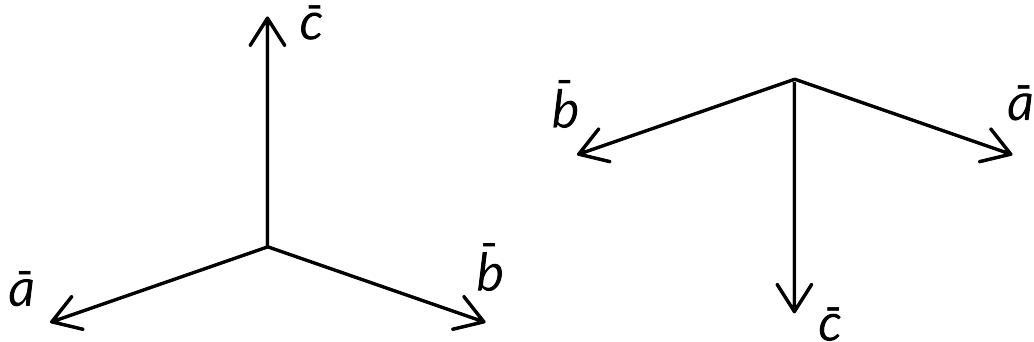


Рис. 34: Векторное произведение  $\vec{a} \times \vec{b}$

Очевидно, что

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$$

В задачах, где важно только направление векторов, величины также могут предварительно нормализовать. Результирующий  $\vec{c}$  тоже будет единичной длины.

Для часто используемой правосторонней декартовой системы координат используется формула:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = (a_y b_z - a_z b_y) \vec{i} + (a_z b_x - a_x b_z) \vec{j} + (a_x b_y - a_y b_x) \vec{k} \quad (9)$$

В случае левого базиса следует добавить минус перед определителем.

В англоязычной литературе векторное произведение называют *cross product* (от *cross* - крест).

## Вектора-строки и вектора-столбцы

Для начала, вектор — это матрица. Просто матрица, у которой количество столбцов или строк равно единице.

В первом случае говорят о векторе-столбце, а во-втором — о векторе-строке:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}, \vec{u} = (v_x \ v_y \ v_z)$$

Легко заметить, что в приведённом примере

$$\vec{u} = \vec{v}^T$$

Неважно, какая форма записи выбрана. Все возможные данные и операции над ними можно описать обоими способами. Но при переходе от одного к другому важно помнить некоторые правила.

Умножить матрицу на вектор-столбец можно так:

$$A\vec{v} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} a_{11}v_x + a_{12}v_y + a_{13}v_z \\ a_{21}v_x + a_{22}v_y + a_{23}v_z \\ a_{31}v_x + a_{32}v_y + a_{33}v_z \end{pmatrix}$$

Но в случае вектора-строки та же самая операция должна быть записана в другом порядке:

$$\vec{u}A^T = \vec{v}^T A^T = (v_x \ v_y \ v_z) \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix} = \\ (a_{11}v_x + a_{12}v_y + a_{13}v_z \ a_{21}v_x + a_{22}v_y + a_{23}v_z \ a_{31}v_x + a_{32}v_y + a_{33}v_z) = (A\vec{v})^T$$

То есть для того, чтобы обе формы записи обозначали одно и то же, пришлось воспользоваться свойством:

$$(AB)^\top = B^\top A^\top$$

Это верно и для большего числа множителей. Предлагаем взять небольшие матрицы и проверить равенство самостоятельно:

$$(ABC)^\top = C^\top B^\top A^\top$$

Так, если к вектору-столбцу  $\vec{v}$  применяют две операции, описываемые матрицами  $A$  и  $B$ , причём сначала применяется  $A$ , а потом  $B$ , то запись должна выглядеть следующим образом:

$$\vec{u} = BA\vec{v}$$

Может быть проще понять происходящее, если разбить выражение по действиям:

$$\vec{u} = B(A\vec{v}) = B\vec{v}'$$

А при переходе к вектору-строке поменяется порядок матриц:

$$\vec{u}^\top = \vec{v}^\top A^\top B^\top$$

Или

$$\vec{u}^\top = (\vec{v}^\top A^\top)B^\top = (A\vec{v})^\top B^\top = \vec{v}'^\top B^\top$$

Чувствовать подобные вещи надо хотя бы потому, что в зависимости от задачи, рабочего коллектива или фреймворка алгоритм может быть написан как на векторах-столбцах, так и на строках. К сожалению, бывает, что даже опытный разработчик не знает математику и подставляет матрицы не в том порядке, а это приводит к сложно отслеживаемым ошибкам. Поэтому некоторые задачи курса будут нацелены на закрепление описанных выше свойств.

## Приложение 3. Введение в теорию сплайн-функций

Сплайн-функции — относительно новая область математики, появившаяся в середине XX века при решении конструкторских задач с описанием сложной гладкой формы, например, частей кузова автомобиля. Впрочем, как это часто бывает, основная идея уходит корнями глубоко в прошлое. Так, например, рассматриваемый ниже интерполяционный многочлен Лагранжа тоже иногда относят к сплайнам, хотя он родом из конца XVIII века.

Сплайны — это мощный развитый инструмент для задач аппроксимации (приближения) и интерполяции. Их главная идея заключается в том, чтобы разбить исходные данные на маленькие части и на каждой из них работать с помощью какой-то локальной функции, носящей название базисной. Таким образом, сплайны — это кусочно-гладкие функции, состоящие из сегментов, чаще всего являющихся полиномами низкой степени. Эти полиномы могут строиться независимо друг от друга, но для того, чтобы сплайн был гладким, т.е. чтобы один сегмент плавно переходил в другой, проводится согласование параметров функций на соседних отрезках.

Благодаря такому разбиению появляется возможность описывать сложные формы набором простых математических конструкций. Кроме того, приходит гибкость в работе: например, можно изменить часть кривой, и это не потянет за собой всё её перестроение. Из-за таких преимуществ сплайны получили широкое применение в самых различных областях: численном анализе, статистике, инженерии, физике, компьютерной графике и т.д.

В зависимости от выбора базисной функции, а также от условий построения уже придуманы сотни, а может и тысячи видов сплайнов. Наиболее известные из них: кубические эрмитовы, кривые Безье, В-сплайны, неоднородные рациональные В-сплайны (NURBS). Бывает, что на практике при работе со сложной кривой её части описывают разными наборами функций. Есть, разумеется, и обобщения на большие размерности: поверхности, объёмы и пр. Ниже мы рассмотрим несколько простейших примеров.

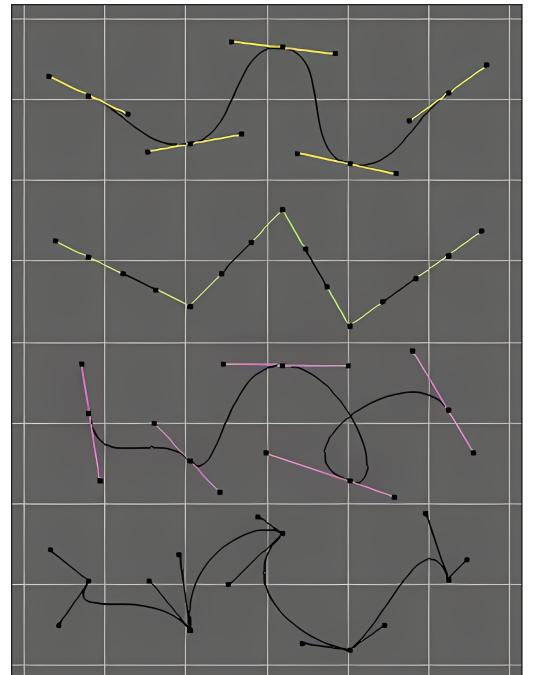


Рис. 35: Работа с кривыми в популярном бесплатном пакете Blender

## Интерполяционный многочлен Лагранжа

Относить многочлен Лагранжа к сплайнам не совсем честно. Хоть он и строится с помощью умножения на базисные полиномы, кусочно-заданной функции в итоге не выходит. Однако удобнее начинать погружение в теорию именно с таких простых конструкций.

Рассмотрим некоторую произвольную функцию  $y = f(x)$ , заданную на отрезке  $[a, b]$ . Выберем набор точек:  $a \leq x_0 < x_1 < \dots < x_n \leq b$ . Решим задачу интерполяции через построение многочлена  $L(x)$  степени  $n$ , удовлетворяющего условиям:

$$L(x_i) = f(x_i), i = 0, 1, \dots, n \quad (10)$$

Полученный полином и будет носить название интерполяционного многочлена Лагранжа.

Легко доказывается, что такой многочлен существует, и он единственный. Если  $L(x) = p_0 + p_1x + \dots + p_nx^n$ , то выражение (10) превращается в систему линейных уравнений:

$$\begin{cases} L(x_0) = p_0 + p_1x_0 + \dots + p_nx_0^n = f(x_0) \\ L(x_1) = p_0 + p_1x_1 + \dots + p_nx_1^n = f(x_1) \\ \vdots \\ L(x_n) = p_0 + p_1x_n + \dots + p_nx_n^n = f(x_n) \end{cases} \quad (11)$$

Определитель системы — так называемый определитель Вандермонда. Он часто встречается в задачах линейной алгебры, а потому хорошо изучен. Для него существует явная формула:

$$\Delta = \begin{vmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{vmatrix} = \prod_{i>j} (x_i - x_j)$$

Поскольку по условию узловые точки  $x_i$  различны, определитель не равен нулю. Система имеет решение. Так как в ней  $n+1$  уравнение с  $n+1$  неизвестными, это решение единственное.

Легко представить, как будет выглядеть многочлен первой степени, при  $n=1$ . Это будет прямая, проходящая через две точки  $(x_0, f(x_0))$  и  $(x_1, f(x_1))$ :

$$L(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

Но можно записать последнее выражение в ином виде:

$$L(x) = f(x_0) \frac{x - x_1}{x_0 - x_1} + f(x_1) \frac{x - x_0}{x_1 - x_0}$$

Можно убедиться, что  $L(x_0) = f(x_0)$ ,  $L(x_1) = f(x_1)$ .

Похожим образом выглядит  $L(x)$  и для других значений  $n$ . Например, при  $n = 2$ :

$$L(x) = f(x_0) \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + \\ + f(x_1) \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f(x_2) \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

Уже можно заметить основную идею подобной записи. Для аргумента  $x_i$  в дроби при  $f(x_i)$  числитель оказывается равным знаменателю, в то время как другие слагаемые в выражении обнуляются.

Для произвольного значения  $n$  справедлива формула:

$$L(x) = \sum_{i=0}^n f(x_i) \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}. \quad (12)$$

Коэффициенты

$$B_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j},$$

удовлетворяющие равенству  $B_i(x_k) = \delta_{jk}$ , называются базисными или узловыми функциями.



У полученного таким образом многочлена с точки зрения практического использования есть серьёзные недостатки. При простых данных и небольших степенях  $n$  он отлично справится с задачей. Но при её усложнении начнут возникать странные эффекты. Между точками кривая может пройти так, как ей вздумается, с экстремумами в тех местах, где их быть не должно. Вне отрезка  $[a, b]$  функция вообще поведёт себя непредсказуемым образом. Кроме того, легко получить ситуацию, при которой малейшие изменения данных ведут к значительным изменениям кривой.

На рисунке 36 показаны некоторые из возможных эффектов даже при небольших степенях полинома.

Строго говоря, ничего странного в таком поведении нет. Единственное, что мы требовали от многочлена при построении — проходить через узловые точки (10). Поведение интерполирующей кривой в других местах никак не обговаривалось.

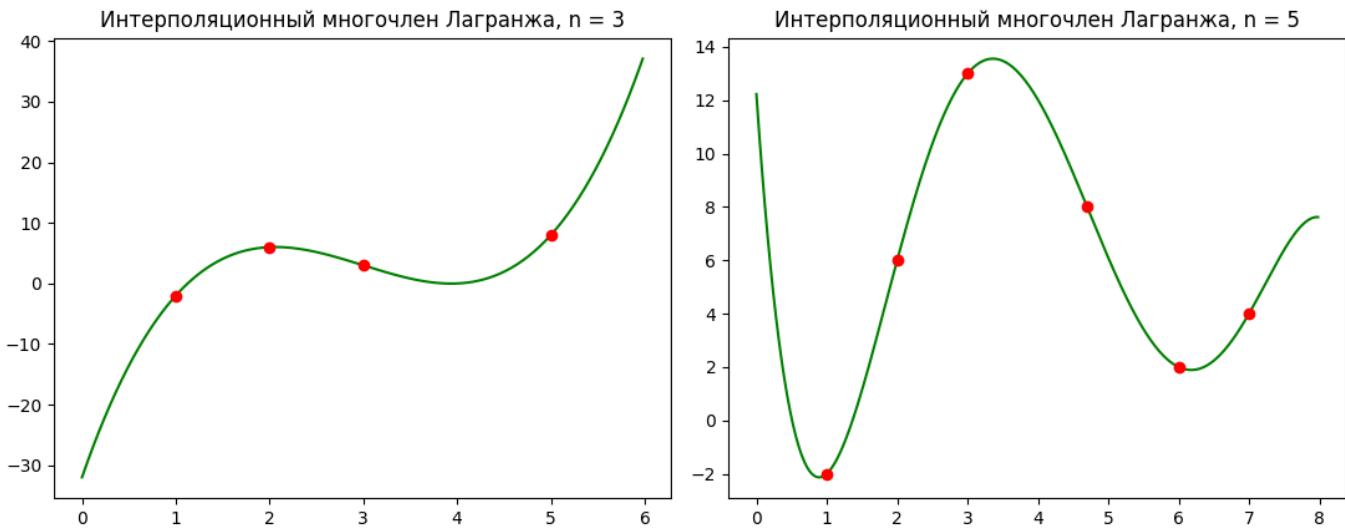


Рис. 36: Примеры построения многочленов Лагранжа 3-й и 5-й степени

Но ещё одна причина, почему полином Лагранжа лучше избегать на большинстве серьёзных задач, заключается в том, что не стоит пытаться подобрать под сложную фигуру одну единственную функцию. Гораздо удобнее и практичнее разбить её на сегменты и описать набором простых конструкций. Именно такой подход будет рассматриваться ниже.

## Кубический интерполяционный сплайн

Продолжим рассматривать задачу интерполяции. Как и прежде, будем работать с набором точек  $x_0 < x_1 < \dots < x_n$ . Построим функцию  $S(x)$ , удовлетворяющую условиям:

$$S(x_i) = f(x_i), i = 0, 1, \dots, n \quad (13)$$

Пусть  $S(x)$  — многочлен 3-й степени на каждом из отрезков  $[x_i, x_{i+1}]$ . Причём на разных сегментах используются разные многочлены, то есть в выражении

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, i = 0, 1, \dots, n-1 \quad (14)$$

коэффициенты  $a_i, b_i, c_i, d_i$  уникальны для каждого отрезка. Итого имеем  $4n$  неизвестных.

Из условия (13) получаем  $2n$  уравнений, описывающих границы каждого отрезка. Дополнительно от  $S(x)$  обычно требуют, чтобы она обладала непрерывными 1-ми и 2-ми производными во всех внутренних точках. Для этого на стыке отрезков нужно приравнять производные слева и справа:

$$\begin{cases} S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}), & i = 0, 1, \dots, n-2 \\ S''_i(x_{i+1}) = S''_{i+1}(x_{i+1}), & \end{cases} \quad (15)$$

Так получаем еще  $2n-2$  уравнений, вместе их уже  $4n-2$ . Для полноты системы не хватает ещё двух условий, которые обычно ставятся на концах кривой, в точках  $x_0$  и  $x_n$ . Такие условия называются краевыми, и возможны различные их варианты. Мы будем использовать обнуление вторых производных:

$$S''_0(x_0) = S''_n(x_n) = 0 \quad (16)$$

Теперь получим набор аналитических формул для построения описанной выше кривой.

Обозначим  $h_i = x_{i+1} - x_i, i = 0, 1, \dots, n-1$ .

Условие (13) превращается в два набора уравнений:

$$\begin{cases} S_i(x_i) = f(x_i), \\ S_i(x_{i+1}) = f(x_{i+1}), \end{cases} \quad i = 0, 1, \dots, n-1 \quad (17)$$

Из первого получаем:

$$\begin{aligned} S_i(x_i) &= a_i + b_i(x_i - x_i) + c_i(x_i - x_i)^2 + d_i(x_i - x_i)^3 = \\ &= a_i = f(x_i) \end{aligned} \quad (18)$$

Так найдена формула для получения коэффициентов  $a_i$ .

Второй набор уравнений превращается в следующее:

$$\begin{aligned} S_i(x_{i+1}) &= a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 + d_i(x_{i+1} - x_i)^3 = \\ &= a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = f(x_{i+1}) \end{aligned} \quad (19)$$

Теперь выведем формулы для производных:

$$S'_i(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \quad (20)$$

$$S''_i(x) = 2c_i + 6d_i(x - x_i) \quad (21)$$

Приравняем производные в соответствии с условиями (15). Для первой производной получим:

$$\begin{aligned} b_i + 2c_i(x_{i+1} - x_i) + 3d_i(x_{i+1} - x_i)^2 &= b_{i+1} \\ b_i + 2c_i h_i + 3d_i h_i^2 &= b_{i+1} \end{aligned} \quad (22)$$

Для второй аналогично:

$$2c_i + 6d_i h_i = 2c_{i+1}$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i} \quad (23)$$

Получена явная формула для  $d_i$ .

Если подставить её, а также выражение для  $a_i$  в (19), уравнение примет следующий вид:

$$f(x_i) + b_i h_i + c_i h_i^2 + \frac{c_{i+1} - c_i}{3h_i} h_i^2 = f(x_{i+1})$$

Откуда, если привести подобные и упростить, получается формула для  $b_i$ :

$$b_i = \frac{f(x_{i+1}) - f(x_i)}{h_i} - \frac{h_i}{3}(c_{i+1} + 2c_i) \quad (24)$$

А вот для  $c_i$ , через которую мы выразили остальные коэффициенты, формула будет сложнее и в неявном виде. Если подставить выражения для  $b_i$  и  $d_i$  в (22), а затем упростить (советуем проделать это самостоятельно для тренировки), получим:

$$h_i c_i + 2(h_i + h_{i+1})c_{i+1} + h_{i+1}c_{i+2} = 3\left(\frac{f(x_{i+2}) - f(x_{i+1})}{h_{i+1}} - \frac{f(x_{i+1}) - f(x_i)}{h_i}\right), \quad (25)$$

где  $i = 0, 1, \dots, n - 2$ .

Из краевых условий, обнуляющих вторые производные (16) также следует, что

$$c_0 = c_n = 0 \quad (26)$$

В итоге для нахождения коэффициентов  $c_i$  для оставшихся  $i = 1, \dots, n - 1$  необходимо решить систему линейных уравнений. После этого можно получить значения для  $b_i$  и  $d_i$ . А вместе с легко получаемыми  $a_i$  так находим все  $4n$  неизвестных и можем построить кривую.

Расширенную матрицу системы, получаемой на основе (25) и (26) можно записать в следующем виде:

$$\left( \begin{array}{cccccc|c} A_0 & B_0 & 0 & 0 & 0 & 0 & 0 & D_0 \\ C_1 & A_1 & B_1 & 0 & 0 & 0 & 0 & D_1 \\ 0 & C_2 & A_2 & B_2 & 0 & 0 & 0 & D_2 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & C_{n-3} & A_{n-3} & B_{n-3} & D_{n-3} \\ 0 & 0 & 0 & 0 & 0 & C_{n-2} & A_{n-2} & D_{n-2} \end{array} \right), \quad (27)$$

где

$$\begin{aligned} A_i &= 2(h_i + h_{i+1}), B_i = h_{i+1}, C_i = h_i, \\ D_i &= 3\left(\frac{f(x_{i+2}) - f(x_{i+1})}{h_{i+1}} - \frac{f(x_{i+1}) - f(x_i)}{h_i}\right). \end{aligned}$$

Полученный набор формул не единственный для кубического сплайна. В литературе можно встретить другие варианты записи, дающие тот же самый результат.



Обратим внимание на одну особенность системы (27). Не считая нулевых  $c_0$  и  $c_n$ , в каждое уравнение входит ровно 3 неизвестных, причем расположены они вдоль главной диагонали. Такие матрицы называются трёхдиагональными.

Решить такую систему можно любым стандартным приёмом, например, методом Гаусса. Но очевидно, что это будет неоптимально, поскольку придётся работать с большим количеством ненулевых элементов. Кроме того, они будут занимать лишнее место в памяти.

Чтобы избежать этого, для таких трёхдиагональных систем был придуман **метод прогонки**.

Отвяжемся сейчас от коэффициентов, используемых в формулах выше, и рассмотрим принцип его работы.

Итак, пусть имеется система линейных уравнений

$$AY = D,$$

где  $Y$  - столбец из  $n$  неизвестных, а  $D$  - правая часть уравнений.

$$Y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}, D = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{pmatrix}.$$

И расширенная матрица системы имеет следующий вид:

$$\left( \begin{array}{cccccc|c} a_0 & b_0 & 0 & 0 & 0 & 0 & 0 & d_0 \\ c_1 & a_1 & b_1 & 0 & 0 & 0 & 0 & d_1 \\ 0 & c_2 & a_2 & b_2 & 0 & 0 & 0 & d_2 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & c_{n-2} & a_{n-2} & b_{n-2} & d_{n-2} \\ 0 & 0 & 0 & 0 & 0 & c_{n-1} & a_{n-1} & d_{n-1} \end{array} \right).$$

Решение в методе прогонки ищется в виде:

$$y_i = v_i y_{i+1} + u_i, i = 0, 1, \dots, n-2 \quad (28)$$

Величины  $v_i$  и  $u_i$  называются коэффициентами прогонки, их сперва нужно найти.

Для начала рассмотрим первое уравнение системы при  $i = 0$ . Выразим  $y_0$  через  $y_1$ :

$$y_0 = -\frac{b_0}{a_0} y_1 + \frac{d_0}{a_0}.$$

Отсюда коэффициенты прогонки:

$$v_0 = -\frac{b_0}{a_0}, u_0 = \frac{d_0}{a_0}. \quad (29)$$

Далее предположим, что нам известны  $u_{i-1}$  и  $v_{i-1}$ , и попробуем на их основе определить  $u_i$  и  $v_i$ . Рассмотрим уравнение системы при  $i > 0$ :

$$c_i y_{i-1} + a_i y_i + b_i y_{i+1} = d_i, i = 1, \dots, n-2$$

Последнее уравнение системы при  $i = n-1$  тоже можно представить в таком виде, если формально положить коэффициент  $b_{n-1}$  и неизвестное  $y_n$ , которого на самом деле нет, равными нулю.

Избавимся в уравнении от  $y_{i-1}$ , подставив туда выражение (28):

$$c_i(v_{i-1}y_i + u_{i-1}) + a_iy_i + b_iy_{i+1} = d_i$$

Если раскрыть скобки и привести подобные, можно прийти к виду:

$$y_i = -\frac{b_i}{c_iv_{i-1} + a_i}y_{i+1} + \frac{d_i - c_iu_{i-1}}{c_iv_{i-1} + a_i}.$$

Отсюда коэффициенты прогонки:

$$v_i = -\frac{b_i}{c_iv_{i-1} + a_i}, u_i = \frac{d_i - c_iu_{i-1}}{c_iv_{i-1} + a_i}. \quad (30)$$

По формулам (29) и (30) можно найти все коэффициенты прогонки. Для удобства иногда формально полагают  $u_{-1} = 0$  и  $v_{-1} = 0$ . Нетрудно заметить, что тогда формула (29) является следствием (30).

Процесс нахождения  $u_i$  и  $v_i$  называется прямым ходом прогонки. При этом мы спускаемся по главной диагонали слева направо. Затем начинается обратный ход — подъём с нахождением  $y_i$  по рекуррентным формулам. При такой структуре метод прогонки очень похож на метод Гаусса и как раз является его частным случаем.

Напомним, что мы решили объявить  $b_{n-1}$  и несуществующий  $y_n$  равными нулю. Поэтому для  $i = n - 1$ :

$$y_{n-1} = v_{n-1}y_n + u_{n-1} = u_{n-1} = \frac{d_{n-1} - c_{n-1}u_{n-2}}{c_{n-1}v_{n-2} + a_{n-1}} \quad (31)$$

То есть при нахождении последних коэффициентов прогонки сразу определяется и  $y_{n-1}$ . Остаётся лишь  $n - 1$  раз применить формулу (28).

# Кривые Безье

Кривые Безье были придуманы в 60-х годах прошлого века независимо друг от друга Пьером Безье и Полем де Кастельжо для задач проектирования кузовов автомобилей. Впрочем, используемые в них базисные функции являются частным случаем многочленов Бернштейна, описанных на полвека ранее.

Для простоты изложения будем считать, что задаем кривую на плоскости. Но для читателя не составит труда заметить, что формулы можно применять и для больших измерений.

Итак, кривой Безье называется:

$$\vec{c}_n(t) = \sum_{k=0}^n \vec{p}_k b_{k,n}(t), \quad (32)$$

где  $n$  — степень кривой, а  $\vec{p}_k$  — это точки, по которым она строится. В отличие от предыдущих примеров кривая через них не проходит, за исключением первой и последней.

$b_{k,n}(t)$  - базисная функция, задаваемая выражением:

$$b_{k,n}(t) = C_n^k t^k (1-t)^{n-k}$$

Напомним, что  $C_n^k$  — число сочетаний:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Внутренний параметр  $t$  удовлетворяет неравенству:

$$0 \leq t \leq 1$$

Очевидно, что  $t = 0$  будет соответствовать началу кривой, а  $t = 1$  — ее концу. Таким образом, имея  $n + 1$  точку  $\vec{p}_k$ , можно выбрать небольшой шаг и, двигаясь вдоль параметра, получить необходимый результат.

На основе (32) легко выписываются выражения для первых степеней:

$$\vec{c}_1(t) = (1-t)\vec{p}_0 + t\vec{p}_1$$

$$\vec{c}_2(t) = (1-t)^2\vec{p}_0 + 2t(1-t)\vec{p}_1 + t^2\vec{p}_2$$

$$\vec{c}_3(t) = (1-t)^3\vec{p}_0 + 3t(1-t)^2\vec{p}_1 + 3t^2(1-t)\vec{p}_2 + t^3\vec{p}_3$$

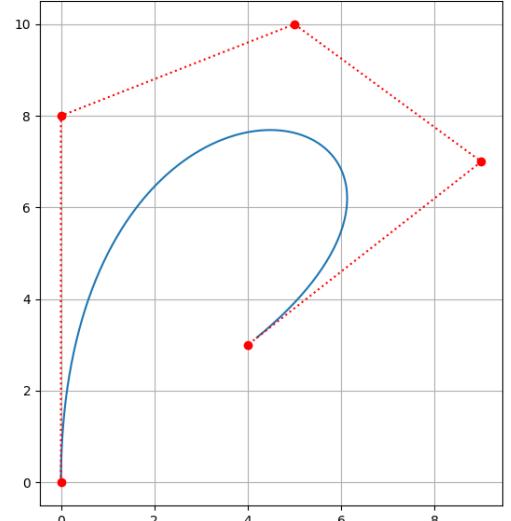


Рис. 37: Кривая Безье 4-й степени

Математически кривые Безье не являются сплайнами, но используются в более сложных конструкциях, например, в сплайнах Безье. С этой точки зрения больший интерес, конечно, представляют полученные выше первые степени: квадратичные и кубические.

Однако и сами по себе кривые находят применение во многих конструкторских, дизайнерских, художественных задачах. Их можно встретить почти в любом соответствующем теме программном пакете и приложении.

## Вспомогательные проекты

В качестве сопровождения к тексту в основном репозитории курса лежит несколько вспомогательных проектов.

### Приложение CubicSplinesSimpleApp

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task2/CubicSplinesSimpleApp>

Реализует интерполяцию с помощью кубического сплайна на языке Python, идеально подходящем для прототипирования, т.е. написания простых приложений с целью проверить какую-то идею. Проект может пригодиться тем, кто захочет самостоятельно запрограммировать кривую. При этом некоторые места в нём специально написаны так, чтобы нельзя было перенести код на другой язык без понимания формул.

От проекта можно оттолкнуться в выборе архитектуры приложения, изучить его реализацию алгоритма. Осмыслить всё окончательно и выписать план действий. Но после этого гораздо проще будет разрабатывать свой код самостоятельно, опираясь только на формулы.

Класс *CubicSpline* реализует логику построения сплайна. Здесь нет ничего необычного: в конструктор передаются узловые точки, а по ним строятся с помощью решения СЛАУ  $4n$  коэффициентов. Метод *point()* затем позволяет принять значение  $x$  и выдать соответствующий  $y$  на кривой.

Больший интерес для вашей собственной реализации может представлять класс *CubicSpline2D*. Благодаря нему появляется возможность не просто интерполировать функцию, но и построить произвольную кривую по точкам на плоскости.

Рассматриваемый выше кубический сплайн позволяет по формулам строить выражение вида  $y = S(x)$ .  $S(x)$  — это функция, она не может быть определена для одного и того же значения аргумента. Это значит, что с помощью неё можно изобразить ситуацию на рисунке (38) слева, но нельзя — ту, что справа.

Для того, чтобы это стало возможно, применяется следующий трюк. Класс *CubicSpline2D* хранит в себе два сплайна. Один отвечает за интерполяцию по  $x = S(t)$ , а второй — по  $y = S(t)$ . В итоге получаем конструкцию вида  $(x, y) = S(t)$ . Переменная  $t$  — внутренняя размерность кривой. Она принимает значения в диапазоне  $[t_0 = 0, t_{n-1}]$ , где  $t_0$  соответствует первой узловой точке  $(x_0, y_0)$ , а  $t_{n-1}$  — последней. Промежуточным значениям  $t_i$  в узловых точках  $(x_i, y_i)$  назначаются величины в соответствии с евклидовым расстоянием между ними.

```
1 class CubicSpline2D:
2     def __init__(self, x, y):
3         self.params = self.__calculate_params(x, y)
4         self.sx = CubicSpline(self.params, x)
5         self.sy = CubicSpline(self.params, y)
```

```

7  def point(self, param):
8      x = self.sx.point(param)
9      y = self.sy.point(param)
10     return x, y
11
12 def __calculate_params(self, x, y):
13     dx = np.diff(x)
14     dy = np.diff(y)
15     self.ds = [np.sqrt(idx ** 2 + idy ** 2) for (idx,
16         idy) in zip(dx, dy)]
17     s = [0.0]
18     s.extend(np.cumsum(self.ds))
        return s

```

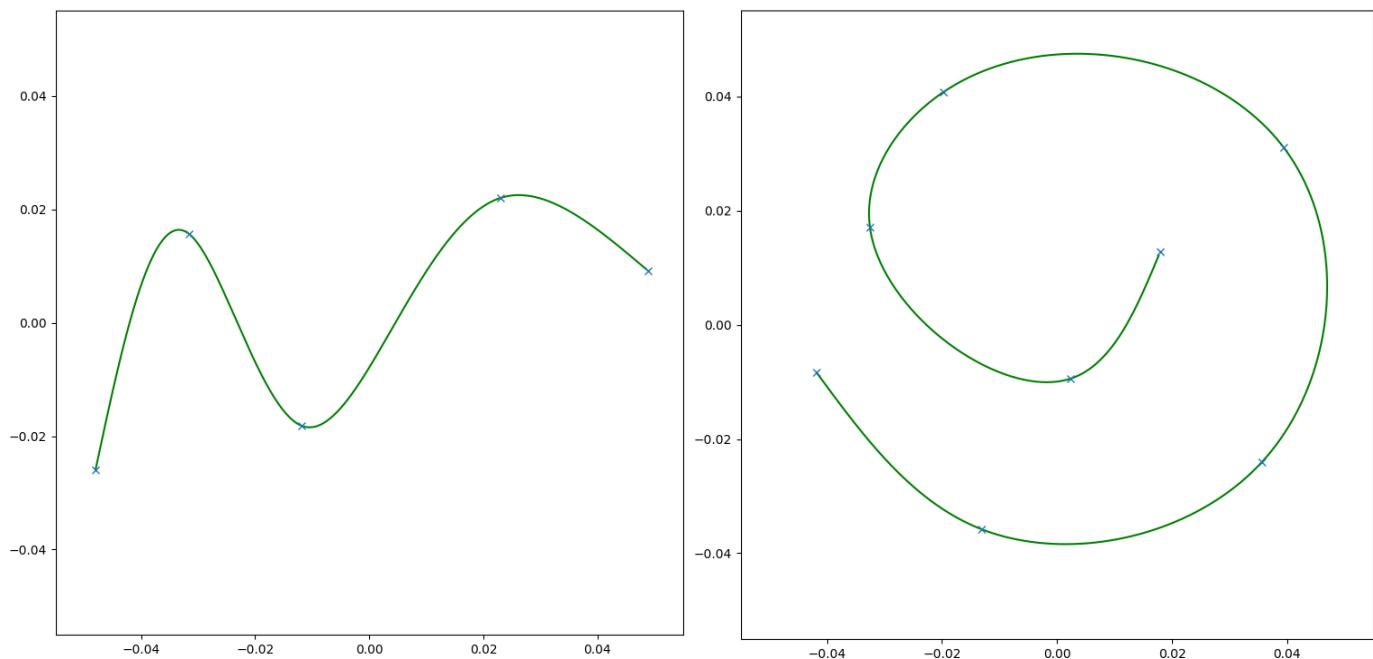


Рис. 38: Результат работы приложения CubicSplinesSimpleApp

## Приложение ProtoCurveFXApp

<https://github.com/kv1tr4vn/CGVSU/tree/main/Task2/ProtoCurveFXApp>

Чтобы вы тратили меньше времени на знакомство с инструментом и быстрее сосредоточили силы на алгоритме, было подготовлено приложение на JavaFX, в котором по клику мыши создаются точки, а по ним строится ломаная. Можно оттолкнуться от этого проекта при написании своей кривой. Но, конечно, её следует оформлять в коде в виде отдельных классов и методов, а потом лишь вызывать в нужных местах для отрисовки.

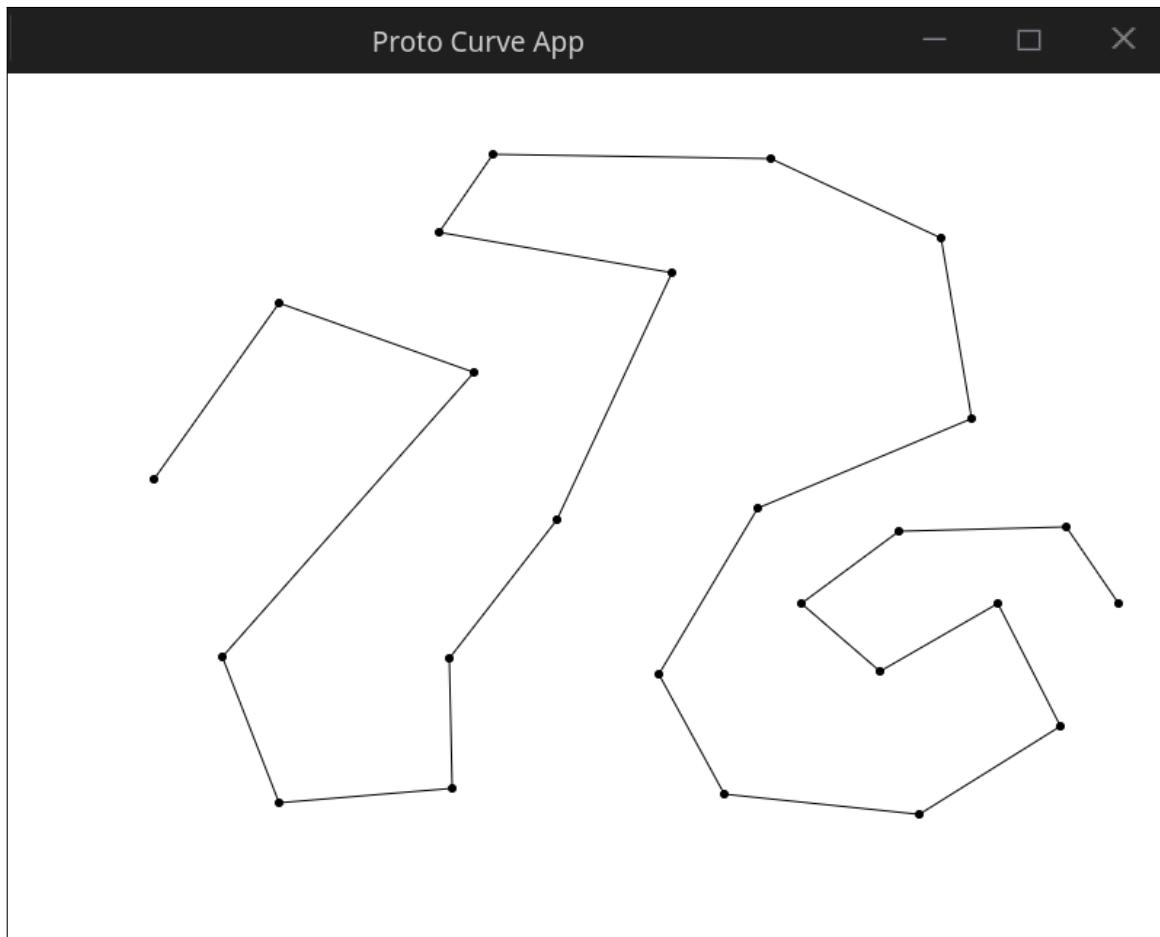


Рис. 39: Результат работы приложения ProtoCurveFXApp

## Список дополнительной литературы

1. Тыртышников Е. Матричный анализ и линейная алгебра / Е.Е. Тартышников.  
— Москва : Физматлит, 2007. — 480 с.
2. Moller, T. Real-Time Rendering / Tomas Akenine-Moller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, Sébastien Hillaire. — Fourth edition — Boca Raton: Taylor & Francis, CRC Press, 2018. — 1196 p.
3. Marschner, S. Fundamentals of Computer Graphics / Steve Marschner, Peter Shirley, Michael Ashikhmin, Michael Gleicher, Naty Hoffman, Garrett Johnson, Tamara Munzner, Erik Reinhard, William B. Thompson, Peter Willemse, Brian Wyvill. — Fourth edition — Boca Raton: Taylor & Francis, CRC Press, 2016. — 737 p.
4. Ammeraal, L. Computer Graphics for Java Programmers / Leen Ammeraal, Kang Zhang. — Third Edition — John Wiley & Sons Ltd, 2017. — 388 p.
5. Vince, J. Calculus for Computer Graphics / John Vince. — Second Edition — London: Springer-Verlag, 2019. — 303 p.
6. Vince, J. Vector Analysis for Computer Graphics / John Vince. — Second Edition — London: Springer-Verlag, 2021. — 252 p.