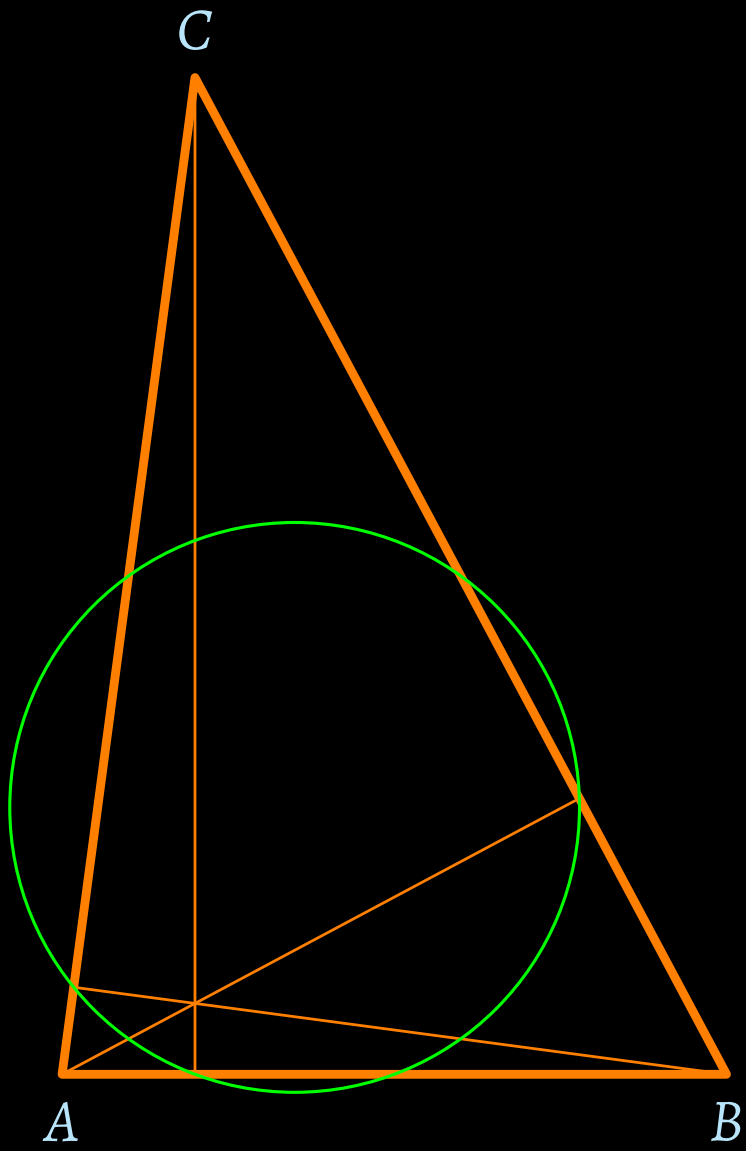


METAPOST

краткий курс

А. Н. Швец

⟨shvetz.anton@gmail.com⟩

2022-12-21T12:16:52+03:00

Что такое METAPOST?

- это алгоритмический язык
- это система программирования (компилятор + библиотеки)

METAPOST был создал Джоном Хобби (John D. Hobby).

Во многом система METAPOST основана на системе METAFONT, созданной известным американским математиком Дональдом Кнутом (Donald E. Knuth). METAFONT задуман как средство изготовления шрифтов для системы компьютерной вёрстки TEX.

Возможности METAPOST

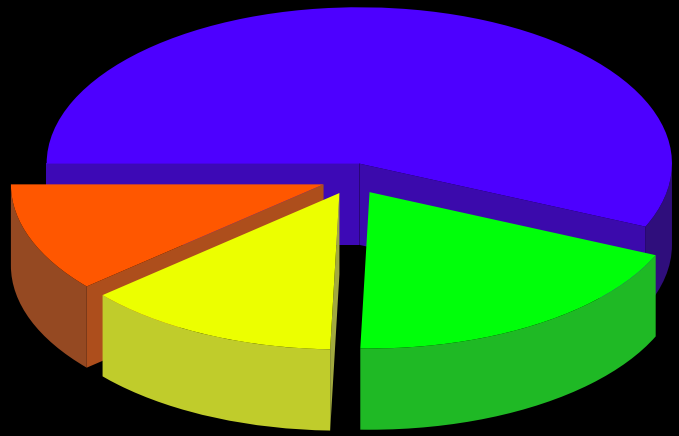
- создание рисунков в векторном графическом формате PostScript
- сложные вычисления при подготовке рисунков
- широкий набор встроенных и библиотечных команд
- возможность программирования пользовательских команд
- вставка тескта, отформатированного в системе TEX

Сфера применения METAPOST

- подготовка чертежей
  и технических рисунков
- диаграмм
- графиков функций
- иллюстраций по геометрии
  (с формулами)
- логотипов

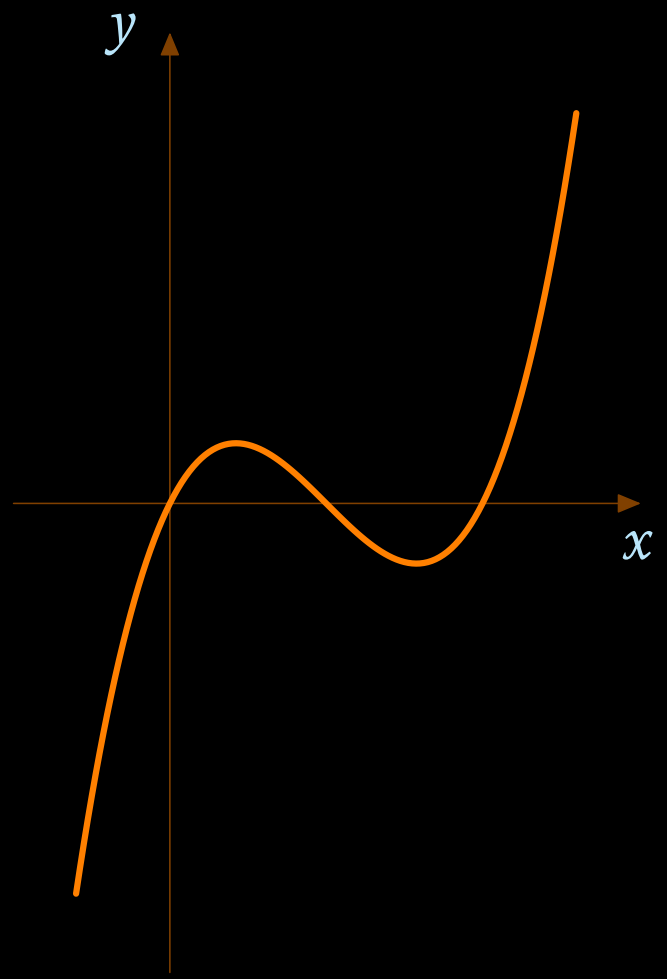
Сфера применения METAPOST

- подготовка чертежей
  и технических рисунков
- диаграмм
- графиков функций
- иллюстраций по геометрии
  (с формулами)
- логотипов



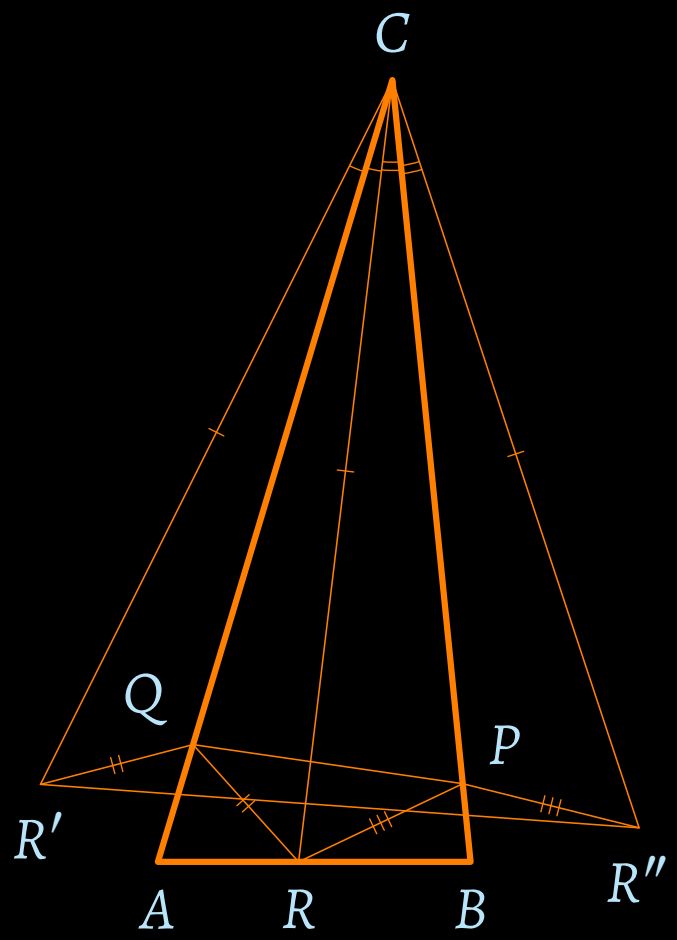
Сфера применения METAPOST

- подготовка чертежей
  и технических рисунков
- диаграмм
- графиков функций
- иллюстраций по геометрии
  (с формулами)
- логотипов



Сфера применения METAPOST

- подготовка чертежей
  и технических рисунков
- диаграмм
- графиков функций
- иллюстраций по геометрии
  (с формулами)
- логотипов



Сфера применения METAPOST

- подготовка чертежей
  и технических рисунков
- диаграмм
- графиков функций
- иллюстраций по геометрии
  (с формулами)
- логотипов

Особенности METAPOST

METAPOST не является системой WYSIWYG (What You See Is What You Get — «вы получаете то, что видите»). Это значит, что рисунок готовится как программа, которая компилируется, и в результате компиляции получается файл в формате PostScript, содержащий изображение.

PostScript

PostScript — это не только графический формат, но и алгоритмический язык. В нём заложены богатые графические и вычислительные возможности.

Однако из-за особенностей этого языка на нём трудно программировать «вручную». Такое программирование требует от программиста совершенно иного мышления, нежели программирование на «традиционных» алгоритмических языках.

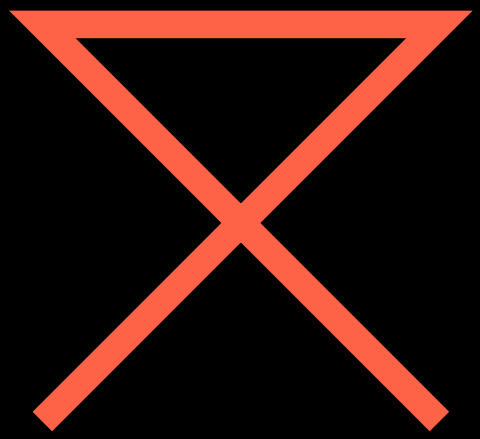
Чаще всего программы на PostScript генерируются автоматически другими программами.

Предназначение PostScript

Главное предназначение PostScript — управление растровыми устройствами, такими как принтеры, экран монитора или графический файл.

Некоторые устройства вроде дорогих профессиональных принтеров имеют встроенный интерпретатор PostScript. Другие понимают более примитивный управляющий язык, поэтому для управления ими требуется программа-переводчик (транслятор). Наиболее популярный транслятор языка PostScript — программа GhostScript.

Пример на языке PostScript



```
%!PS-Adobe

5 setlinewidth
1 .388235294117647 .27843137254902
    setrgbcolor

newpath
8 dup moveto
72 dup rlineto
-72 0 rlineto
72 -72 rlineto
stroke

showpage
```

Где взять METAPOST

- METAPOST для Linux, Windows, Mac OS X входит в пакет программ TEX Live: https://tug.org/texlive/
- METAPOST для Windows входит в пакет программ MiKTEX: https://miktex.org/

Что почитать про METAPOST

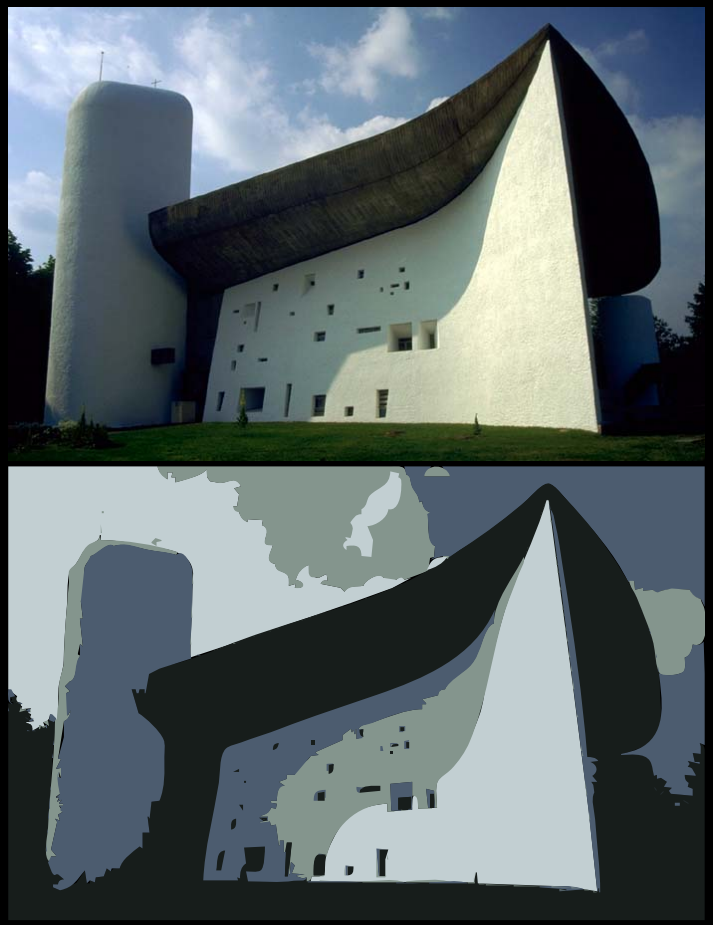
- A User's Manual for METAPOST
  Hobby, John D.
  https://www.tug.org/docs/metapost/mpman.pdf
- METAPOST. Руководство пользователя
  Hobby, John D.
  http://www.ctan.org/tex-archive/info/metapost/doc/russian/mpman-ru/mpman-ru.pdf
- METAPOST: A Reference Manual
  Grogono, Peter
  http://users.encs.concordia.ca/~grogono/Writings/mpref.pdf
- METAPOST — Википедия
  http://ru.wikipedia.org/wiki/MetaPost
  Статья в русской Википедии
- Exemples d'utilisation de Métapost
  Zoonekynd, Vincent
  http://zoonek.free.fr/LaTeX/Metapost/metapost.html

Что почитать про METAPOST *(продолжение)*

- Создание иллюстраций в METAPOST
  Балдин, Е. М.
  http://www.inp.nsk.su/~baldin/mpost/
  Статья для журнала «Linux Format»
- Learning METAPOST by Doing
  Heck, André
  http://remote.science.uva.nl/~heck/Courses/mptut.pdf
- METAPOST: A Very Brief Tutorial
  Urs, Oswald
  http://www.ursoswald.ch/metapost/tutorial.pdf
- Practical introduction to METAPOST
  Hurlin, Clément
  http://www-sop.inria.fr/everest/Clement.Hurlin/misc/Practical-introduction-to-MetaPost.pdf

Что почитать про METAPOST *(продолжение)*

- Drawing with METAPOST
  Thurston, Toby
  https://raw.githubusercontent.com/thruston/Drawing-with-Metapost/master/Drawing-with-Metapost.pdf
- Всё про METAFONT
  Кнут, Дональд Э.
  Москва : Вильямс, 2003. — ISBN 5-8459-0442-0 (рус.)



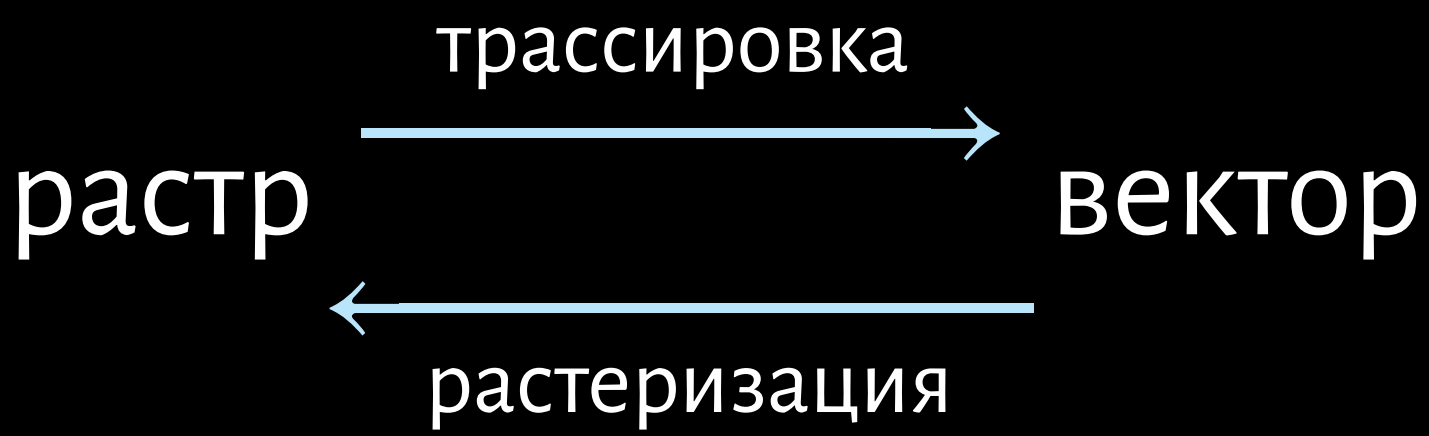
Растровая графика

- изображение формируется как прямоугольный массив точек (пикселов)
- наиболее подходит для фотоизображений
- популярные форматы: PNG, JPEG, GIF, PCX, BMP, ...
- популярные программы обработки изображений: Adobe PhotoShop, The Gimp, ...
- плохо переносит масштабирование и вращение

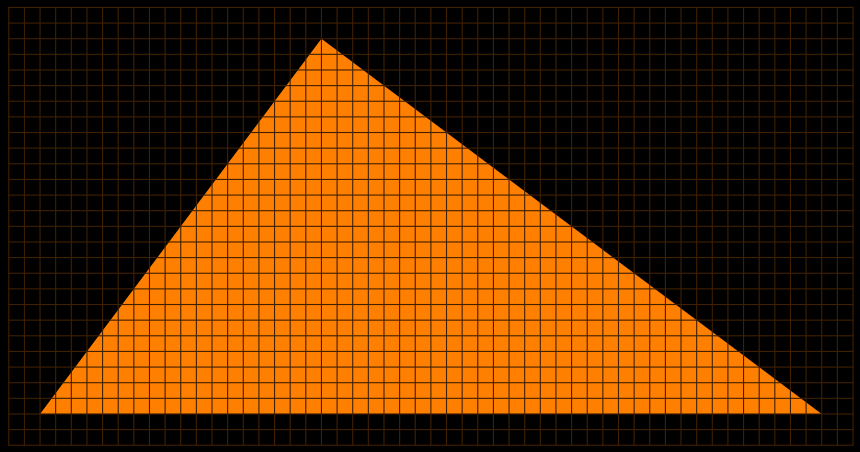
Векторная графика

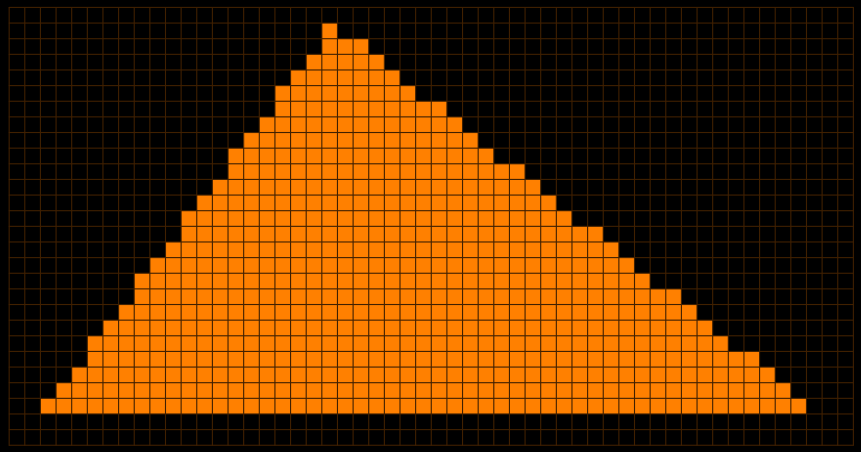
- изображение формируется из графических примитивов (линий, фигур, надписей)
- наиболее подходит для чертежей, графиков, диаграмм
- популярные форматы: SVG, PostScript, PDF, ...
- популярные программы обработки изображений: CorelDraw, InkScape, ...
- отлично переносит масштабирование и вращение

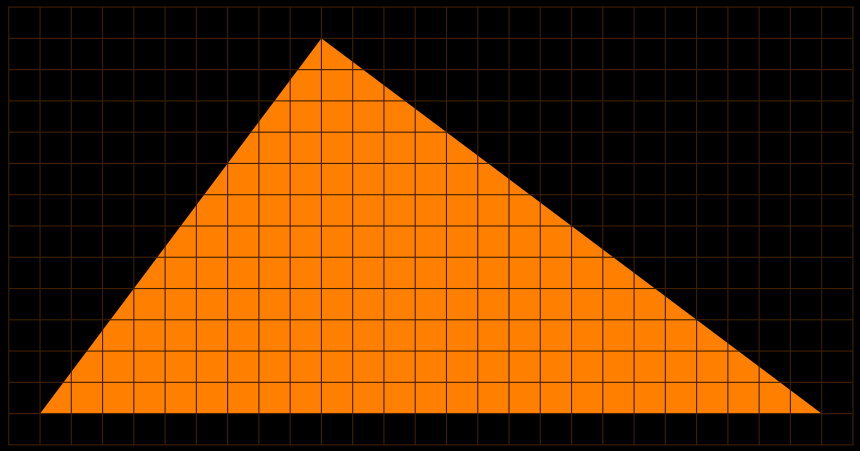
Трассировка и растеризация

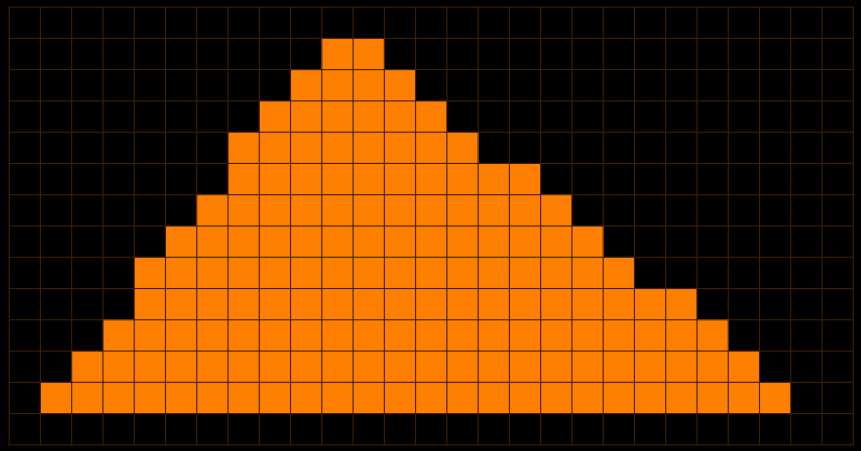


Растеризация









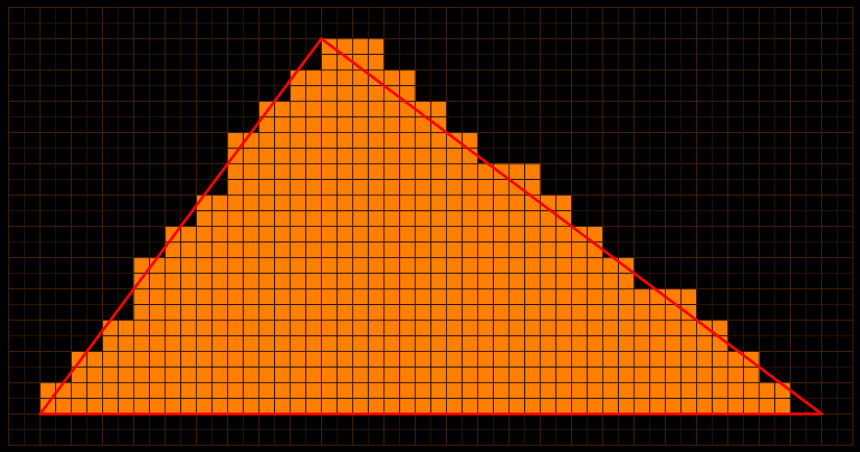
Растеризация

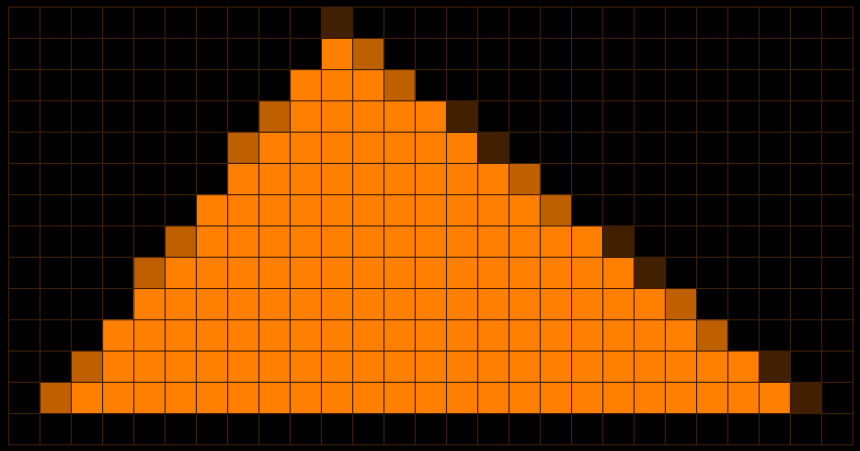
Результат зависит:

- от разрешения (размера клеток решётки)
- от положения фигуры на решётке
- от того, считать точки на границе фигуры её частью, или нет

Чем выше разрешение, тем лучше результат.

Сглаживание (antialiasing)





- измельчается решётка в 2 раза (иногда в 4 или 8 раз)
- для каждой крупной клетки подсчитывается доля мелких клеток, задевающих фигуру
- крупная клетка закрашивается цветом пропорционально найденной доле

Сглаживание (antialiasing)

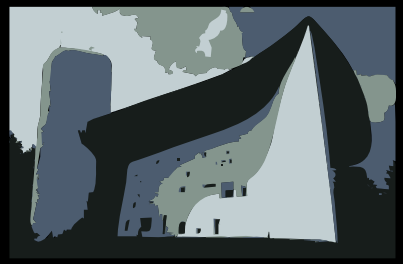
Сглаживание изображений букв в программах компьютерной вёрстки устроено сложнее.

В частности, особого подхода требуют прямолинейные горизонтальные или вертикальные участки границы буквы.

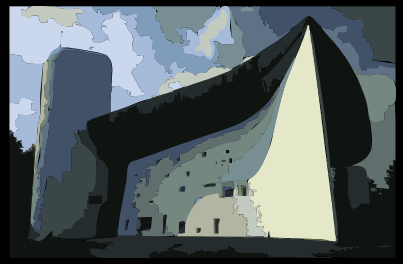
Для лучшего качества на таких участках сглаживание подавляется.

Трассировка (векторизация)

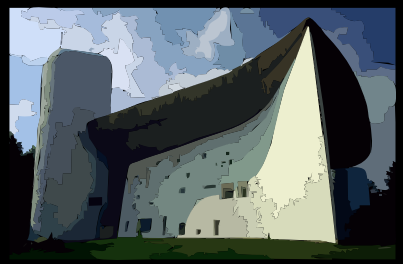
2 цвета



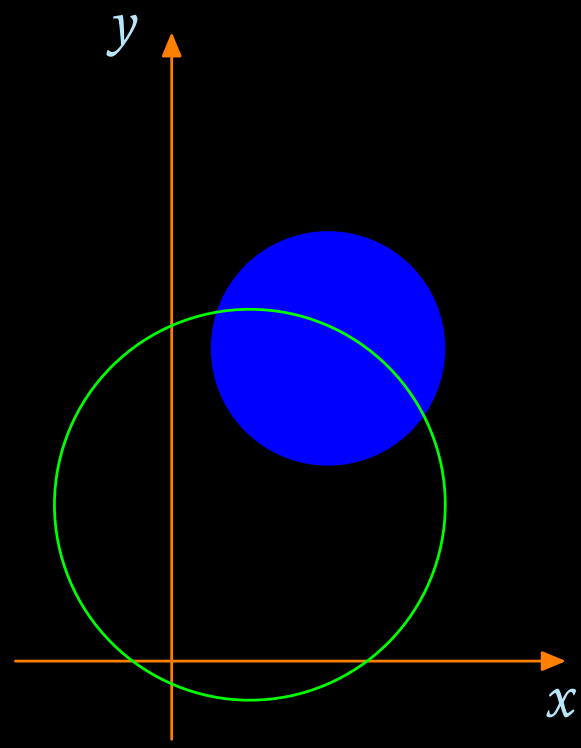
4 цвета



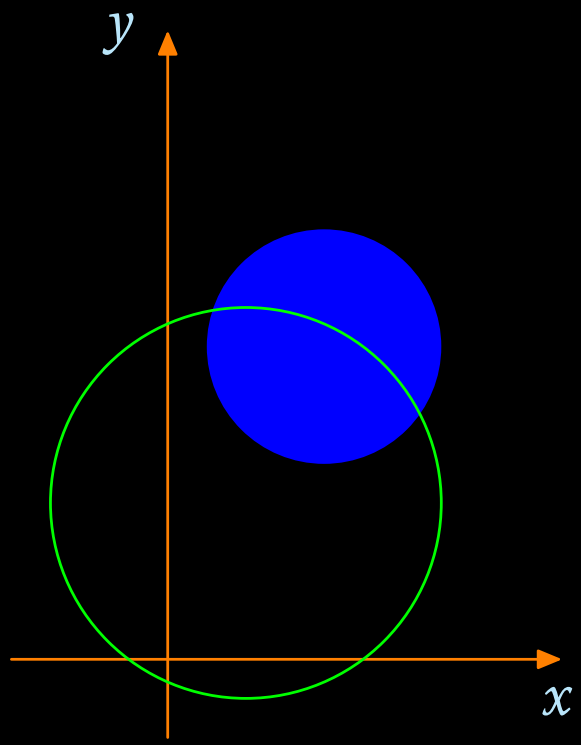
16 цветов



256 цветов



Пример: шаг за шагом



Пример: шаг за шагом

Эта команда начинает рисунок № 1:

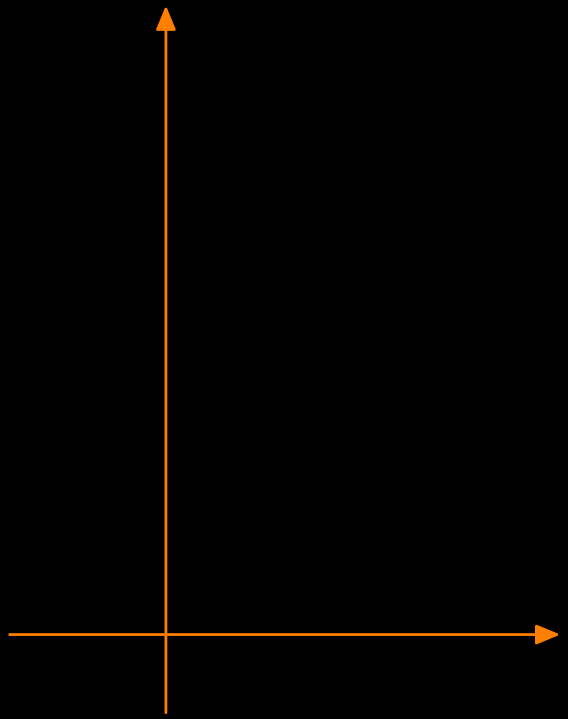
```
beginfig(1)
```

Пример: шаг за шагом

Определяем 6 точек на плоскости с помощью уравнений:

```
z1=(-.5cm, 0);
z2=(4cm, 0);
z3=(0, -.5cm);
z4=(0, 5.5cm);
z5=(3cm, 4cm);
z6=(2cm, 3cm);
```

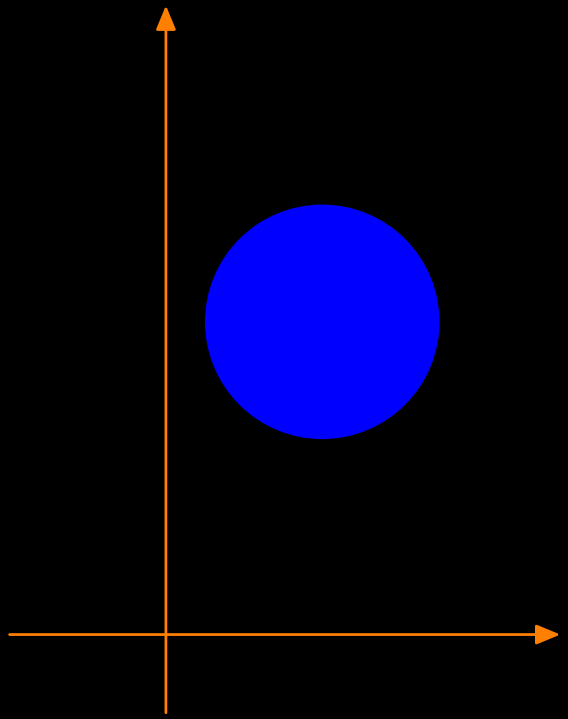
Пример: шаг за шагом



Рисуем координатные оси:

```
drawarrow z1--z2;
drawarrow z3--z4;
```

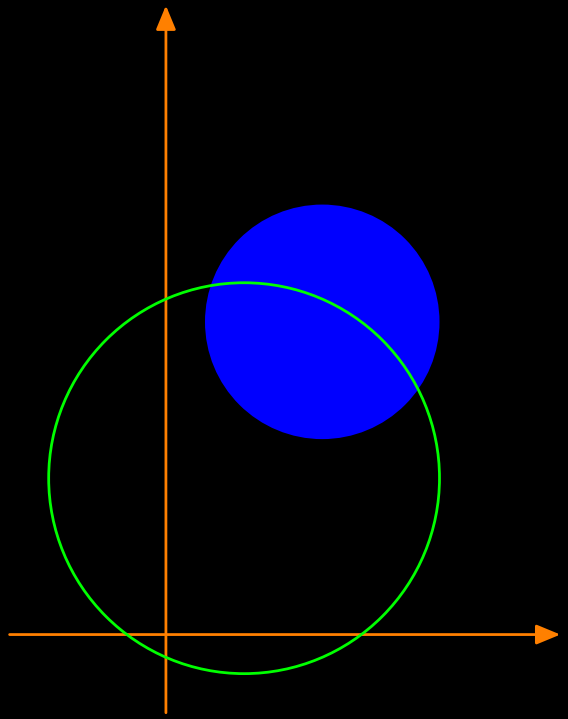
Пример: шаг за шагом



Рисуем сплошной синий круг диаметром 1 см:

```
fill fullcircle
    scaled 1cm
    shifted z5
    withcolor blue;
```

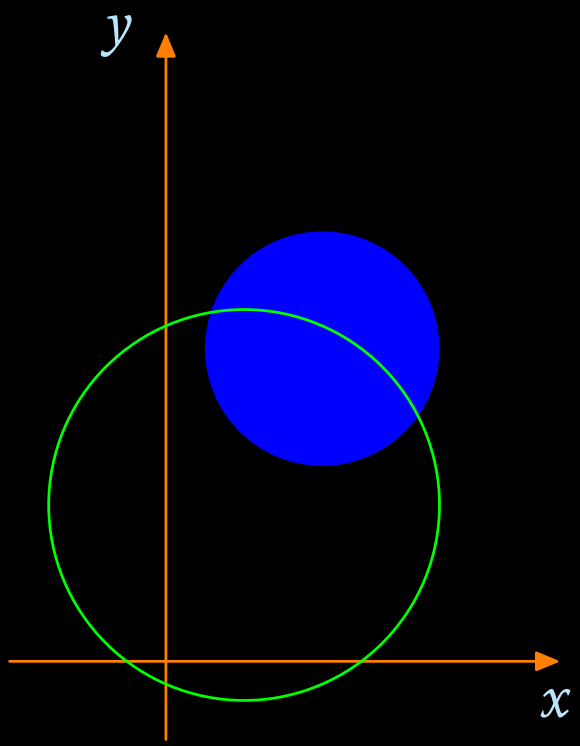
Пример: шаг за шагом



Рисуем контурную зелёную окружность диаметром 2 см:

```
draw fullcircle
    scaled 2cm
    shifted z6
    withcolor green;
```

Пример: шаг за шагом



Наносим надписи (метки):

```
label.bot(btex $x$ etex, z2);
label.lft(btex $y$ etex, z4);
```

Пример: шаг за шагом

Завершаем рисунок:

```
endfig
```

Завершаем программу:

```
bye.
```

Пример: шаг за шагом

Теперь записываем текст программы в файл (например, `example.mp`):

```
% vim example.mp
```

Пример: шаг за шагом

Теперь записываем текст программы в файл (например, `example.mp`):

```
% vim example.mp
```

Запускаем компилятор:

```
% mpost example.mp
```

Пример: шаг за шагом

Теперь записываем текст программы в файл (например, `example.mp`):

```
% vim example.mp
```

Запускаем компилятор:

```
% mpost example.mp
This is MetaPost, version 2.02 (TeX Live 2022/TeX Live)
(example.mp [1] )
1 output file written: example.1
Transcript written on example.log.
```

Всё готово!

Пример: шаг за шагом

Теперь записываем текст программы в файл (например, `example.mp`):

```
% vim example.mp
```

Запускаем компилятор:

```
% mpost example.mp
Это MetaPost, версия 2.02 (TeX Live 2022/TeX Live)
(example.mp [1] )
Записан 1 выходной файл: example.1
Протокол записан в example.log.
```

Всё готово!

Результат компиляции

В результате компиляции файла `example.mp` получаются файлы `example.1`, `example.2`, … — для каждой команды `beginfig(1)`, `beginfig(2)`, …

В этих файлах содержится графика в формате PostScript.

Кроме того, записывается файл протокола `example.log` с подробностями процесса компиляции.

numeric
pair
boolean
color
transform
path
pen
picture
string

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

число

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

числовая пара

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

логическое значение

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

цвет

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

аффинное
преобразование
плоскости

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

кривая

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

рисовальное перо

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

картинка

Типы в METAPOST

- numeric
- pair
- boolean
- color
- transform
- path
- pen
- picture
- string

строка

Переменные

Перед использованием переменную необходимо объявить:

```
numeric n, k;
```

Переменные

Перед использованием переменную необходимо объявить:

```
numeric n, k;
pair c, d, e;
```

Переменные

Перед использованием переменную необходимо объявить:

```
numeric n, k;
pair c, d, e;
path p;
```

Переменные

Перед использованием переменную необходимо объявить:

```
numeric n, k;
pair c, d, e;
path p;
```

Для переменных типа `numeric` объявление не обязательно.

Числа и пары

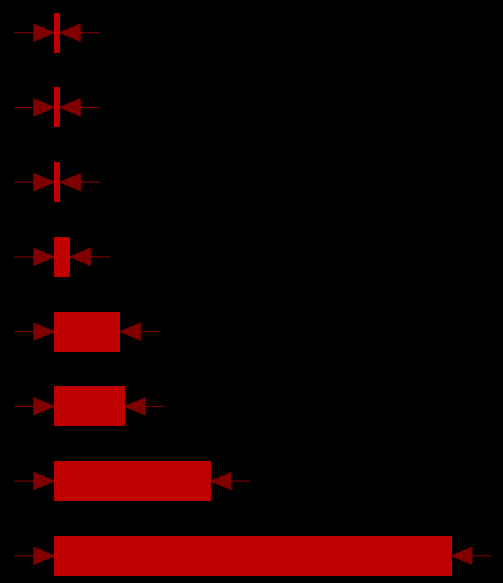
В METAPOST значения числового типа используются, помимо прочего, для представления координат точек.

Сами точки представляются как пары координат.

Единицы масштаба

Единицей масштаба служит большой типографский пункт (bp) — ему соответствует число 1. Для этой и некоторых других традиционных для полиграфии единиц измерения длин в библиотеке plain.mp определены числовые переменные:

пункт	0,99626	pt
большой пункт	1	bp
пункт Дидо́	1,06601	dd
миллиметр	2,83464	mm
пика	11,95517	pc
ци́церо	12,79213	cc
сантиметр	28,34645	cm
дюйм	72	in



Числа: арифметика и тригонометрия

`+x`	$+x$
`-x`	$-x$
`x+y`	$x+y$
`x-y`	$x-y$
`x*y`	xy
`x/y`	x/y
`x**y`	x^y
`x++y`	$\sqrt{x^2+y^2}$
`x+-+y`	$\sqrt{x^2-y^2}$
`sqrt x`	$\sqrt{x}$
`abs x`	$\lvert x\rvert$

`floor` x	целая часть x
`round` x	округление x
x `mod` y	остаток от деления x на y
x `div` y	целая часть отношения x и y
`cosd` ϑ	$\cos(\vartheta \cdot \pi/180)$
`sind` ϑ	$\sin(\vartheta \cdot \pi/180)$
$t[a,b]$	медиатор *(см. дальше)*

Пары: декомпозиция и арифметика

`xpart` (x, y)	x
`ypart` (x, y)	y
(x, y) `+` (u, v)	$(x + u, y + v)$
t `*` (x, y), (x, y) `*` t	(tx, ty)
(x, y) `/` t	$(x/t, y/t)$
(x, y) `dotprod` (u, v)	$xu + yv$
`abs` (x, y)	$\sqrt{x^2 + y^2}$
`unitvector` (x, y)	(x, y) `/` `abs` (x, y)
`angle` (x, y)	$180/\pi \cdot \operatorname{arctg}(y/x)$
`dir` ϑ	$(\cos(\vartheta \cdot \pi/180), \sin(\vartheta \cdot \pi/180))$

Медиатор

Для числа t и величин a и b одинакового типа, либо `numeric`, либо `pair`, либо `color`, в METAPOST определяется операция *медиатор (аффинная комбинация)*:

$$t[a,\ b] \quad \equiv \quad (1-t)\,a + t\,b$$

Медиатор

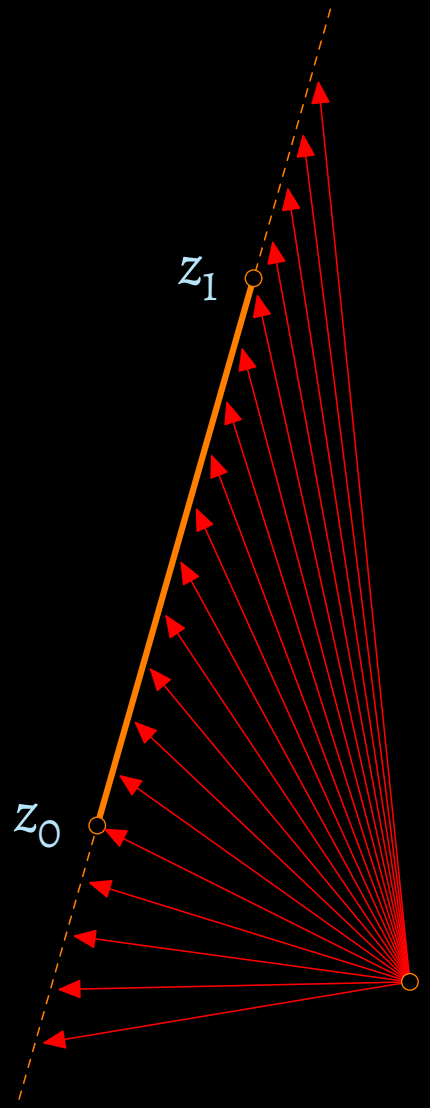
$$.2[3,\ 8] \quad \equiv \quad 4$$

$$.6[z1,\ z2]$$

$$\equiv$$

точка, делящая отрезок от $z1$ до $z2$ в отношении $3 : 2$

Кинематический смысл медиатора

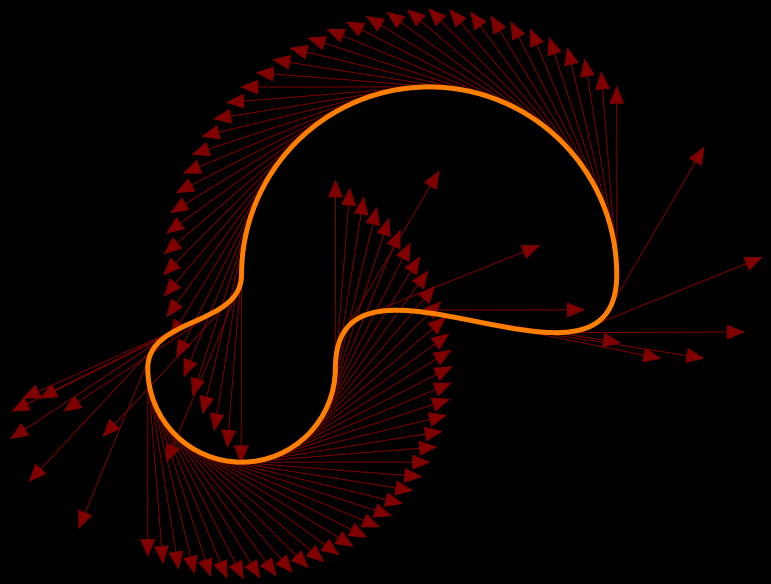


Между прочим, выражение

$$z(t) = (1 - t)z_0 + tz_1$$

задаёт закон равномерного прямолинейного движения, при котором точка с радиусом-вектором $z(t)$ в момент времени $t = 0$ проходит через точку z_0, а при $t = 1$ — через z_1.

Отрицательные значения t соответствуют точкам, находящимся на продолжении отрезка z_0z_1 за точку z_0, а значения, бóльшие единицы, отвечают точкам на продолжении этого отрезка за точку z_1.



Полином Бернштейна

Задаётся $n+1$ точкой z_0, z_1, $\ldots$, z_n (*контрольные точки*).

Полином Бернштейна n-й степени:

$$z(t) = \sum_{i=0}^{n} \binom{n}{i} z_i (1-t)^i t^{n-i}.$$

Биномиальный коэффициент:

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}.$$

Полином назван в честь советского математика Сергея Бернштейна.

Кривые Безье

Кривые, заданные параметрически полиномом Бернштейна, называются *кривыми Безье* соответствующей степени.

Активно используются в компьютерной графике для представления линий.

В METAPOST применяются кривые Безье 3-й степени.

Названы в честь французского инженера Пьера Безье (Piere Étienne Bésier), указавшего на их важность в компьютерной графике.

Свойства кривой Безье

- выходит из точки z_0 при $t = 0$
- приходит в точку z_n при $t = 1$
- касательная в точке z_0 «смотрит» в точку z_1
- касательная в точке z_n «смотрит» из точки z_{n-1}
- при $t \in [0; 1]$ целиком содержится в выпуклой оболочке всех контрольных точек

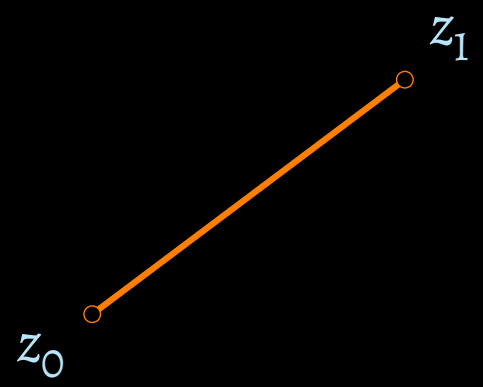
Кривая Безье 0-й степени

$$z(t) \equiv z_0$$

точка

Кривая Безье 1-й степени

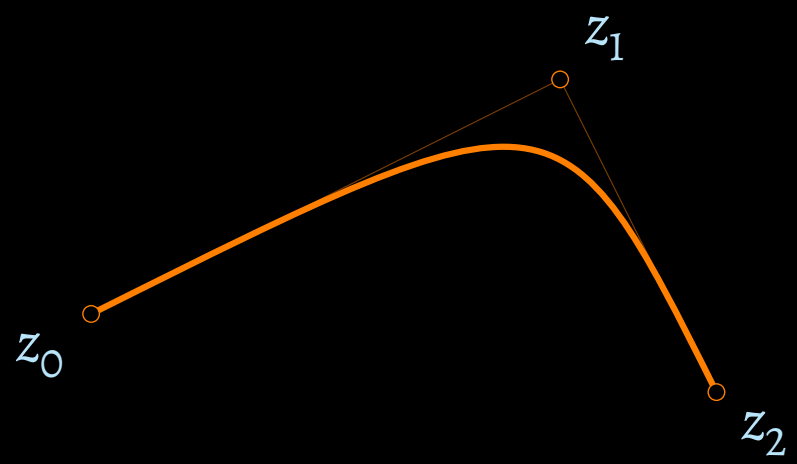
$$z(t) = z_0(1-t) + z_1 t$$



прямая линия

Кривая Безье 2-й степени

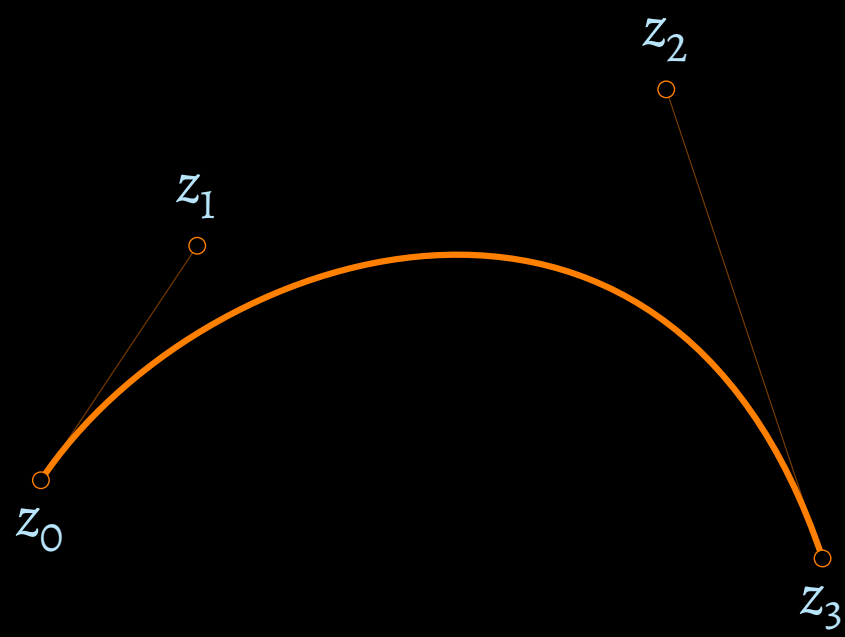
$$z(t) = z_0(1-t)^2 + 2z_1(1-t)t + z_2t^2$$



парабола

Кривая Безье 3-й степени

$$z(t) = z_0(1-t)^3 + 3z_1(1-t)^2 t + 3z_2(1-t)t^2 + z_3 t^3$$

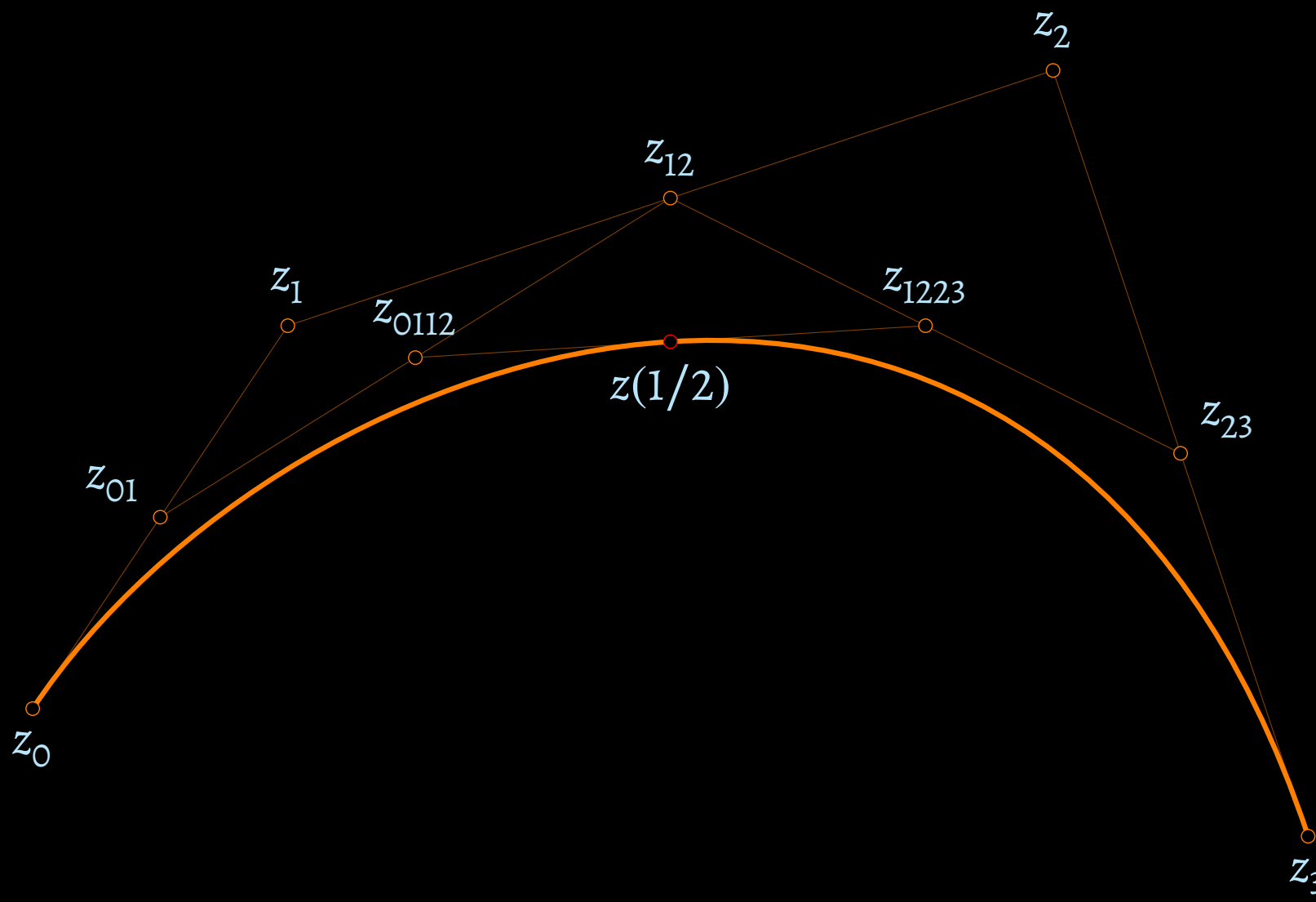


кубическая кривая (в школе не проходят)

Преимущества кривых Безье 3-й степени

- хорошо приближают типичные кривые линии
- в памяти компьютера представляются восьмёркой чисел (координатами четырёх точек)
- строятся с помощью простого рекурсивного алгоритма

Построение точки $z(1/2)$ для кривой 3-й степени

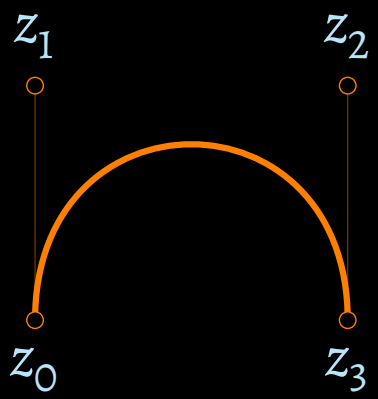


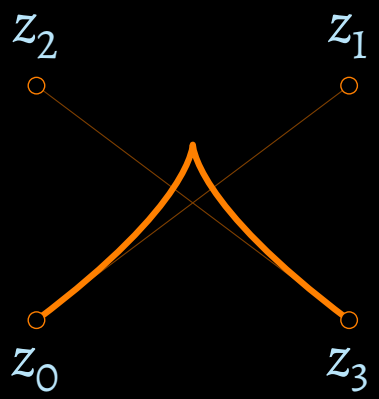
Построение кривых Безье 3-й степени

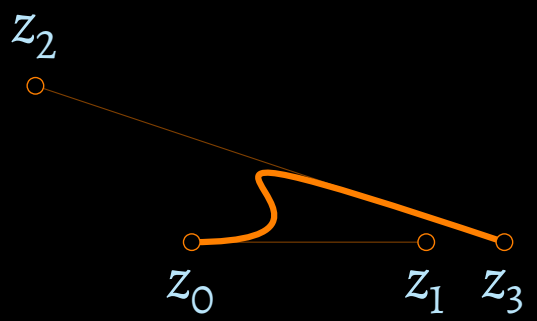
Точка $z(1/2)$ строится, как показано на предыдущем слайде. Отрезок кривой при $t \in [0; 1/2]$, оказывается, — это кривая с контрольными точками z_0, z_{01}, z_{0112}, $z(1/2)$. А отрезок при $t \in [1/2; 1]$ — кривая с контрольными точками $z(1/2)$, z_{1223}, z_{23}, z_3.

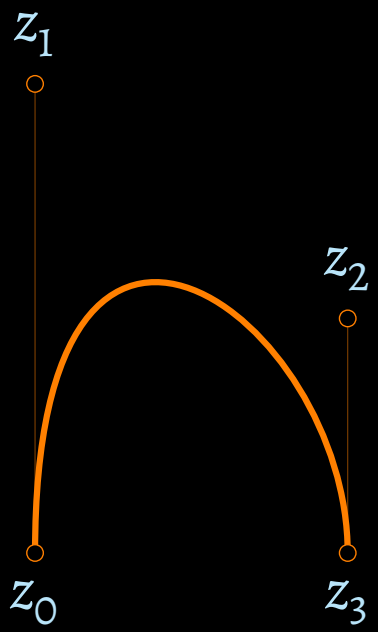
Для этих двух отрезков также находятся точки, отвечающие $t = 1/2$. Эти точки разбивают каждый из отрезков на половинки. Для них опять находятся «серединные» точки. Когда «серединных» точек будет достаточно много, они вычертят линию.

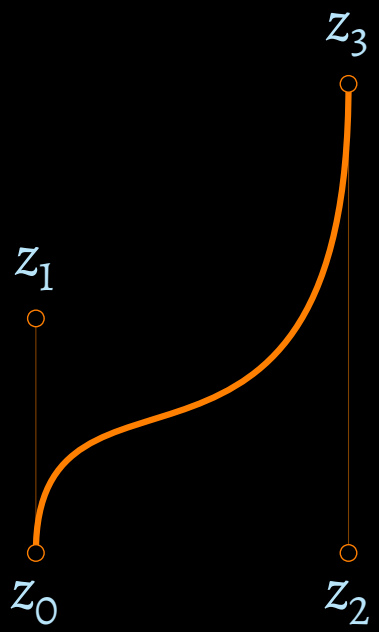
Примеры кривых Безье 3-й степени

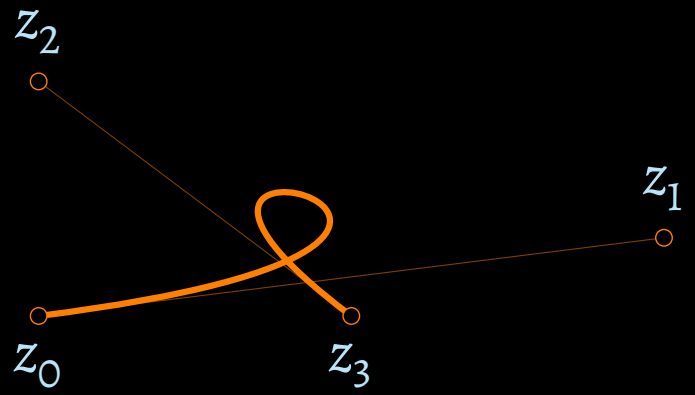








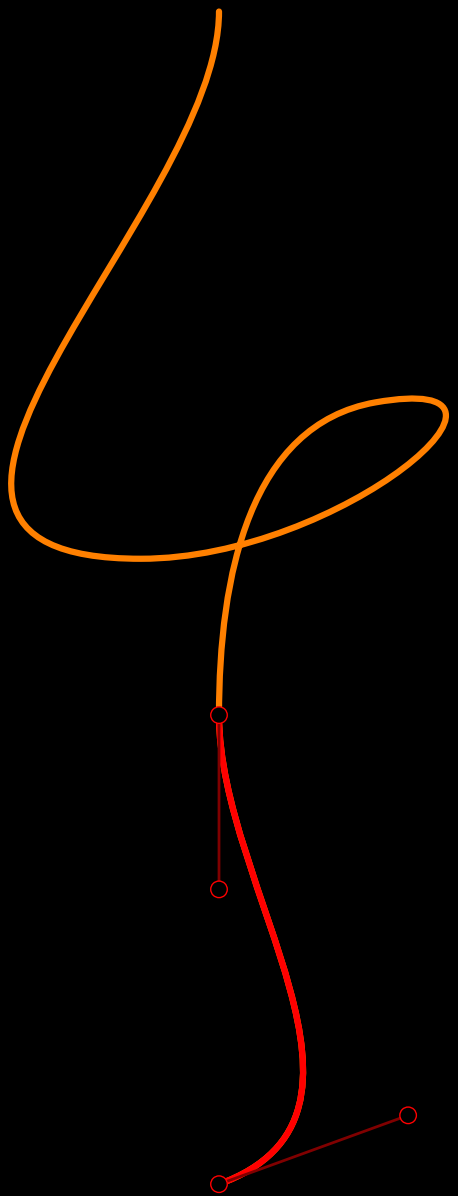




Пути в METAPOST

Пути в METAPOST составлены из сегментов — кривых Безье:

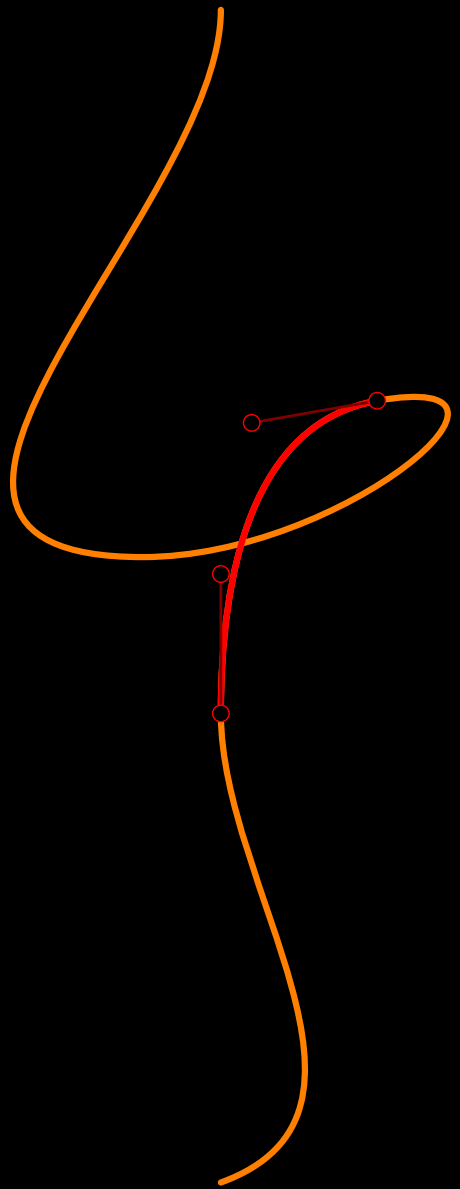
Пути в METAPOST



Пути в METAPOST составлены из сегментов — кривых Безье:

первый сегмент

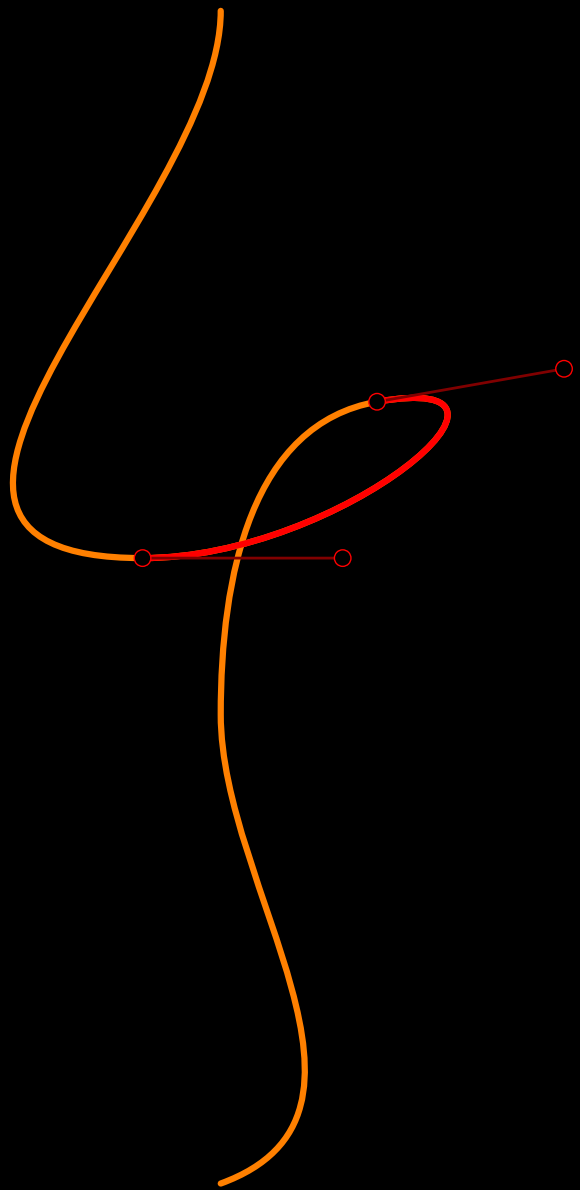
Пути в METAPOST



Пути в METAPOST составлены из сегментов — кривых Безье:

второй сегмент

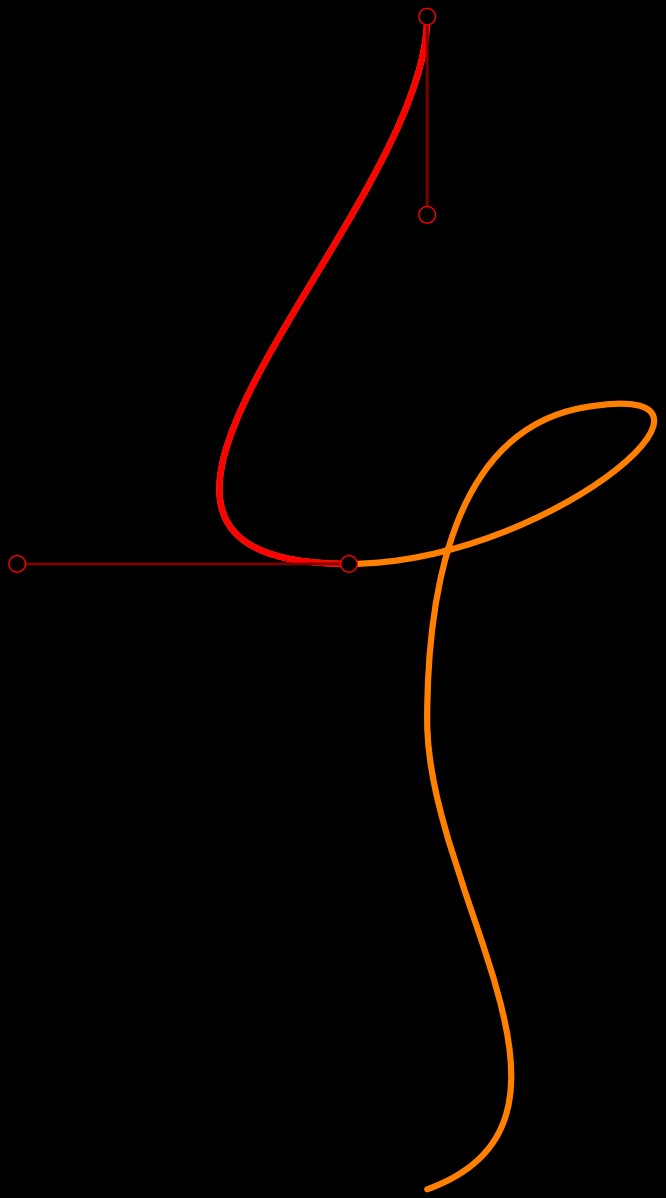
Пути в METAPOST



Пути в METAPOST составлены из сегментов — кривых Безье:

третий сегмент

Пути в METAPOST



Пути в METAPOST составлены из сегментов — кривых Безье:

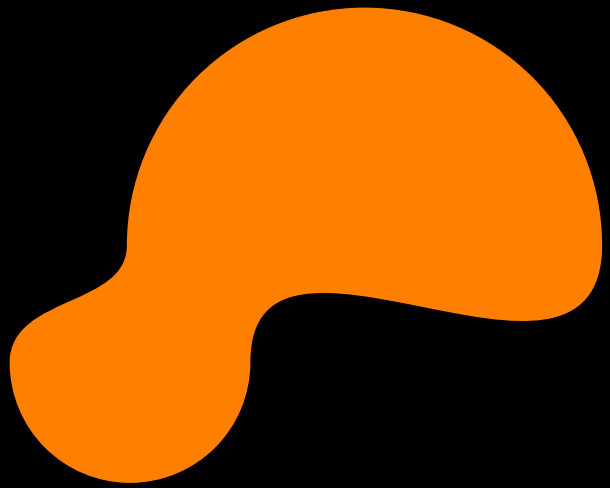
четвёртый сегмент

Использование путей в METAPOST

Пути можно:

- начертить
- закрасить
- использовать как границу рисунка (to clip)

Использование путей в METAPOST



Пути можно:

- начертить
- закрасить
- использовать как границу рисунка (to clip)

только *циклические!*

Использование путей в METAPOST

Пути можно:

- начертить
- закрасить
- использовать как границу рисунка (to clip)

только *циклические!*

Циклические пути

Путь называется *замкнутым*, если его начальная и конечная точки совпадают.

Замкнутый путь, имеющий атрибут цикличности, называется *циклическим*.

Лишь циклические пути могут быть закрашены или использованы как граница рисунка.

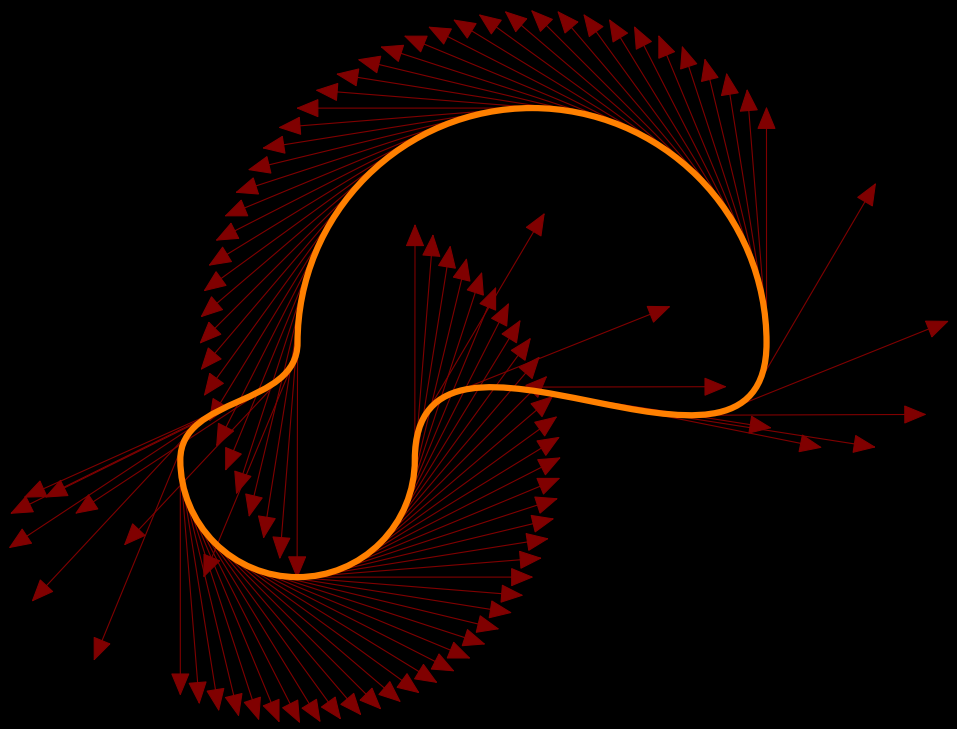
Атрибут цикличности устанавливается при создании пути.

Число вращения

Для замкнутых путей определяется *число вращения*.

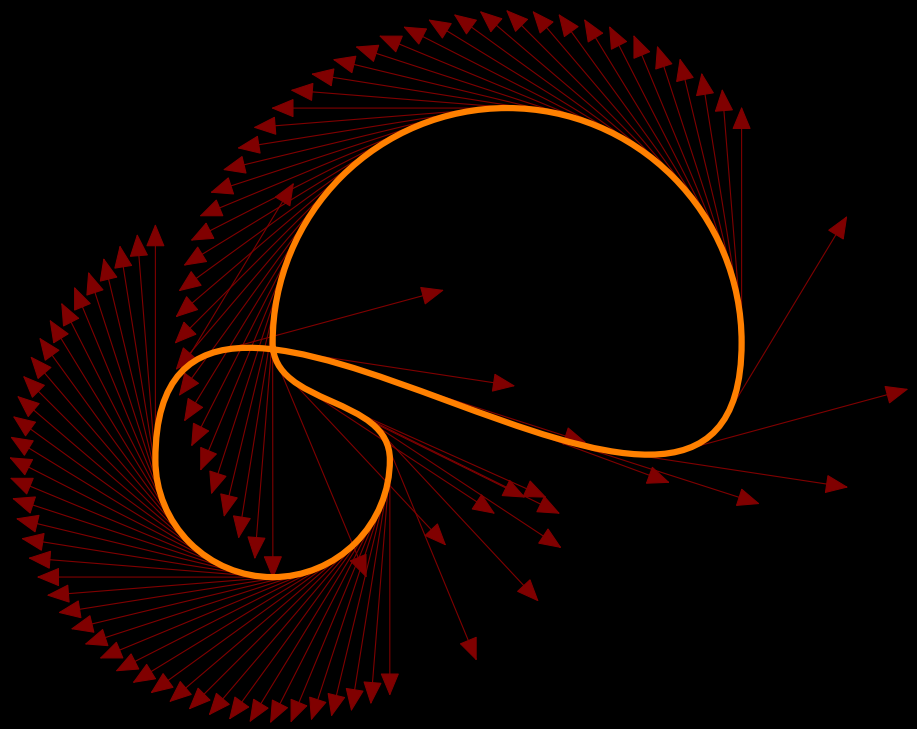
Это количество полных оборотов, которые сделает единичный касательный вектор, если точка касания полностью обойдёт путь.

Число вращения



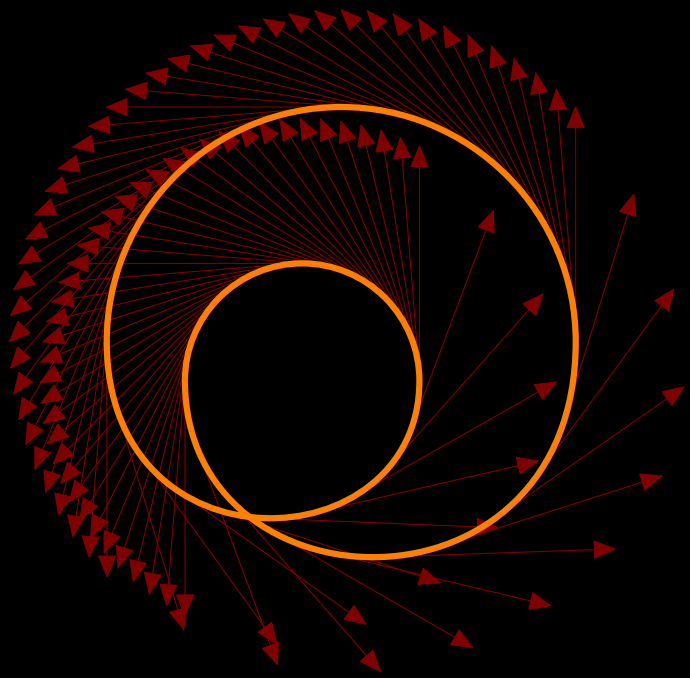
число вращения 1

Число вращения



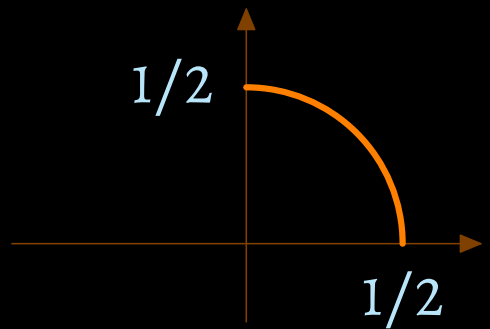
число вращения 0

Число вращения



число вращения 2

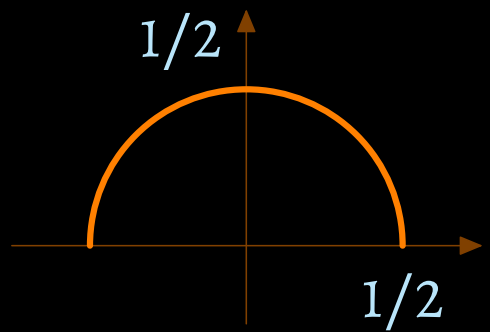
Предопределённые переменные — пути



`quartercircle`

четверть окружности радиуса $1/2$;
центр в начале координат

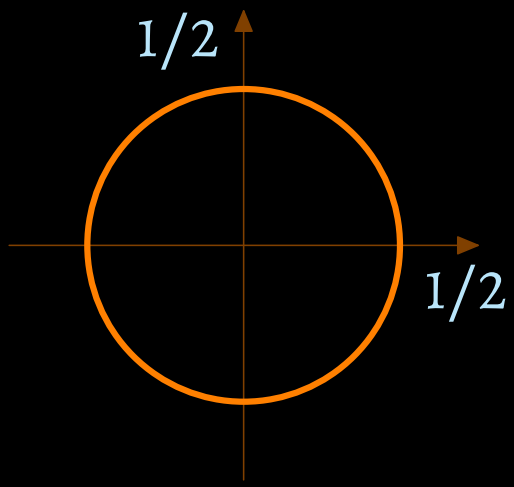
Предопределённые переменные — пути



`halfcircle`

половина окружности радиуса $1/2$;
центр в начале координат

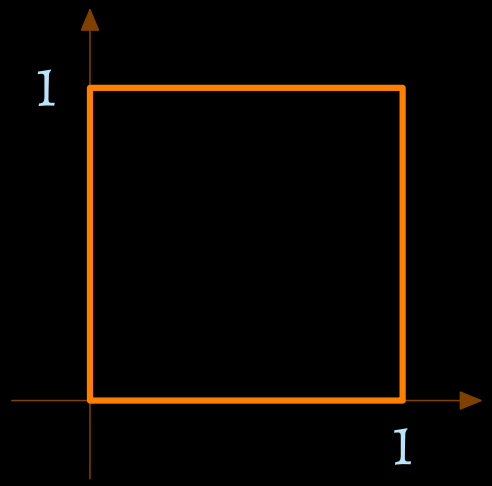
Предопределённые переменные — пути



`fullcircle`

окружность радиуса $1/2$;
центр в начале координат
этот путь циклический

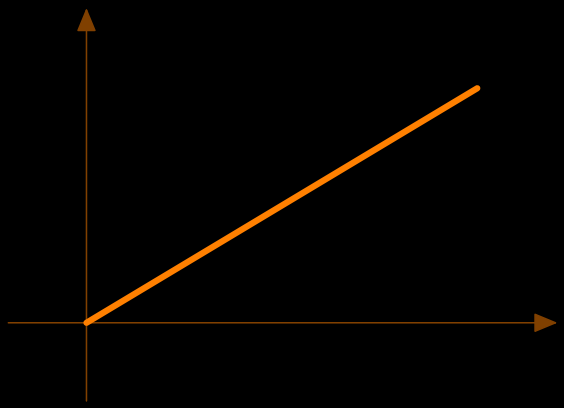
Предопределённые переменные — пути



`unitsquare`

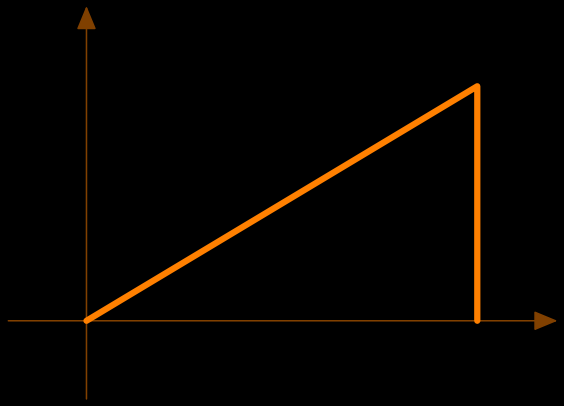
единичный квадрат;
левый нижний угол в начале координат
этот путь циклический

Ломаные



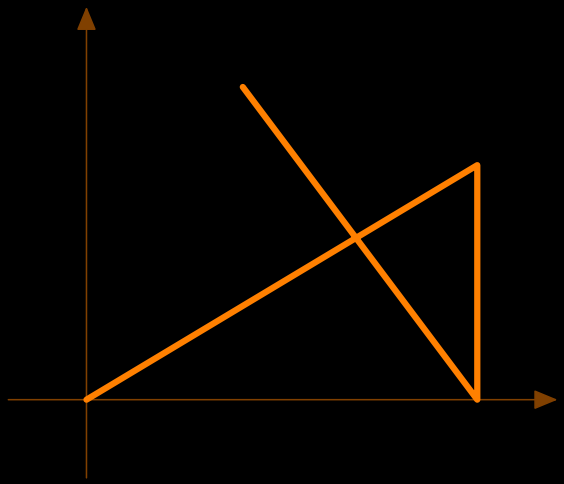
```
origin
    --(2.5cm, 1.5cm)
```

Ломаные



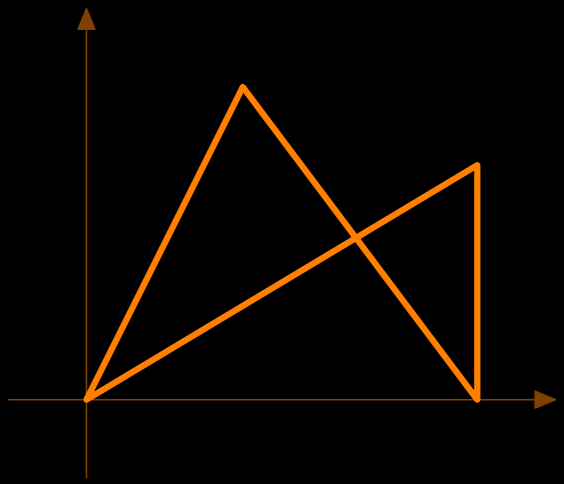
```
origin
    --(2.5cm, 1.5cm)
    --(2.5cm, 0)
```

Ломаные



```
origin
    --(2.5cm, 1.5cm)
    --(2.5cm, 0)
    --(cm, 2cm)
```

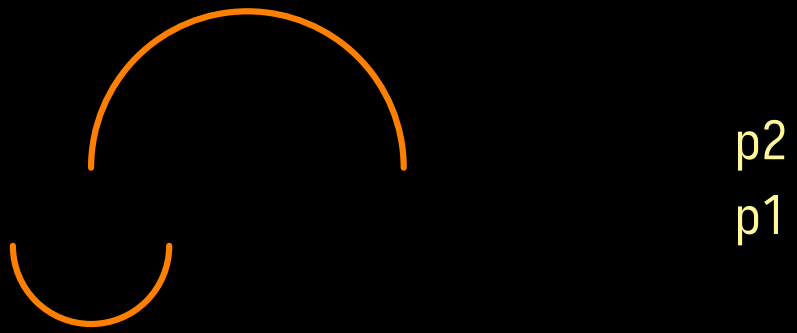
Ломаные



```
origin
    --(2.5cm, 1.5cm)
    --(2.5cm, 0)
    --(cm, 2cm)
    --cycle
```

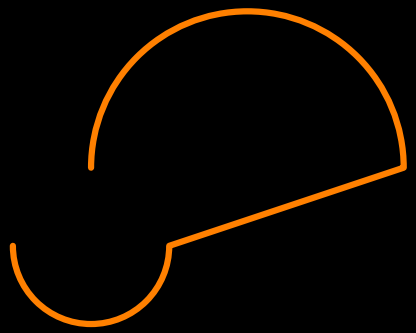
Связка путей «--»

Из путей p_1 и p_2 образуется новый путь: он состоит из пути p_1, прямолинейного отрезка, соединяющего конец p_1 с началом p_2, и пути p_2.



Связка путей «--»

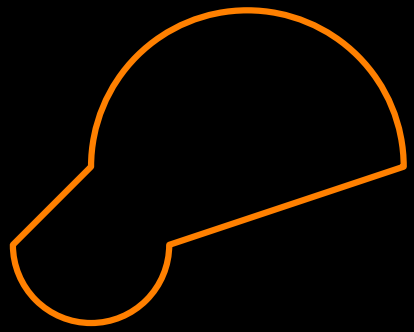
Из путей p_1 и p_2 образуется новый путь: он состоит из пути p_1, прямолинейного отрезка, соединяющего конец p_1 с началом p_2, и пути p_2.



```
p1--p2
```

Связка путей «--»

Из путей p_1 и p_2 образуется новый путь: он состоит из пути p_1, прямолинейного отрезка, соединяющего конец p_1 с началом p_2, и пути p_2.

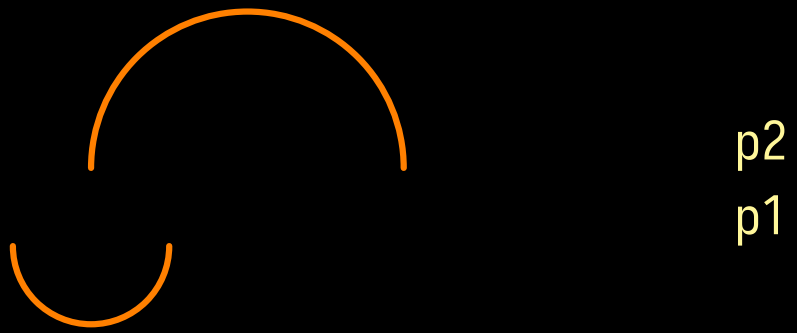


```
p1--p2--cycle
```

Связка путей «. .»

Из путей p_1 и p_2 образуется новый путь: он состоит из пути p_1, криволинейного отрезка, гладко соединяющего конец p_1 с началом p_2, и пути p_2.

Детали того, как строится криволинейная связка, довольно сложны.



Связка путей «..»

Из путей p_1 и p_2 образуется новый путь: он состоит из пути p_1, криволинейного отрезка, гладко соединяющего конец p_1 с началом p_2, и пути p_2.

Детали того, как строится криволинейная связка, довольно сложны.

p1..p2

Связка путей «..»

Из путей p_1 и p_2 образуется новый путь: он состоит из пути p_1, криволинейного отрезка, гладко соединяющего конец p_1 с началом p_2, и пути p_2.

Детали того, как строится криволинейная связка, довольно сложны.

```
p1..p2..cycle
```

Связка путей «&»

Из путей p_1 и p_2 образуется новый путь: он получается присоединением p_2 к p_1.

Конец p_1 должен совпадать с началом p_2, иначе ошибка:

```
! Paths don't touch; `&' will be changed to `..'.
```

METAPOST заменит опасную связку «&» на безопасную «..».

Связка путей «&»

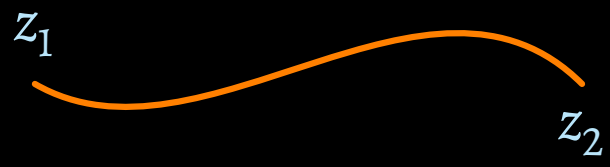
Из путей p_1 и p_2 образуется новый путь: он получается присоединением p_2 к p_1.

Конец p_1 должен совпадать с началом p_2, иначе ошибка:

```
! Пути не соединяются; `&' будет заменён на `..'.
```

METAPOST заменит опасную связку «&» на безопасную «..».

Указание направлений




```
z1{dir -30}
    ..{dir -45}z2
```

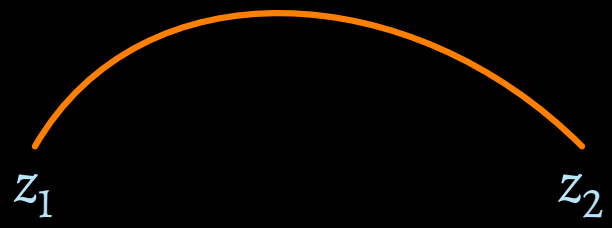
Указание направлений


```
z1{dir 0}
    ..{dir -45}z2
```

Указание направлений

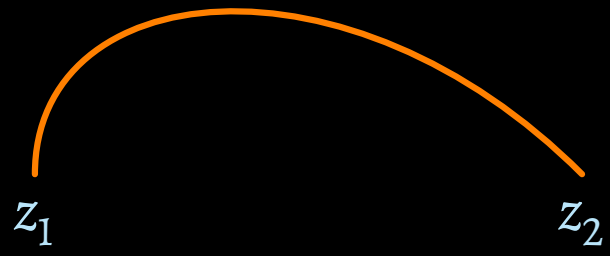

```
z1{dir 30}
    ..{dir -45}z2
```

Указание направлений




```
z1{dir 60}
    ..{dir -45}z2
```

Указание направлений




```
z1{dir 90}
    ..{dir -45}z2
```

Параметрическая длина пути

Оператор `length` вычисляет так называемую параметрическую длину. Это целое число — количество сегментов кривых Безье, из которых составлен путь:

$$\texttt{length}\ p$$

Параметризация пути

Каждому числовому значению $t \in [0; l]$, где l — параметрическая длина пути p, можно поставить в соответствие точку на этом пути.

Пусть $\alpha = [t]$ (целая часть t), $\beta = \{t\}$ (дробная часть t).

Возьмём сегмент p_α кривой Безье пути p, имеющий номер α. Обозначим как $z_{0,1,2,3}$ контрольные точки этого сегмента. Возьмём точку, заданную значением β в полиноме Бернштейна для этого сегмента:

$$z(\beta) = (1-\beta)^3 z_0 + 3(1-\beta)^2 \beta z_1 + 3(1-\beta)\beta^2 z_1 + \beta^3 z_3.$$

Полученная точка $z(\beta)$ и есть искомая точка на пути p, отвечающая значению параметра t.

Оператор `point of`

Оператор `point of` делает все эти вычисления:

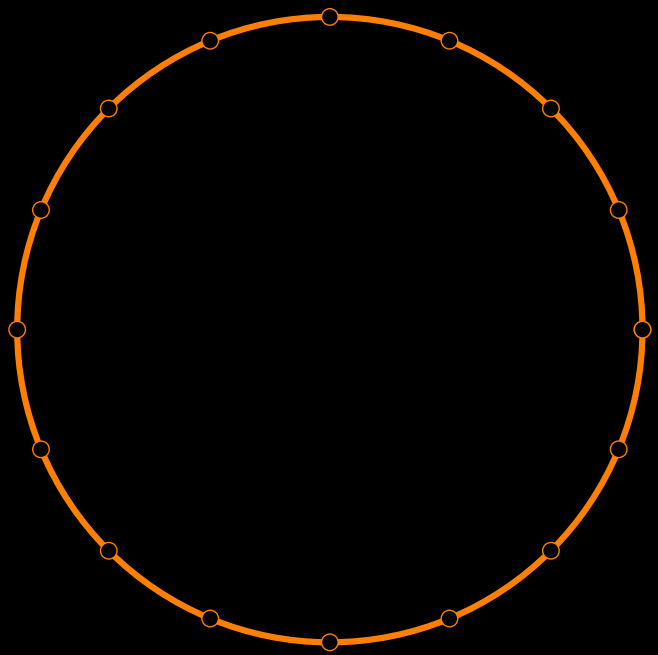
$$\texttt{point } t \texttt{ of } p$$

Значения t, расположенные вне отрезка $[0; l]$, сдвигаются в нужную сторону, пока не окажутся в пределах этого отрезка.

Таким образом, `point -1 of` p даст тот же результат, что и `point 0 of` p.

`point infinity of` p — это всегда конечная точка пути. (`infinity` — предопределённая числовая переменная, самое большое число, представимое в METAPOST.)

Оператор `point of`



На рисунке на пути $p =$ `fullcircle` расставлены отметки в точках `point` t `of` p с шагом $\frac{1}{4}$.

Если приглядеться, видно, что отметки расположены не совсем равномерно.

Пересечение путей

Оператор `intersectionpoint` вычисляет точку пересечения двух путей p_1 и p_2:

$$p_1 \texttt{ intersectionpoint } p_2$$

Если пути не пересекаются, возникает ошибка:

```
! The paths don't intersect.
```

Пересечение путей

Оператор `intersectionpoint` вычисляет точку пересечения двух путей p_1 и p_2:

$$p_1 \texttt{ intersectionpoint } p_2$$

Если пути не пересекаются, возникает ошибка:

```
! Пути не пересекаются.
```

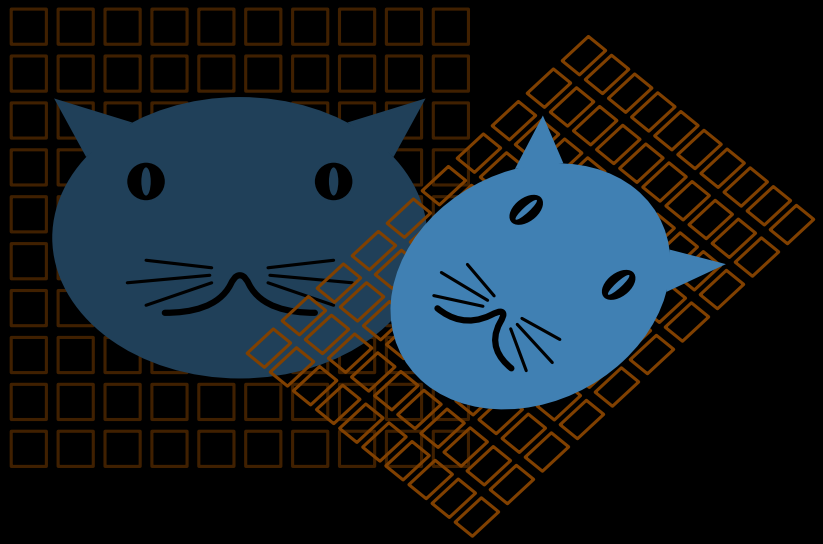
Пересечение путей

Если точек пересечения путей p_1 и p_2 несколько, вычисляется по возможности та из них, которая лежит ближе к началам каждого из путей.

Поскольку это пожелание по своей сути противоречиво, алгоритм поиска точки пересечения путей определённым образом упорядочивает точки пересечения, и выдаёт первую из них (в смысле этого упорядочения).

Принцип упорядочения не самый очевидный, а детали мы опустим.

Вообще говоря, оператор `intersectionpoint` не коммутативен.



Определение аффинных преобразований плоскости

Аффинные преобразования плоскости задаются шестёркой чисел T_{xx}, T_{xy}, T_x, T_{yx}, T_{yy}, T_y при помощи формул:

$$x' = T_{xx}\,x + T_{xy}\,y + T_x$$
$$y' = T_{yx}\,x + T_{yy}\,y + T_y$$

Определение аффинных преобразований плоскости

Аффинные преобразования плоскости задаются шестёркой чисел T_{xx}, T_{xy}, T_x, T_{yx}, T_{yy}, T_y при помощи формул:

$$x' = T_{xx}\,x + T_{xy}\,y + T_x$$
$$y' = T_{yx}\,x + T_{yy}\,y + T_y$$

Определение аффинных преобразований плоскости

Аффинные преобразования плоскости задаются шестёркой чисел T_{xx}, T_{xy}, T_x, T_{yx}, T_{yy}, T_y при помощи формул:

$$x' = T_{xx}\,x + T_{xy}\,y + T_x$$
$$y' = T_{yx}\,x + T_{yy}\,y + T_y$$

Новые координаты точки выражаются через старые линейным образом.

Определение аффинных преобразований плоскости

Аффинные преобразования плоскости задаются шестёркой чисел T_{xx}, T_{xy}, T_x, T_{yx}, T_{yy}, T_y при помощи формул:

$$x' = T_{xx}\,x + T_{xy}\,y + T_x$$
$$y' = T_{yx}\,x + T_{yy}\,y + T_y$$

Новые координаты точки выражаются через старые линейным образом.

Невырожденность аффинных преобразований

Невырожденным называется аффинное преобразование, если

$$T_{xx}T_{yy} - T_{xy}T_{yx} \neq 0.$$

Свойства невырожденных аффинных преобразований

- прямые преобразуются в прямые
- параллельные прямые преобразуются в параллельные прямые
- вполне определяется тремя точками — вершинами треугольника, и тремя их образами

Кошки Арнольда

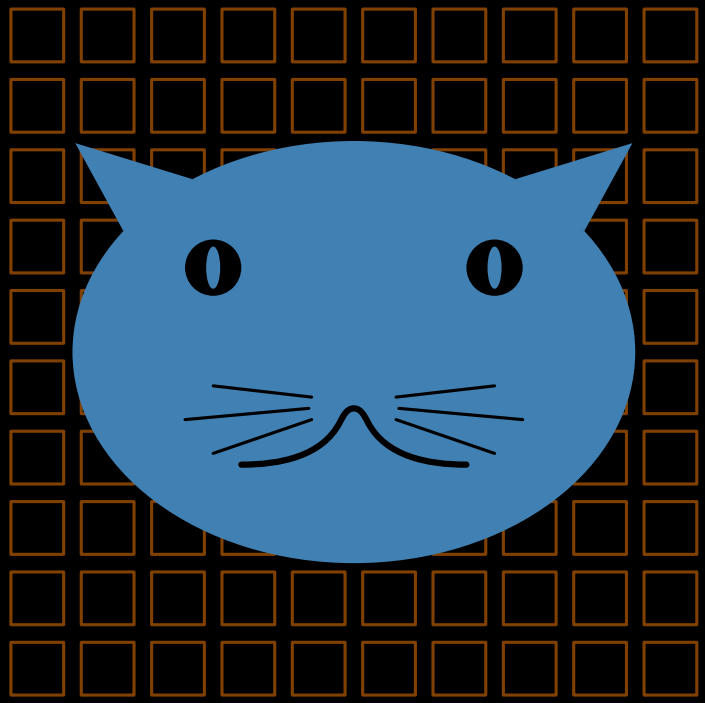
Академик В. И. Арнольд очень любил кошек.

Он придумал иллюстрировать преобразования плоскости при помощи кошачьих морд.

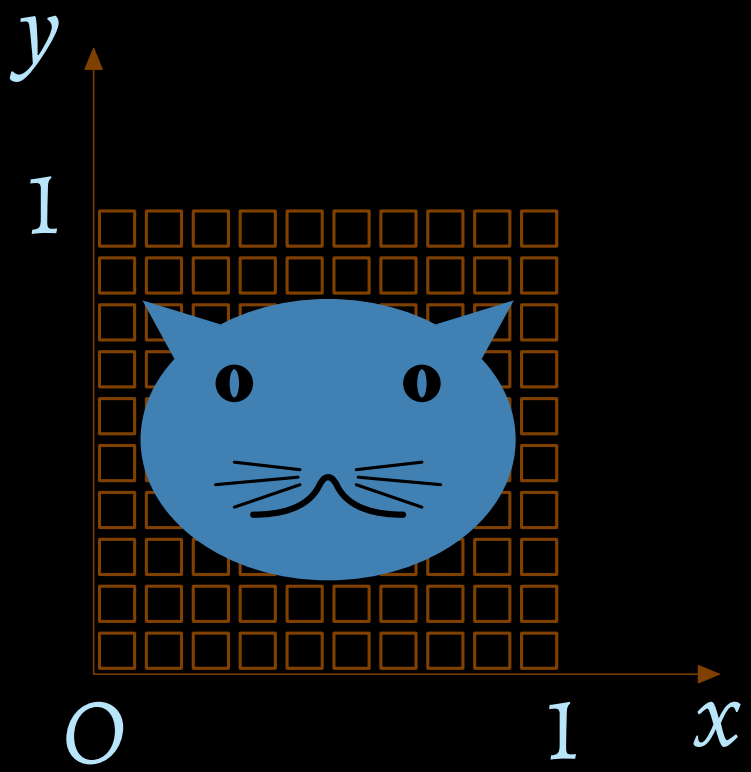
Кошки Арнольда

Академик В. И. Арнольд очень любил кошек.

Он придумал иллюстрировать преобразования плоскости при помощи кошачьих морд.

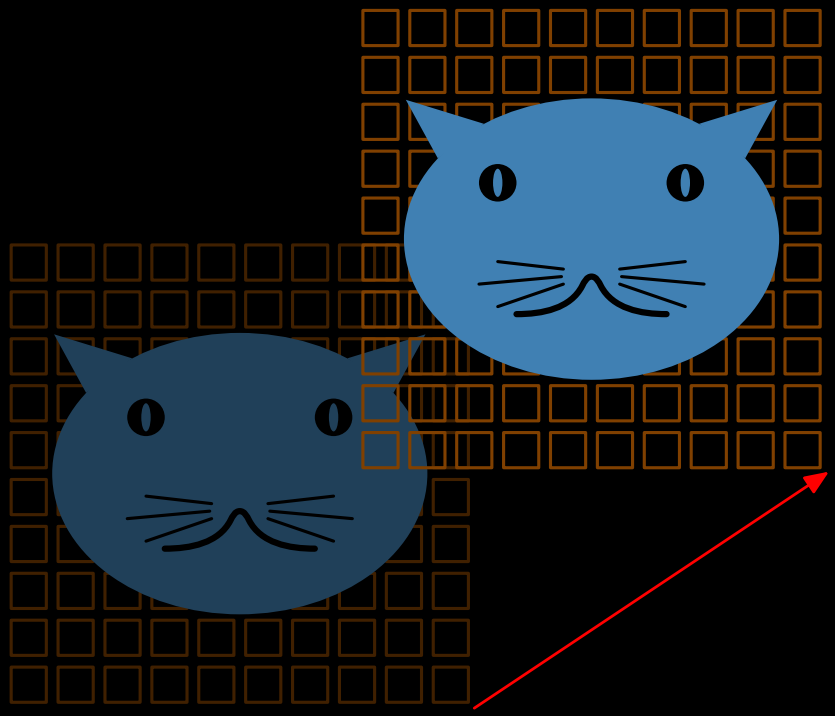


Тождественное преобразование




```
identity
```

Сдвиг (параллельный перенос)

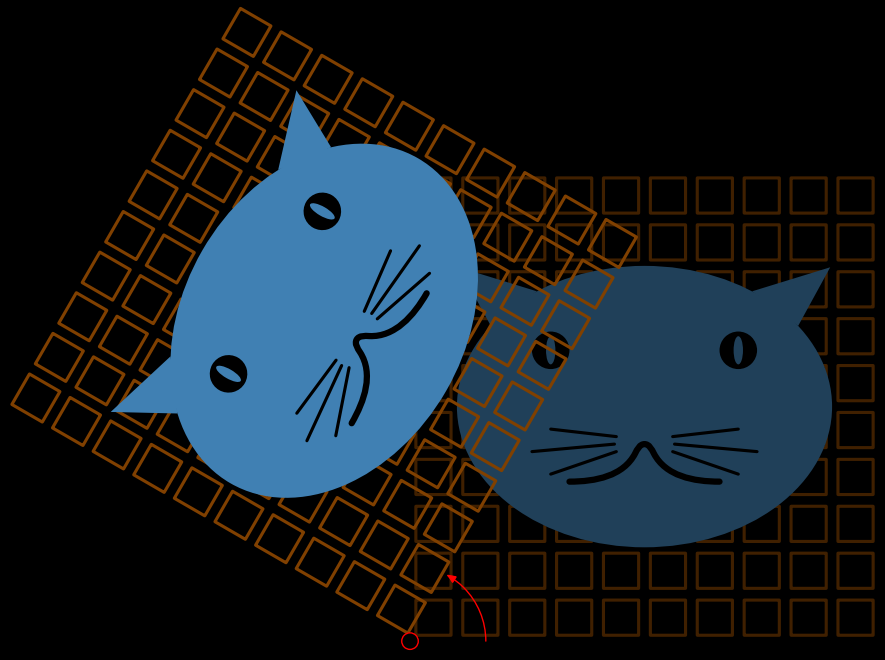


```
identity shifted (.75, .5)
```

Сдвиг (параллельный перенос)

$$(x, y)\ \texttt{shifted}\ (a, b)$$
$$\equiv$$
$$(x + a, y + b)$$

Поворот

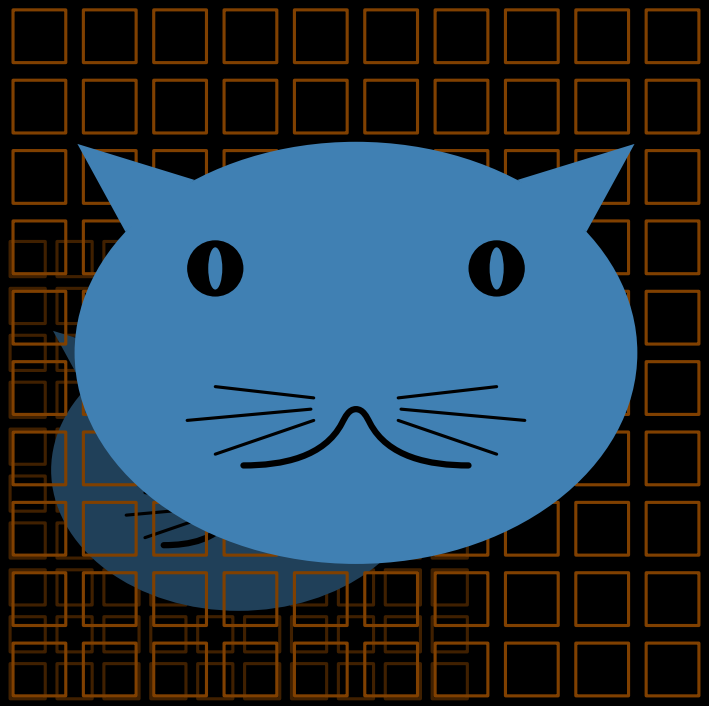


```
identity rotated 60
```

Поворот

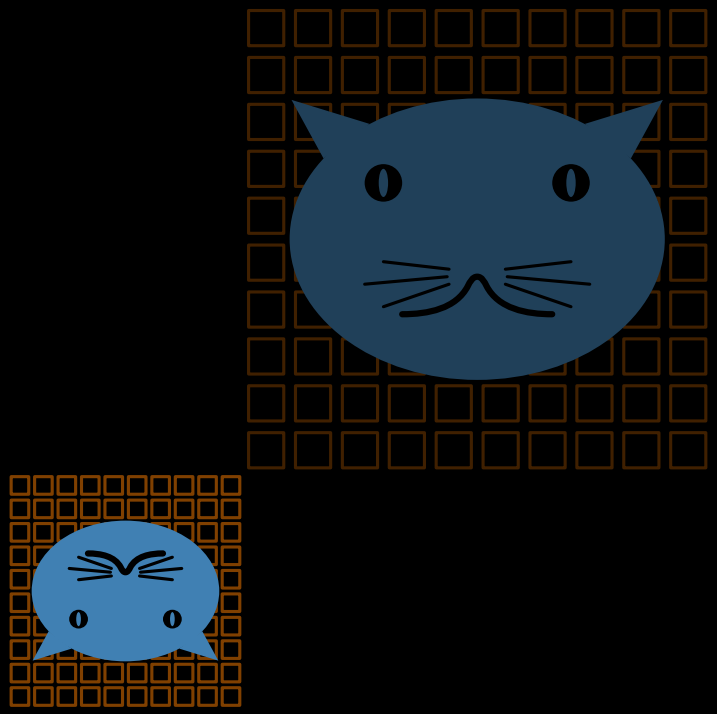
$$(x, y) \ \texttt{rotated} \ \vartheta$$
$$\equiv$$
$$(x \cos \vartheta - y \sin \vartheta, x \sin \vartheta + y \cos \vartheta)$$

Растяжение



identity scaled 1.5

Растяжение

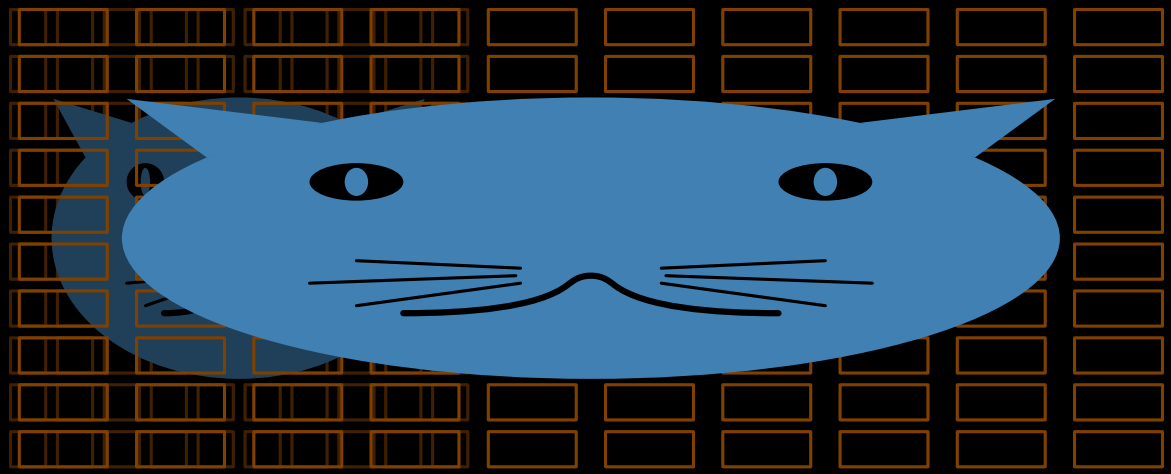


```
identity scaled -.5
```

Растяжение

$$(x, y) \texttt{ scaled } k$$
$$\equiv$$
$$(kx, ky)$$

Анаморфотные преобразования



```
identity xscaled 2.5
```

Анаморфотные преобразования

$$(x, y)\ \texttt{xscaled}\ k$$
$$\equiv$$
$$(kx, y)$$

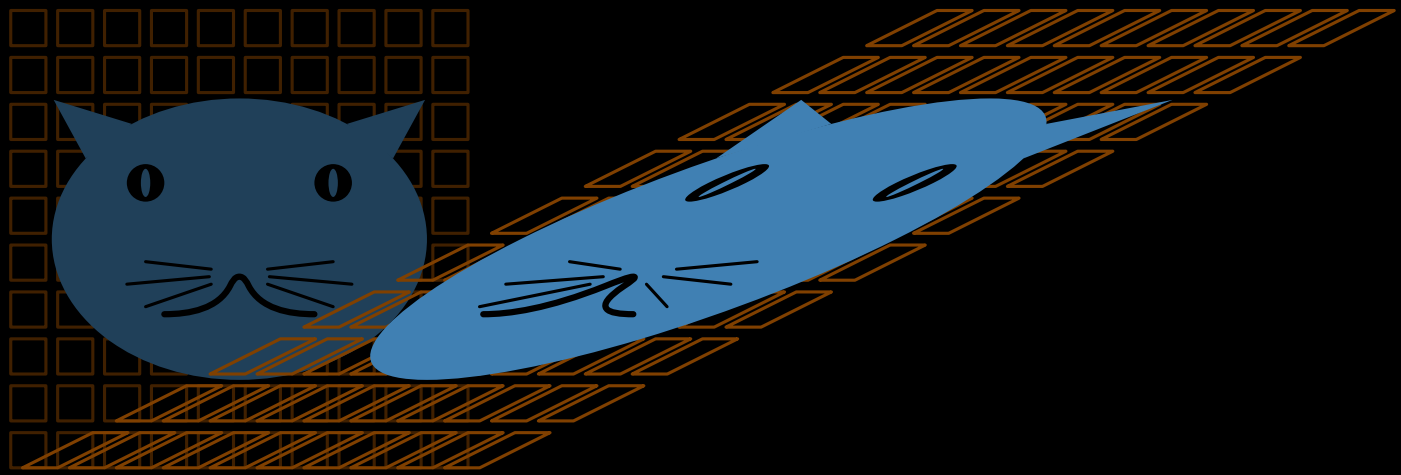
Анаморфотные преобразования

```
identity yscaled .25
```

Анаморфотные преобразования

$$(x, y)\ \texttt{yscaled}\ k$$
$$\equiv$$
$$(x, ky)$$

Наклон

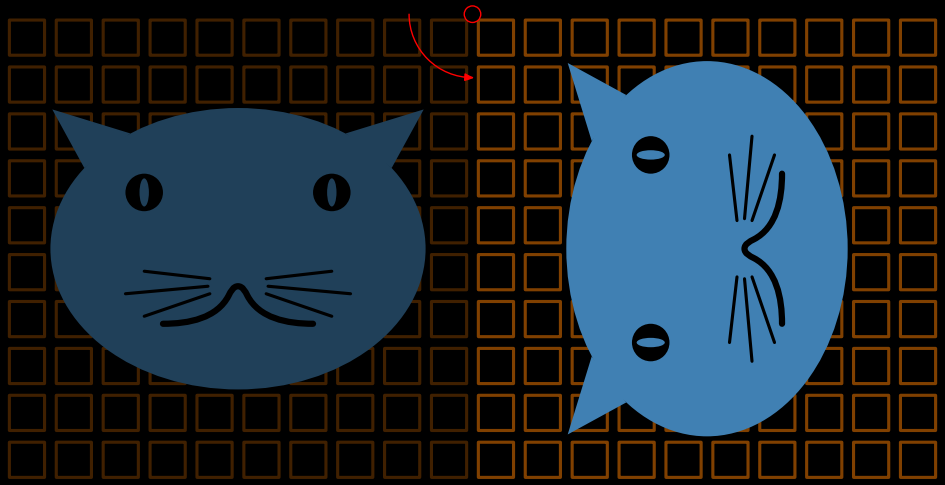


identity slanted 2

Наклон

$$(x, y) \texttt{ slanted } k$$
$$\equiv$$
$$(x + ky, y)$$

Поворот вокруг заданной точки



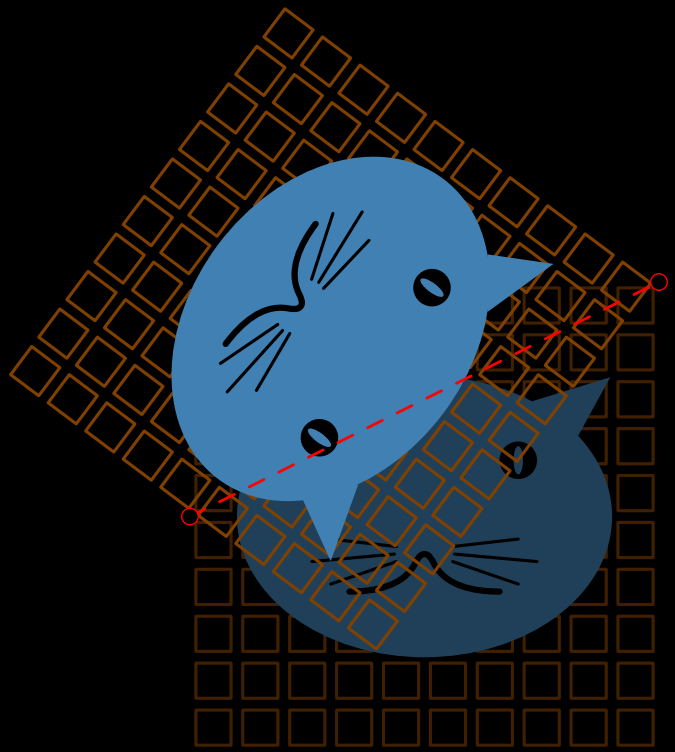
```
identity
```

```
rotatedabout((1, 1), 90)
```

Поворот вокруг заданной точки

$$p \ \texttt{rotatedabout}(z, \ \vartheta)$$
$$\equiv$$
$$p \ \texttt{shifted} \ -z \ \texttt{rotated} \ \vartheta \ \texttt{shifted} \ z$$

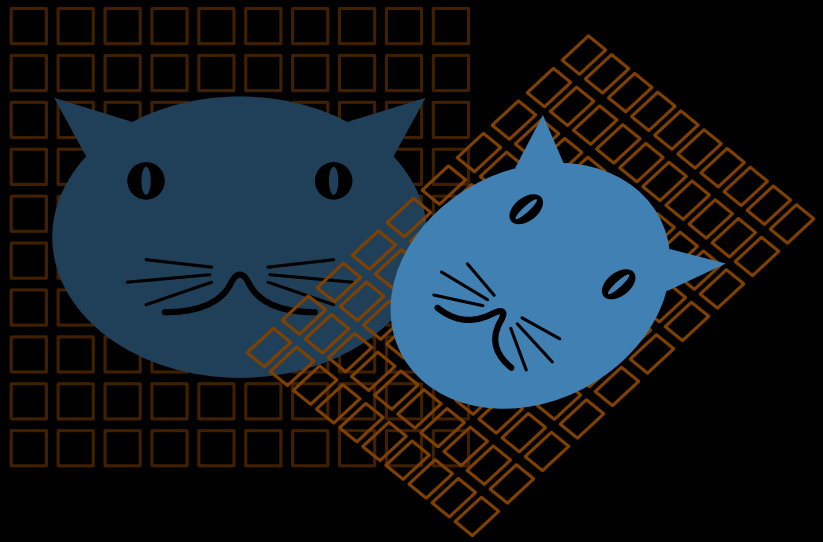
Осевая симметрия (зеркальное отражение)



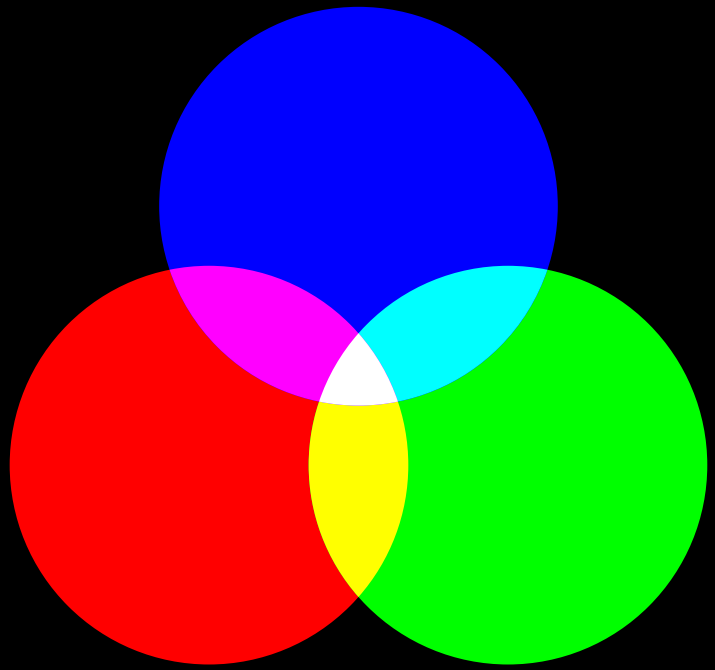
```
identity
```

```
reflectedabout((0, .5), (1, 1))
```

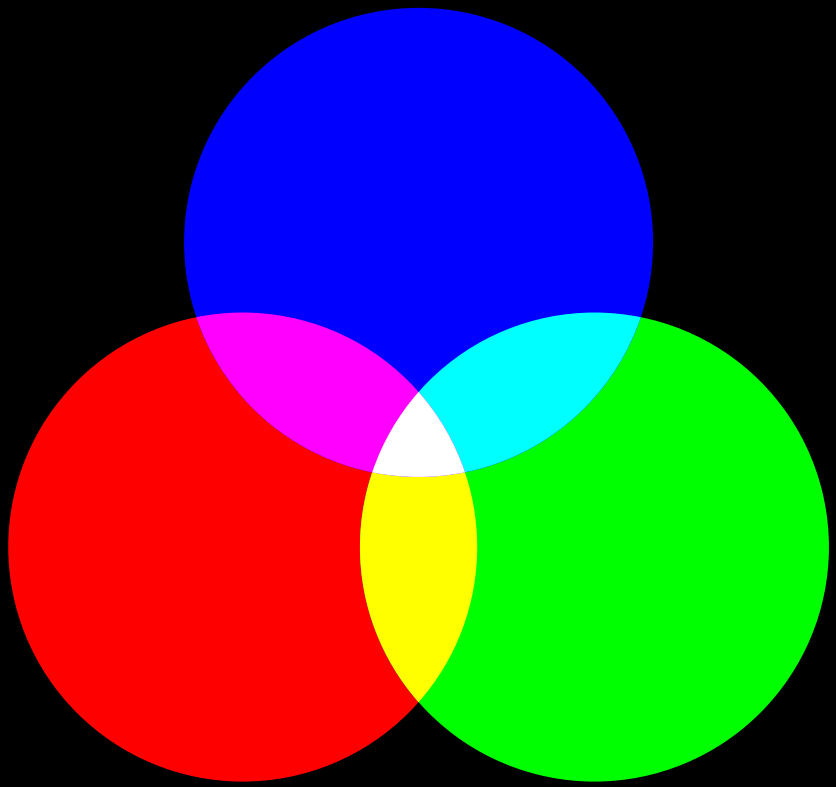
Аффинное преобразование общего вида



```
identity scaled .8 rotated -30
slanted .5 shifted (.5, .25)
```



Цветовая схема RGB

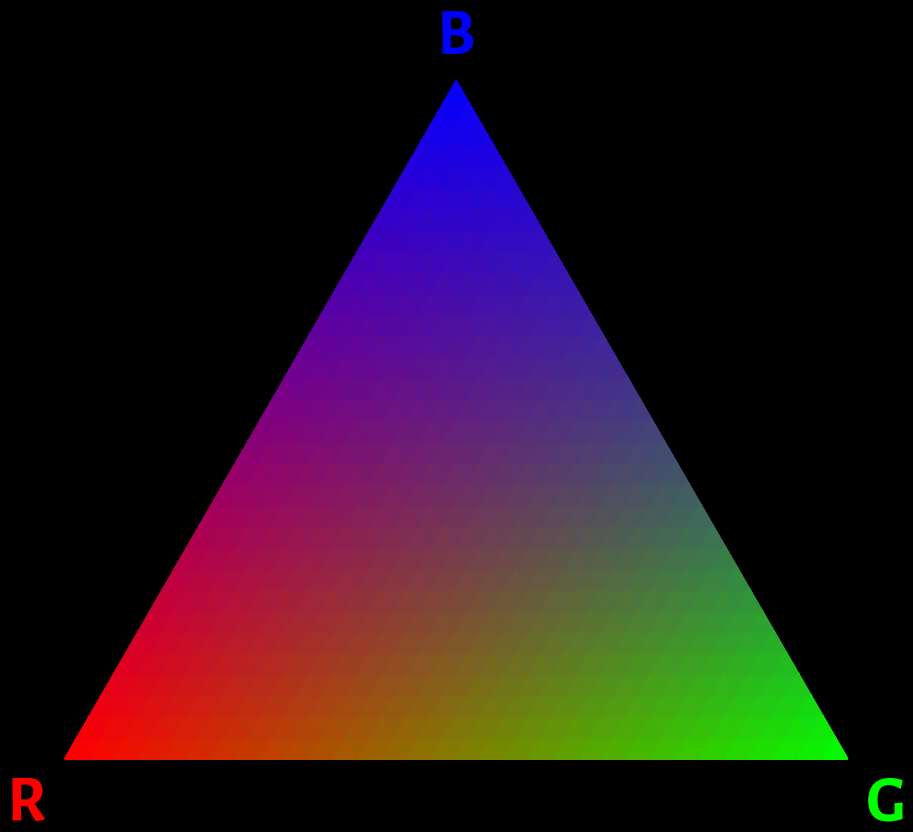


В METAPOST возможно цветное рисование. Для представления цветов предусмотрен тип `color`.

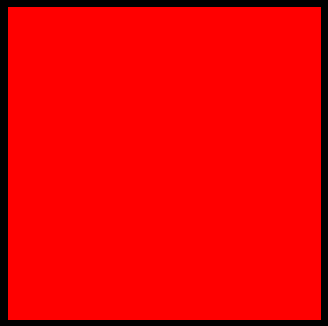
В METAPOST принята трёхкомпонентная цветовая схема RGB. Каждый цвет представляется тройкой чисел (R, G, B), каждое из которых заключено в отрезке $[0; 1]$.

Каждый цвет в модели RGB считается смесью трёх «основных» цветов. Числа R, G, B — количества соответственно красного, зелёного и синего цветов в такой смеси.

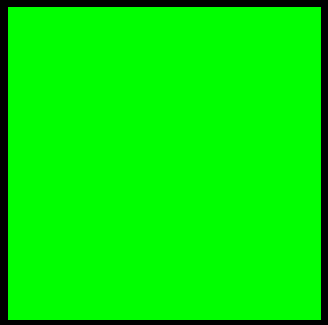
Цветовой треугольник



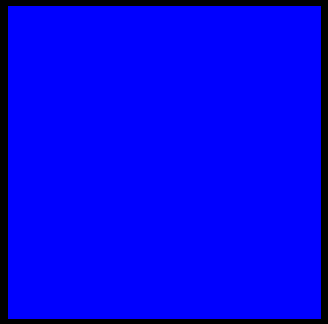
Предопределённые цветовые переменные



`red` ≡ `(1, 0, 0)`

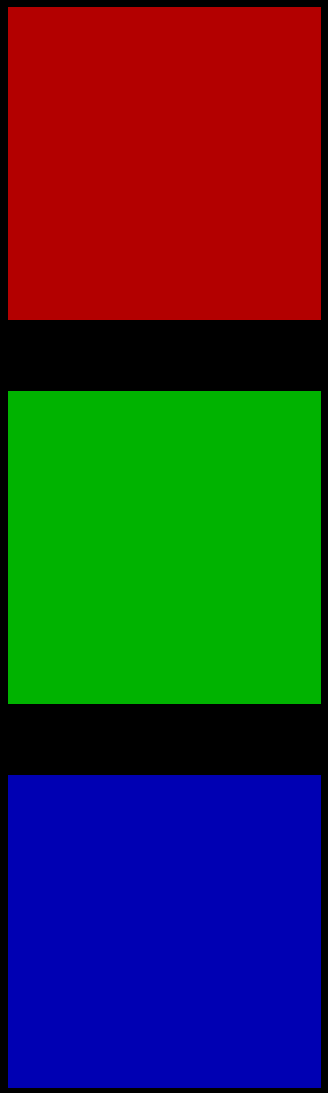


`green` ≡ `(0, 1, 0)`



`blue` ≡ `(0, 0, 1)`

Цветовые выражения

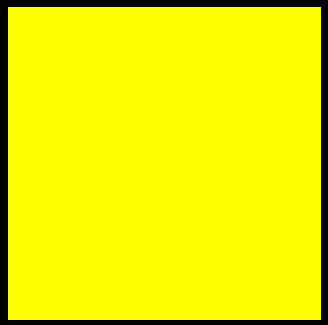


.7red

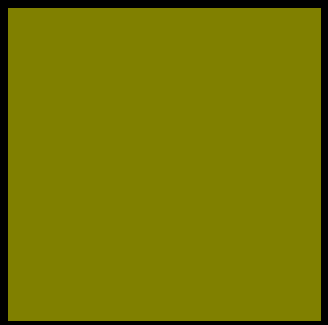
.7green

.7blue

Цветовые выражения



red+green



.5[red, green]

Определение цветовых переменных

```
color yellow, cyan, magenta;

yellow:=red+green;
cyan:=green+blue;
magenta:=blue+red;
```

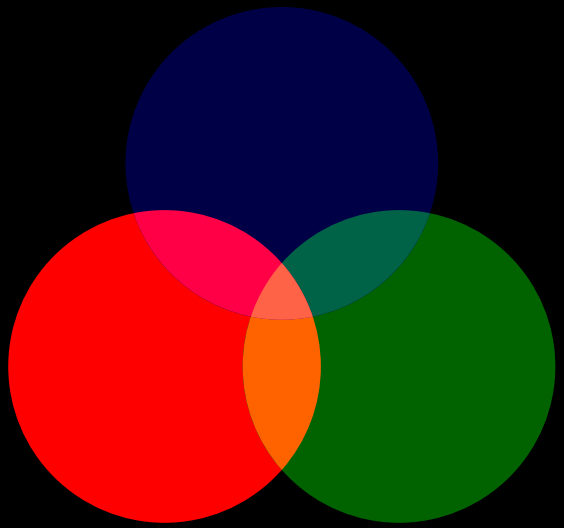
Определение цветовых переменных

```
color yellow, cyan, magenta;

yellow:=(1, 1, 0);
cyan:=(0, 1, 1);
magenta:=(1, 0, 1);
```

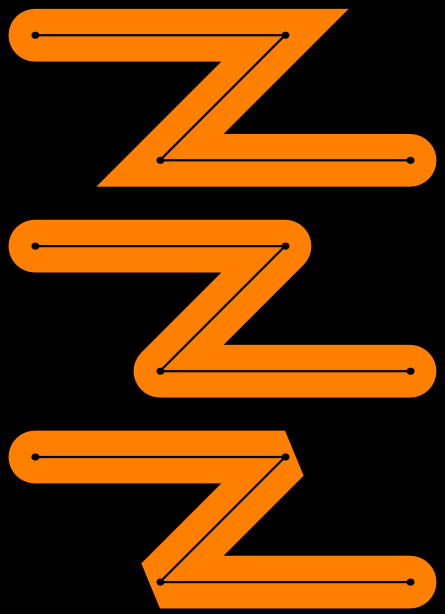
Декомпозиция цветов

Операторы `redpart`, `greenpart` и `bluepart` извлекают из цветового выражения соответственно красную, зелёную и синюю составляющую:

```
color tomato;
tomato=(1, .388235294117647, .278431372549002);
```



$$\texttt{redpart tomato} \equiv 1$$
$$\texttt{greenpart tomato} \equiv .38823$$
$$\texttt{bluepart tomato} \equiv .27843$$

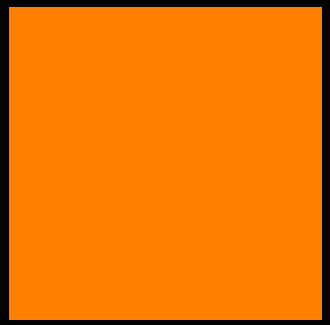


Рисование

В METAPOST рисование осуществляется командами `draw` и `fill`.

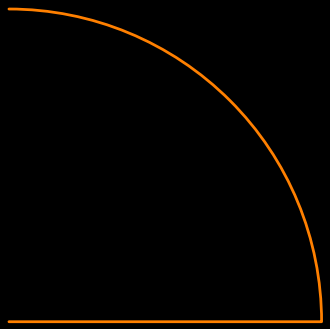
Примеры рисования

```
draw unitsquare scaled 2cm;
```



```
fill unitsquare scaled 2cm;
```

Примеры рисования



```
draw origin--quartercircle
    scaled 4cm;
```

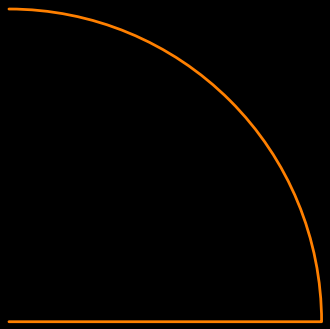
?

```
fill origin--quartercircle
    scaled 4cm;
```

Ошибка:

```
! Not a cycle.
```

Примеры рисования



```
draw origin--quartercircle
    scaled 4cm;
```

?

```
fill origin--quartercircle
    scaled 4cm;
```

Ошибка:

```
! Не цикл.
```

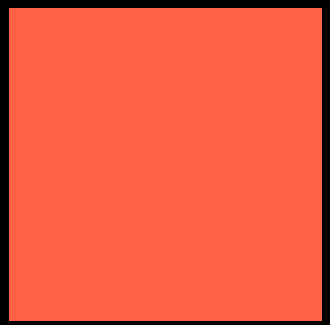
Цвет по умолчанию

Для рисования в METAPOST используется чёрный цвет, если не указан иной.

В настоящем руководстве по понятным причинам в качестве цвета по умолчанию выбран другой — оранжевый.

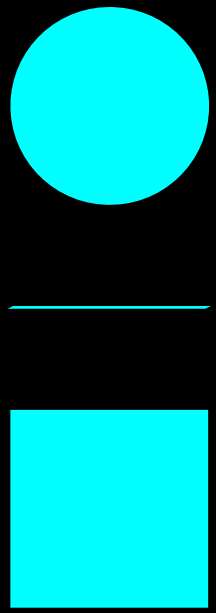
Цветное рисование

```
draw unitsquare scaled 2cm
    withcolor green;
```



```
fill unitsquare scaled 2cm
    withcolor (1, .388, .278);
% цвет «tomato»
```

Рисование различными перьями



pencircle

penrazor

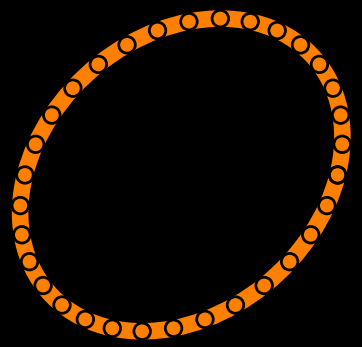
pensquare

Перо «pencircle»

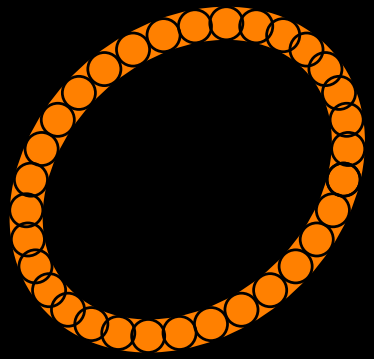
```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pencircle
        scaled 3bp;
```

```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pencircle
        scaled 6bp;
```

Перо «pencircle»



```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pencircle
        scaled 3bp;
```



```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pencircle
        scaled 6bp;
```

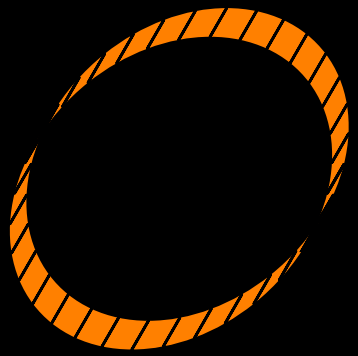
Перо «penrazor»

```
draw fullcircle
    slanted .25 scaled 2cm
    withpen penrazor scaled 6bp;
```

```
draw fullcircle
    slanted .25 scaled 2cm
    withpen penrazor
        rotated 60 scaled 6bp;
```

Перо «penrazor»

```
draw fullcircle
    slanted .25 scaled 2cm
    withpen penrazor scaled 6bp;
```



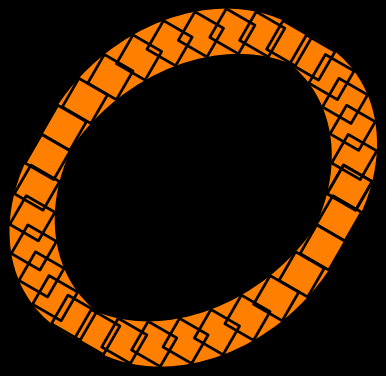
```
draw fullcircle
    slanted .25 scaled 2cm
    withpen penrazor
        rotated 60 scaled 6bp;
```

Перо «pensquare»

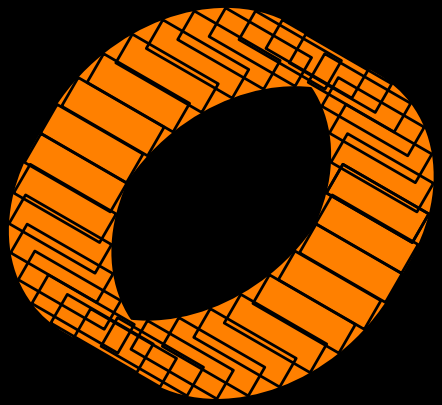
```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pensquare
        rotated 60
        scaled 6bp;
```

```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pensquare
        yscaled 3
        rotated 60
        scaled 6bp;
```

Перо «pensquare»

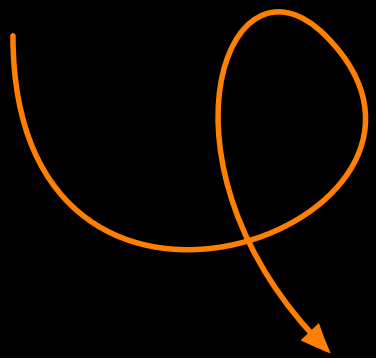


```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pensquare
        rotated 60
        scaled 6bp;
```



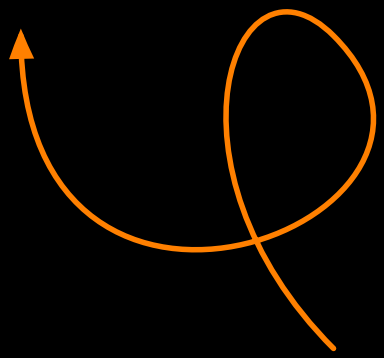
```
draw fullcircle
    slanted .25 scaled 2cm
    withpen pensquare
        yscaled 3
        rotated 60
        scaled 6bp;
```

Стрелки



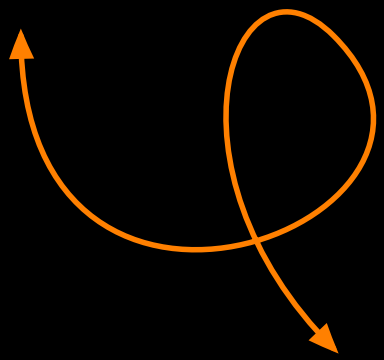
```
drawarrow
    (0, 2cm){down}
        ..(2cm, 2cm){dir 135}
        ..{dir -45}(2cm, 0)
    withpen pencircle
        scaled bp;
```

Стрелки



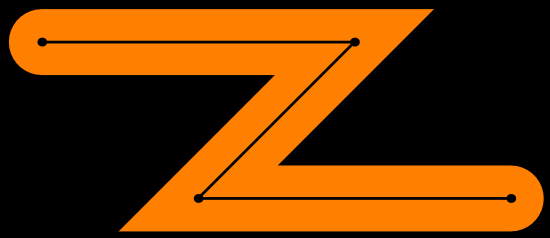
```
drawarrow
    reverse((0, 2cm){down}
        ..(2cm, 2cm){dir 135}
        ..{dir -45}(2cm, 0))
    withpen pencircle
        scaled bp;
```

Стрелки



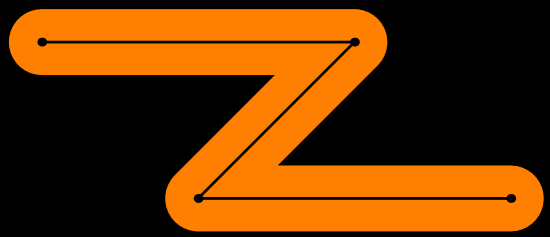
```
drawdblarrow
    (0, 2cm){down}
        ..(2cm, 2cm){dir 135}
        ..{dir -45}(2cm, 0)
    withpen pencircle
        scaled bp;
```

Острые углы



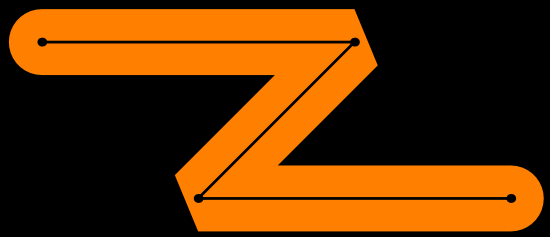
```
linejoin:=mitered;
draw p;
```

Острые углы



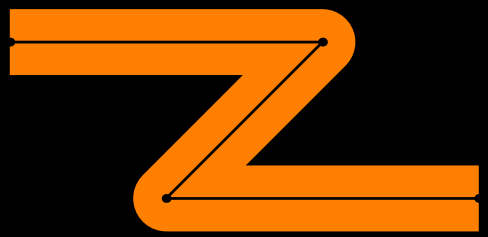
```
linejoin:=rounded;
draw p;
```

Острые углы



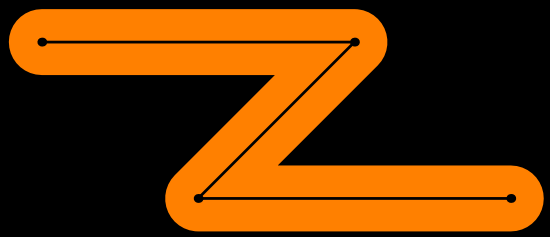
```
linejoin:=beveled;
draw p;
```

Концевые точки



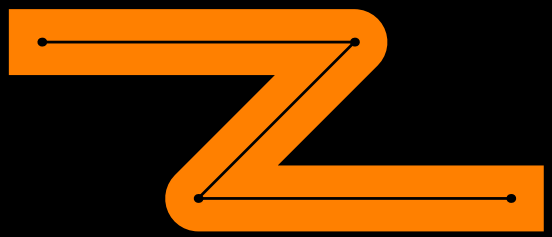
```
linecap:=butt;
draw p;
```

Концевые точки

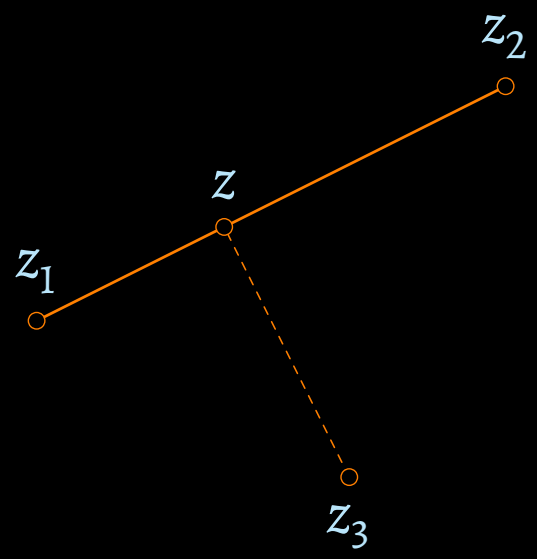


```
linecap:=rounded;
draw p;
```

Концевые точки



```
linecap:=squared;
draw p;
```

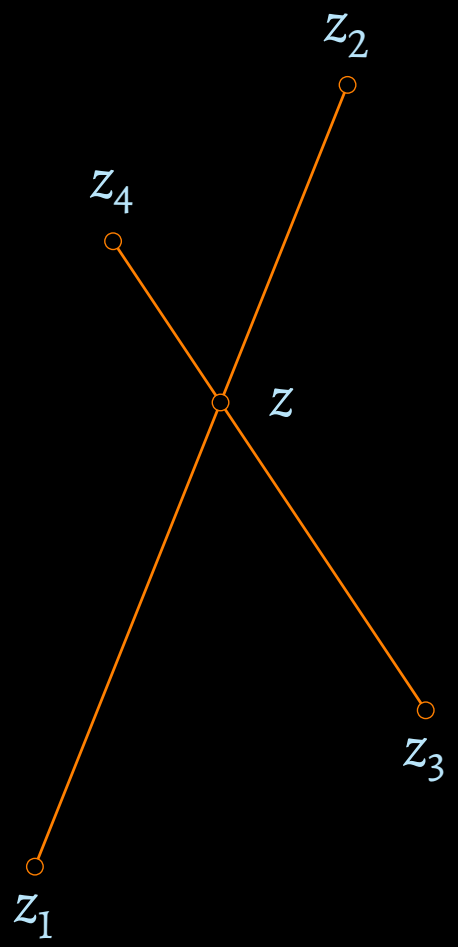


Уравнения в METAPOST

В METAPOST имеется уникальный встроенный механизм решения линейных алгебраических уравнений.

Наиболее распространённое применение этого механизма — вычисление точки пересечения двух прямых, каждая из которых задана парой точек.

Пересечение двух прямых



Утверждение о том, что точка z лежит на прямой $z_1 z_2$, алгебраически выражается так:

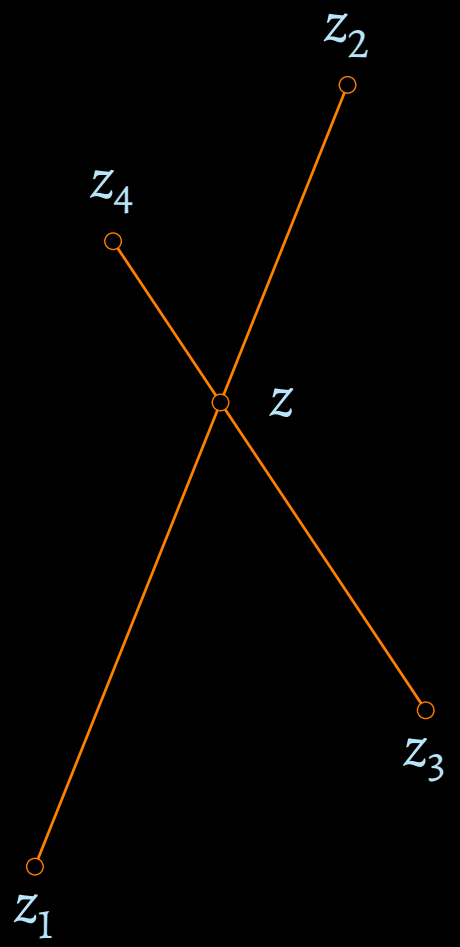
$$z = \lambda[z_1, z_2], \tag{1}$$

где λ — некоторое число. Аналогично, если точка z к тому же лежит на другой прямой $z_3 z_4$, получим

$$z = \mu[z_3, z_4]. \tag{2}$$

Достаточно найти λ, приравнивая правые части уравнений (1), (2), и подставить в (1), чтобы найти выражение для z.

Пересечение двух прямых



Утверждение о том, что точка z лежит на прямой z_1z_2, алгебраически выражается так:

$$z = (1 - \lambda)z_1 + \lambda z_2, \tag{1}$$

где λ — некоторое число. Аналогично, если точка z к тому же лежит на другой прямой z_3z_4, получим

$$z = (1 - \mu)z_3 + \mu z_4. \tag{2}$$

Достаточно найти λ, приравнивая правые части уравнений (1), (2), и подставить в (1), чтобы найти выражение для z.

Пересечение двух прямых

Векторное уравнение

$$(1-\lambda)z_1 + \lambda z_2 = (1-\mu)z_3 + \mu z_4$$

можно рассматривать как систему из двух скалярных уравнений с двумя неизвестными λ, μ:

$$\begin{cases} (1-\lambda)x_1 + \lambda x_2 = (1-\mu)x_3 + \mu x_4, \\ (1-\lambda)y_1 + \lambda y_2 = (1-\mu)y_3 + \mu y_4. \end{cases}$$

Выражение для λ получается довольно громоздким:

$$\lambda = \frac{(x_3 - x_1)(y_4 - y_3) - (x_4 - x_3)(y_3 - y_1)}{(x_2 - x_1)(y_4 - y_3) - (x_4 - x_3)(y_2 - y_1)},$$

а для z — и того хуже.

Пересечение двух прямых

Встроенный в METAPOST механизм решения линейных алгебраических уравнений избавляет нас от всех этих вычислений:

```
numeric lambda, mu;
pair z;
z=lambda[z1, z2]=mu[z3, z4];
```

Теперь пара `z` получит нужное значение.

Команда whatever

Обратите внимание на то, что из трёх найденных значений z, λ, μ нас в конечном итоге интересовало только z.

Переменные λ и μ являются вспомогательными, они отвечают за положение точки z на прямых z_1z_2 и z_3z_4.

Чтобы не изобретать имена для вспомогательных числовых переменных, чтобы не объявлять эти переменные, удобно воспользоваться командой `whatever`:

```
pair z;
z=whatever[z1, z2]=whatever[z3, z4];
```

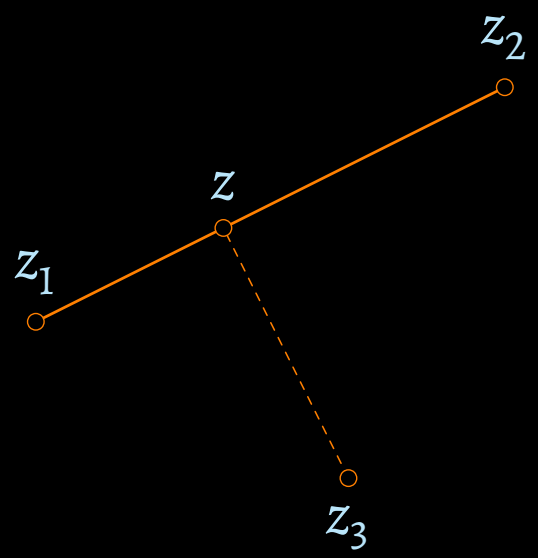
Команда `whatever`

Команда `whatever` создаёт новую безымянную числовую переменную и тут же её подставляет в текст программы.

Название этой команды очень хорошо отражает её предназначение: выражение `z=whatever[z1, z2]` значит «точка z лежит где-то на прямой $z_1 z_2$».

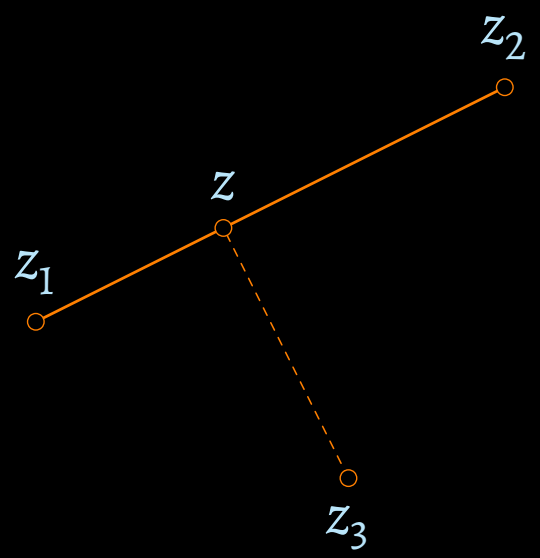
Обратите внимание: каждая команда `whatever` в программе создаёт числовую переменную, полностью независимую от других, созданных той же командой.

Пример: проекция точки на прямую



Применим механизм решения уравнений для вычисления ортогональной проекции точки на прямую.

Пример: проекция точки на прямую

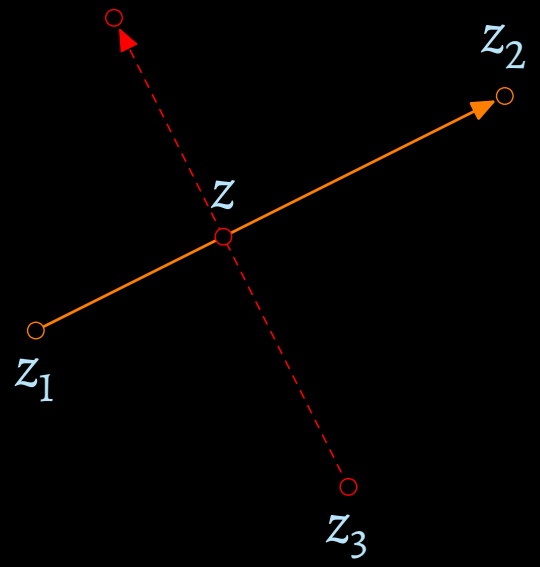


Применим механизм решения уравнений для вычисления ортогональной проекции точки на прямую.

Во-первых, точка z лежит на прямой $z_1 z_2$:

```
z=whatever[z1, z2];
```

Пример: проекция точки на прямую



Во-вторых, z находится на перпендикуляре к прямой z_1z_2, проходящем через точку z_3.

Вторую недостающую точку этого перпендикуляра можно найти, сдвинув точку z_3 на вектор z_1z_2, повёрнутый на 90°:

```
z=whatever[z3, z3+((z2-z1) rotated 90)];
```

Способы придания значений переменным

В METAPOST переменная может получить значение двумя способами:

- с помощью присваивания
- с помощью уравнений

Способы придания значений переменным

В METAPOST переменная может получить значение двумя способами:

- с помощью присваивания
- с помощью уравнений

```
a:=5
```

Способы придания значений переменным

В METAPOST переменная может получить значение двумя способами:

- с помощью присваивания
- с помощью уравнений

```
8=a+3
```

Синтаксис присваивания

⟨*переменная*⟩:=⟨*выражение*⟩;

Синтаксис уравнения

⟨*линейное выражение*⟩=⟨*линейное выражение*⟩;

Синтаксис уравнения

Правая и левая части уравнения должны иметь одинаковый тип.

⟨*линейное выражение*⟩ должно быть линейным по отношению ко всем неизвестным числам.

Допускаются комбинации из нескольких уравнений:

```
z4-z1=z3-z2=(3, y5);
```

Несовместные уравнения

По мере появления в программе уравнений значения участвующих в них переменных уточняются.

```
x+y=1;
3-2y=2x;
```

Несовместные уравнения

По мере появления в программе уравнений значения участвующих в них переменных уточняются.

```
x+y=1;
3-2y=2x;
```

При появлении уравнения, противоречащего предыдущим, возникает ошибка:

```
! Inconsistent equation (off by -1).
```

Несовместные уравнения

По мере появления в программе уравнений значения участвующих в них переменных уточняются.

```
x+y=1;
3-2y=2x;
```

При появлении уравнения, противоречащего предыдущим, возникает ошибка:

```
! Несовместное уравнение (отклонение на -1).
```

Избыточные уравнения

Уравнения, которые возможно преобразовать к тривиальному уравнению `0=0`, вызывают ошибку:

```
3=5-2;
```

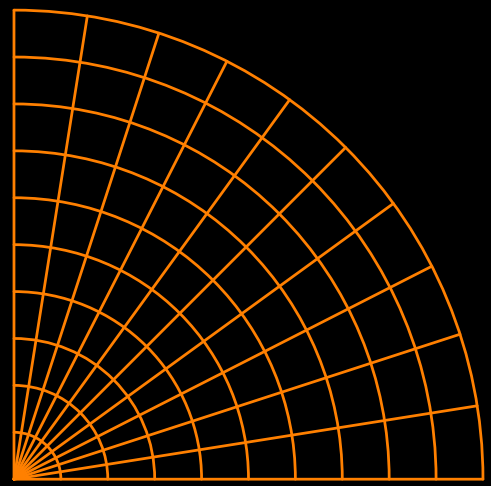
```
! Redundant equation.
```

Избыточные уравнения

Уравнения, которые возможно преобразовать к тривиальному уравнению `0=0`, вызывают ошибку:

```
3=5-2;
```

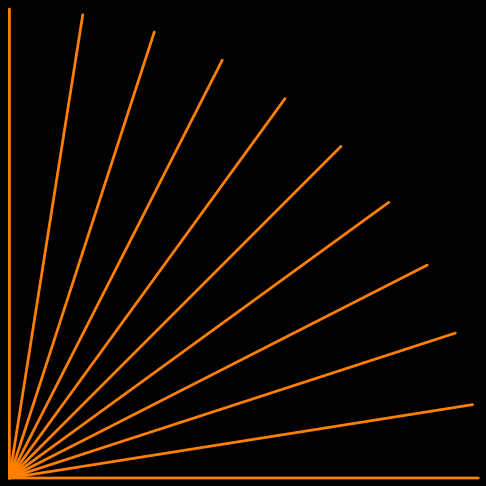
```
! Избыточное уравнение.
```



Управляющие конструкции в METAPOST

В METAPOST, как и в других императивных языках, имеются конструкции, влияющие на порядок выполнения команд: *условная конструкция* и *цикл*.

Цикл for

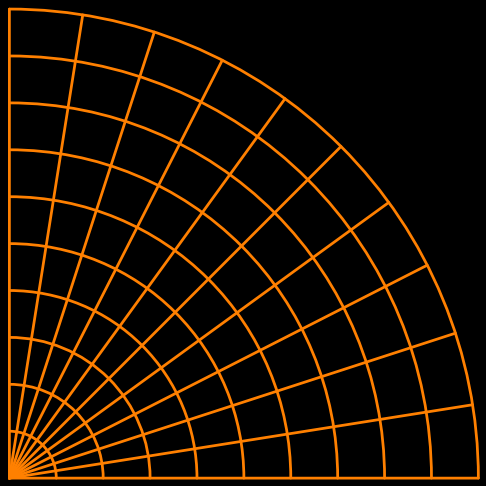


```
for i:=0 step 1 until 10:
    draw origin--3cm*dir(9i);
endfor
```

Цикл for

```
for i:=0 step 1 until 10:
    draw quartercircle
        scaled (6cm*i/10);
endfor
```

Цикл for



```
for i:=0 step 1 until 10:
    draw origin--3cm*dir(9i);
    draw quartercircle
        scaled (6cm*i/10);
endfor
```

Сокращения upto и downto

В циклах for с шагом 1 можно воспользоваться удобным сокращением (макрокомандой) upto:

```
for i:=0 step 1 until 10:
    …
endfor
```

Для фразы step -1 until имеется сокращение downto.

Сокращения upto и downto

В циклах for с шагом 1 можно воспользоваться удобным сокращением (макрокомандой) upto:

```
for i:=0 upto 10:
    …
endfor
```

Для фразы step -1 until имеется сокращение downto.

Цикл for с явным перечислением

Рассмотренные ранее циклы for по очереди присваивали переменной цикла значения из некоторой арифметической прогрессии.

Есть возможность явным образом перечислить значения, которые будут перебираться в цикле:

```
for i:=2, 3, 5, 7, 11, 13, 17, 19, 23:
    …
endfor;
```

Цикл for с явным перечислением

Перечисляемые значения не обязаны быть одного типа:

```
for i:=2, 2.71828459045, "METAPOST", up:
    show(i);
endfor;
```

Этот код приведёт к выводу всех перечисленных разнотипных значений на терминал и в файл протокола:

```
>> 2
>> 2.71828
>> "METAPOST"
>> (0,1)
```

Особенности цикла for

Тело цикла for не обязательно должно быть законченной командой. При обнаружении цикла for компилятор METAPOST производит многократное повторение тела цикла с подстановкой вместо имени переменной цикла (i) её текущего, изменяющегося в цикле, значения. Таким образом, команда

```
s:=for i:=1 upto 100: i+ endfor 0;
```

воспринимается как

```
s:=1+2+3+4+5+…+95+96+97+98+99+100+0;
```

Особенности цикла for

Впрочем, возможно и «классическое» использование цикла для подсчёта суммы чисел от 1 до 100:

```
s:=0;
for i:=1 upto 100: s:=s+i; endfor;
```

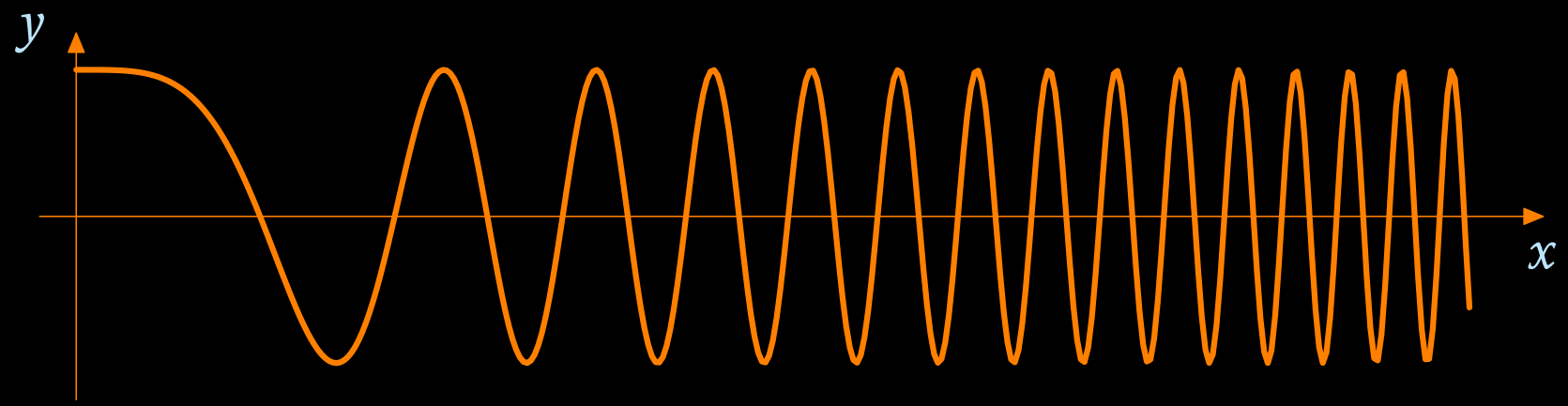
Этот код интерпретируется как

```
s:=0;
s:=s+1;
s:=s+2;
…
s:=s+99;
s:=s+100;
```

Построение путей в цикле

Упомянутая особенность цикла `for` позволяет строить пути в цикле.

Рассмотрим эту возможность на примере графика функции $y = \cos x^2$. График будет представлен как ломаная.



Построение путей в цикле

```
drawarrow (-.25cm, 0)--(10cm, 0)
    withpen thinpen;
drawarrow (0, -1.25cm)--(0, 1.25cm)
    withpen thinpen;

draw
    ((0, 1) for i:=0 step .025 until 9.5:
        --(i, cosd(i**2*rad))
    endfor)
    scaled cm withpen boldpen;

label.bot(btex $x$ etex, (10cm, 0));
label.lft(btex $y$ etex, (0, 1.25cm));
```

Построение путей в цикле

В приведённом примере предполагается, что определены переменные типа `pen` — `thinpen` и `boldpen`, хранящие перья, тонкое и толстое соответственно.

В числовой переменной `rad` содержится количество градусов в одном радиане, т. е. $180/\pi \approx 57{,}3$.

Условная конструкция

```
if ⟨условие⟩: ⟨код⟩
fi
```

Условная конструкция

```
if ⟨условие⟩: ⟨код⟩
else: ⟨код⟩
fi
```

Условная конструкция

```
if ⟨условие⟩: ⟨код⟩
elseif ⟨условие⟩: ⟨код⟩
⟨...⟩
elseif ⟨условие⟩: ⟨код⟩
else: ⟨код⟩
fi
```

Условная конструкция

```
if ⟨условие⟩: ⟨код⟩
elseif ⟨условие⟩: ⟨код⟩
⟨...⟩
elseif ⟨условие⟩: ⟨код⟩
else: ⟨код⟩
fi
```

В качестве ⟨*условия*⟩ может выступать любое логическое выражение.

Примеры логических выражений

x = y	x равно y
x <> y	x не равно y
x < y	x меньше y
x <= y	x меньше либо равно y
x > y	x больше y
x >= y	x больше либо равно y
not q	не q
p and q	p и q
p or q	p или q
cycle p	путь p циклический
color v	значение v имеет тип color

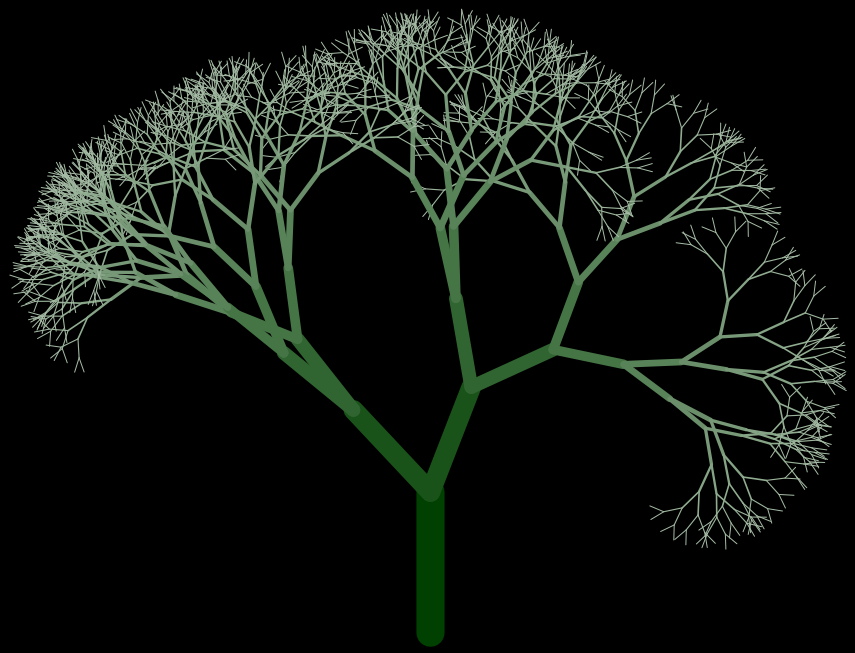
Цикл forever и оператор exitif

В METAPOST имеется конструкция безусловного цикла:

```
forever:
   ⟨код⟩
endfor
```

Такой цикл имеет смысл только в сочетании с возможностью его прервать при выполнении определённого условия. Для досрочного прерывания цикла, в том числе безусловного, служит оператор

```
exitif ⟨условие⟩
```



Метод декомпозиции в программировании

В соответствии с методом декомпозиции программист стремится разбивать исходную, стоящую перед ним задачу на более простые подзадачи. Так же он поступает с подзадачами до тех пор, пока наиболее мелкие подзадачи не станут предельно простыми.

Тем самым программист добивается следующего:

- появляется возможность коллективной работы над задачей, распределения работы между сотрудниками, каждый из которых занят решением своей подзадачи;
- будущая программа перестаёт быть аморфной и приобретает структуру, в результате работа над ней упрощается;
- если для подзадач имеются готовые (возможно, чужие) решения, их можно использовать, сократив интеллектуальные издержки.

Метод декомпозиции и процедуры

Во многих алгоритмических языках для поддержки метода декомпозиции предусмотрены процедуры (функции, подпрограммы).

Программа представляется в виде последовательности вызовов процедур: главная процедура вызывает процедуры, решающие подзадачи второго уровня; те, в свою очередь, вызывают процедуры третьего уровня, и т. д.

Принцип повторного использования кода

Принцип повторного использования кода в программировании требует, чтобы программист вычленял в исходной задаче группы однотипных подзадач.

Готовые подпрограммы для решения подзадач могут вызываться многократно.

За счёт этого код программы становится короче, яснее, его легче отлаживать (замеченная неточность в подпрограмме исправляется один раз, а не во многих местах программного кода).

Общезначимые процедуры могут быть собраны в библиотеки, которые могут быть подключены не только к данной программе, но и к другим.

Макрокоманды

В METAPOST аналогами процедур являются *макрокоманды*.

Несмотря на некоторое отличие от процедур по способу реализации, макрокоманды играют ту же роль, и используются сходным образом.

Макрокоманды, как и процедуры, могут принимать параметры. Тогда они могут решать не одну единственную задачу, а класс подзадач.

Макрокоманды без параметров

Синтаксис:

```
def ⟨имя⟩=⟨тело макрокоманды⟩ enddef
```

Например,

```
def upto=
    step 1 until
enddef;
```

Макрокоманды без параметров

Синтаксис:

def ⟨*имя*⟩=⟨*тело макрокоманды*⟩ enddef

Например,

```
def downto=
    step -1 until
enddef;
```

Макрокоманды с параметрами

Синтаксис:

```
def ⟨имя⟩(expr ⟨список параметров⟩)=
    ⟨тело макрокоманды⟩  enddef
```

Например,

```
def rotatedabout(expr z, d)=
    shifted -z rotated d shifted z
enddef;
```

Библиотека plain.mp

Приведённые определения макрокоманд взяты из файла plain.mp, который METAPOST прочитывает перед началом работы.

Таким образом, многие полезные макрокоманды становятся доступными для нас.

Кроме определений макрокоманд, файл содержит и другие команды, приводящие METAPOST в рабочее состояние.

Например, там объявляются и получают значения полезные переменные, вроде cm, origin, right, up, left, down, red, green, blue, mitered, rounded, ...

Библиотеки

Возможно создавать собственные библиотеки.

Для этого нужные команды, тематически сгруппированные, следует поместить в файл, скажем, `planimetry.mp`, поместить этот файл в такое место, где его найдёт METAPOST, например, в текущую директорию.

Затем в наших программах (или в других библиотеках) можно подключить эту библиотеку командой

```
input planimetry;
```

Группировка и локализация переменных

Группировка — это способ объединения нескольких команд в одну, составную. При этом изменения переменных, которые происходят внутри группы, можно локализовать.

Это значит, что изменения переменных, произошедшие в группе, будут забыты после её завершения (закрытия), если, конечно, программист позаботится о локальности этих переменных.

Для группировки необходимо поместить последовательность команд (после каждой — точка с запятой) между словами `begingroup` и `endgroup`:

`begingroup` ⟨*команды*⟩ `endgroup`

Группировка и локализация переменных

Для того, чтобы сделать переменные локальными в группе, нужно временно сохранить прежние значения командой save:

```
begingroup
    save a, b, c;
    numeric a, b;
    pair c;
    …
endgroup
```

Группы как функции

У группировки имеется и другое предназначение. Если группа завершается выражением (то есть в конце группы отсутствует точка с запятой) METAPOST подставляет вместо группы значение выражения:

begingroup
⟨*команды*⟩ ⟨*выражение*⟩ endgroup

Обычно ⟨*команды*⟩ посвящены закулисным делам, которые нужны для вычисления ⟨*выражения*⟩.

Пример: вычисление факториала

Допустим, в программе потребовалось где-то значение $6!$. Конечно, можно прямо написать 720, но мы на этом примере проиллюстрируем группировку, чтобы в более трудных случаях действовать по аналогии.

```
numeric six_factorial;
six_factorial=begingroup
    save f;
    numeric f;
    f:=1;
    for i:=2 upto 6: f:=f*i; endfor;
    f    % нет точки с запятой!
endgroup;
```

Пример: вычисление факториала

Благодаря нашим заботам приключения переменной `f` внутри группы никак не повлияют на внешний мир.

В частности, если снаружи группы уже была объявлена такая переменная, то она сохранит своё прежнее значение и тип (который мог отличаться от `numeric`).

Если же переменная `f` вне группы не определялась, такое же положение дел сохранится и после приведённого кода.

Макрокоманда для факториала

Чтобы сделать код для вычисления факториала более универсальным (пригодным для вычисления не только $6!$, но и $n!$ для произвольного n), оформим его как макрокоманду с параметром:

```
def factorial(expr n)=
    begingroup
        save f;
        numeric f;
        f:=1;
        for i:=2 upto n: f:=f*i; endfor;
        f
    endgroup    % нет точки с запятой!
enddef;
```

Макрокоманда для факториала

Обратите внимание на отсутствие точки с запятой после `endgroup`. Если бы она была там, команда

```
s:=factorial(5)+factorial(6);
```

воспринималось бы как

```
s:=120;+720;;
```

Это бы привело, во-первых, к неправильному значению `s`, и, во-вторых, к ошибке:

```
>> 720
! Isolated expression.
```

Макрокоманда для факториала

Обратите внимание на отсутствие точки с запятой после `endgroup`. Если бы она была там, команда

```
s:=factorial(5)+factorial(6);
```

воспринималось бы как

```
s:=120;+720;;
```

Это бы привело, во-первых, к неправильному значению `s`, и, во-вторых, к ошибке:

```
>> 720
! Изолированное выражение.
```

Изолированное выражение

Ошибка «изолированное выражение» является довольно распространённой и обычно свидетельствует о неполадках с точками с запятой.

```
s:=120;+720;;
```

В нашем примере проблемной будет выделенная команда.

Выражение `+720` будет вычислено (значение `720`), и затем никак не использовано. Это является недопустимым в METAPOST, хотя в некоторых алгоритмических языках это не криминал.

Макрокоманды vardef

Поскольку часто определение макрокоманды сочетается с группировкой, в METAPOST имеется вариант макроопределений — vardef. В теле таких определений группировка уже предполагается неявно:

```
def factorial(expr n)=
    begingroup
        save f;
        numeric f;
        f:=1;
        for i:=2 upto n: f:=f*i; endfor;
        f
    endgroup
enddef;
```

Макрокоманды vardef

Поскольку часто определение макрокоманды сочетается с группировкой, в METAPOST имеется вариант макроопределений — vardef. В теле таких определений группировка уже предполагается неявно:

```
vardef factorial(expr n)=

        save f;
        numeric f;
        f:=1;
        for i:=2 upto n: f:=f*i; endfor;
        f

enddef;
```

Макрокоманды vardef

Поскольку часто определение макрокоманды сочетается с группировкой, в METAPOST имеется вариант макроопределений — vardef. В теле таких определений группировка уже предполагается неявно:

```
vardef factorial(expr n)=
    save f;
    numeric f;
    f:=1;
    for i:=2 upto n: f:=f*i; endfor;
    f
enddef;
```

Пример: пересечение прямых

Оформим код для нахождения пересечения прямых как макрокоманду. На вход макрокоманда будет получать четыре точки (пары), на выходе будет возвращать точку пересечения прямых, проходящих через первые две (p и q) и последние две (r и s) точки.

```
vardef intersection(expr p, q, r, s)=
    save z;
    pair z;
    z=whatever[p, q]=whatever[r, s];
    z
enddef;
```

Определение макрокоманды whatever

В библиотеке plain.mp команда whatever определяется так:

```
vardef whatever=save ?; ? enddef;
```

В теле этого макроопределения локализуется переменная ?
(в METAPOST возможны и такие имена), и, не получая никакого значения, возвращается. Имя этой переменной тут же забывается, а неопределённое значение можно использовать.

Таким образом, команда whatever — это своего рода генератор анонимных числовых переменных.

Пример: описанная окружность

Определяем центр описанной окружности как точку пересечения серединных перпендикуляров (медиатрис) сторон треугольника:

```
vardef circumcenter(expr p, q, r)=
    intersection(
        .5[p, q],
        .5[p, q]+((q-p) rotated 90),
        .5[q, r],
        .5[q, r]+((r-q) rotated 90)
    )
enddef;
```

Пример: описанная окружность

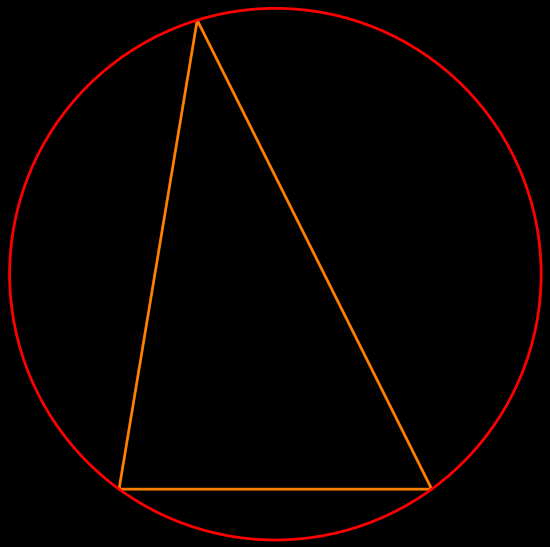
Радиус находим как расстояние от центра до одной из вершин (неважно какой):

```
vardef circumradius(expr p, q, r)=
    abs(circumcenter(p, q, r)-p)
enddef;
```

Сама окружность (путь):

```
vardef circumcircle(expr p, q, r)=
    fullcircle
        scaled (2circumradius(p, q, r))
        shifted circumcenter(p, q, r)
enddef;
```

Использование макрокоманды circumcircle



```
z1=origin;
z2=(2cm, 0);
z3=(.5cm, 3cm);
draw z1--z2--z3--cycle;
draw circumcircle(z1, z2, z3)
    withcolor red;
```

Пример: вписанная окружность

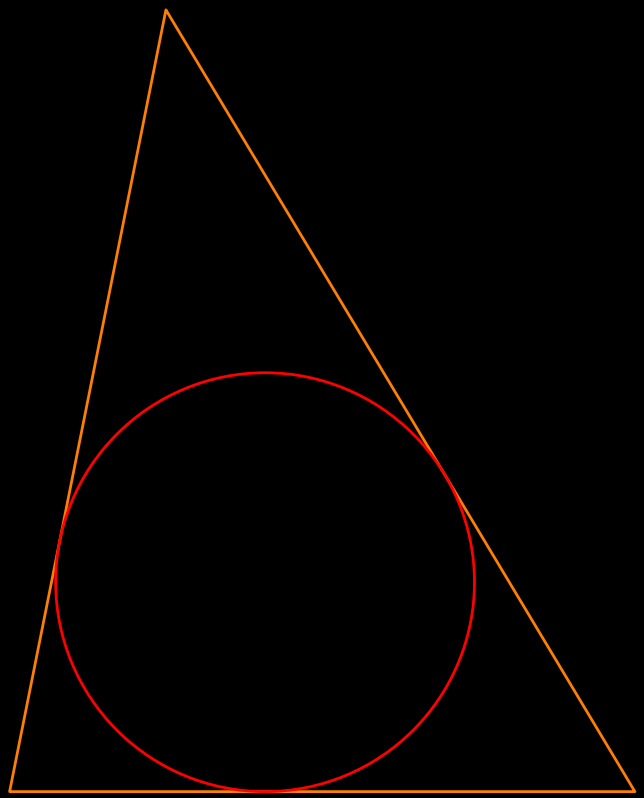
Центр вписанной в треугольник окружности найдём как точку пересечения двух биссектрис. Воспользуемся тем, что сумма двух единичных векторов служит биссектрисой угла между этими векторами, так что биссектриса угла `a` треугольника проходит через точку `a+unitvector(b-a)+unitvector(c-a)` (`b` и `c` — остальные вершины).

```
vardef incenter(expr a, b, c)=
    intersection(
        a, a+unitvector(b-a)+unitvector(c-a),
        b, b+unitvector(c-b)+unitvector(a-b)
    )
enddef;
```

Пример: вписанная окружность

```
vardef inradius(expr a, b, c)=
    save i;
    pair i;
    i=incenter(a, b, c);
    abs(i-intersection(a, b, i, i+((b-a) zscaled up)))
enddef;

vardef incircle(expr a, b, c)=
    fullcircle scaled 2inradius(a, b, c)
        shifted incenter(a, b, c)
enddef;
```

Использование макрокоманды incircle



```
z1=origin;
z2=(4cm, 0);
z3=(cm, 5cm);
draw z1--z2--z3--cycle;
draw incircle(z1, z2, z3)
    withcolor red;
```

Рекурсия

В METAPOST разрешена *рекурсия*, то есть ситуация, когда макрокоманда вызывает сама себя, прямо или косвенно.

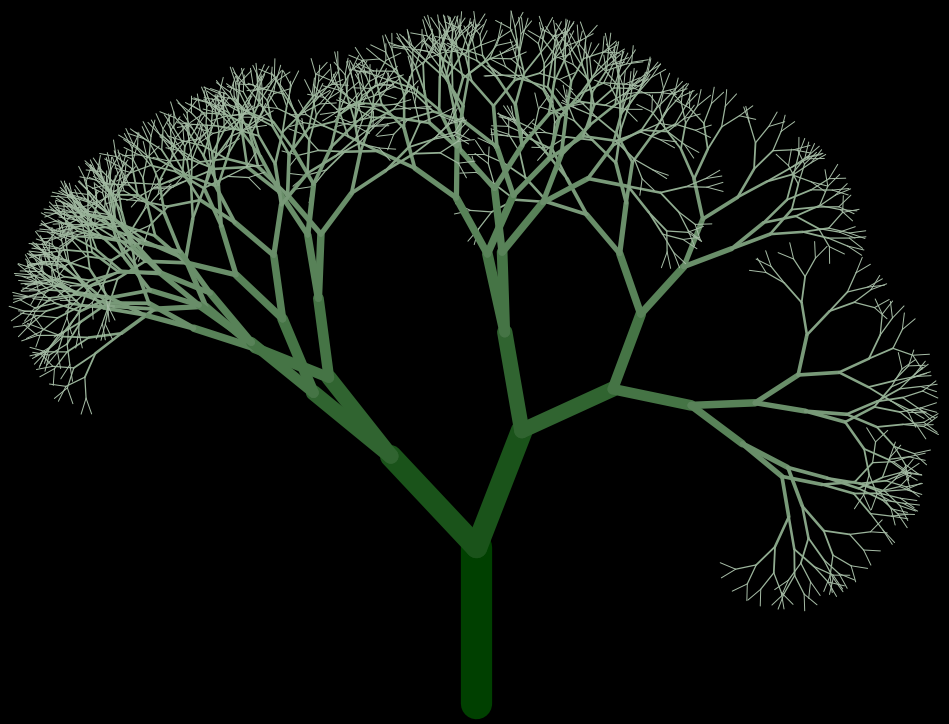
Проиллюстрируем этот тезис на классическом примере — вычислении факториала.

Рекурсивное вычисление факториала

Воспользуемся известным соотношением $n! = n(n-1)!$. Вместе с равенством $0! = 1$ эта формула даёт способ рекурсивного вычисления факториала:

```
def factorial(expr n)=
    if n=0: 1 else: n*factorial(n-1) fi
enddef;
```

Дерево



Дерево как рекурсивная структура

Наблюдения за деревьями, этими прекрасными творениями природы, показывают, что они имеют рекурсивное устройство.

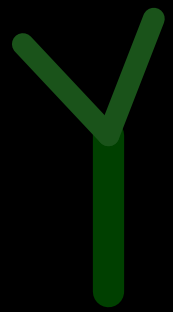
Дерево состоит из ствола и растущих из него ветвей, каждая из которых является уменьшенной копией дерева.

Такое определение дерева предусматривает бесконечное количество ветвей. Оно нуждается в уточнении.

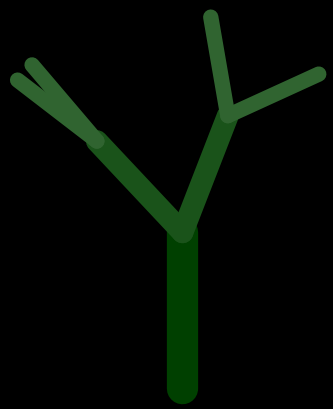
Дерево n-го порядка (при $n > 0$) состоит из ствола и растущих из него ветвей, каждая из которых представляет собой дерево $(n-1)$-го порядка. Дерево 0-го порядка — это голый ствол.

Рост дерева: 0-й порядок

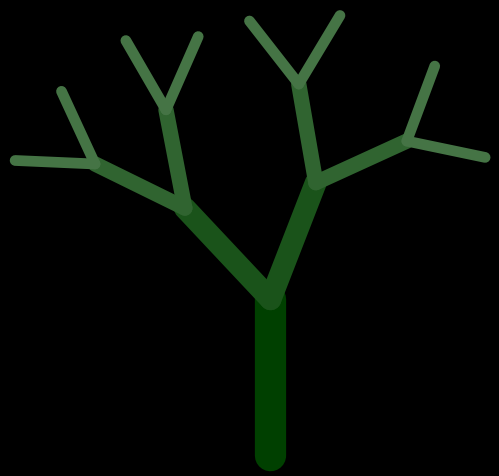
Рост дерева: 1-й порядок



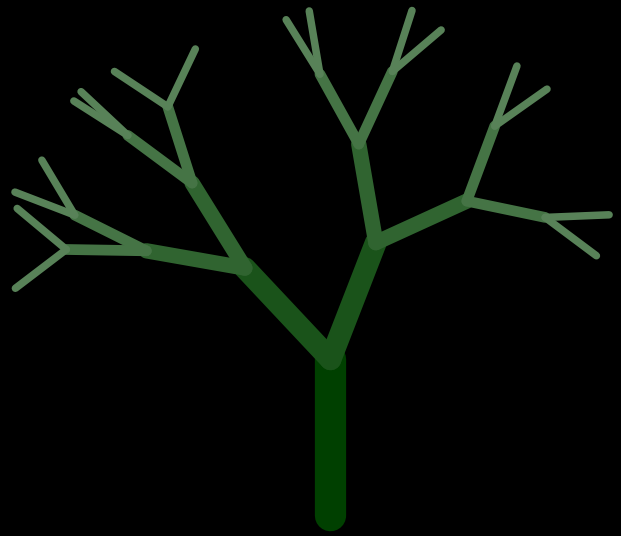
Рост дерева: 2-й порядок



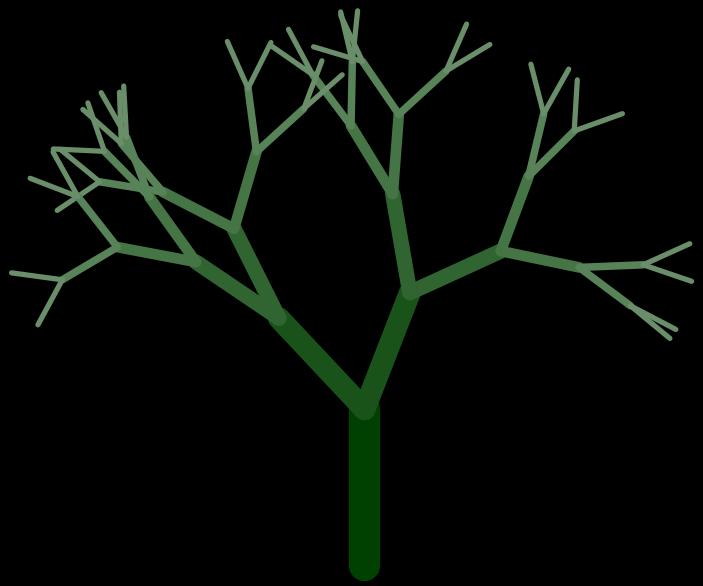
Рост дерева: 3-й порядок



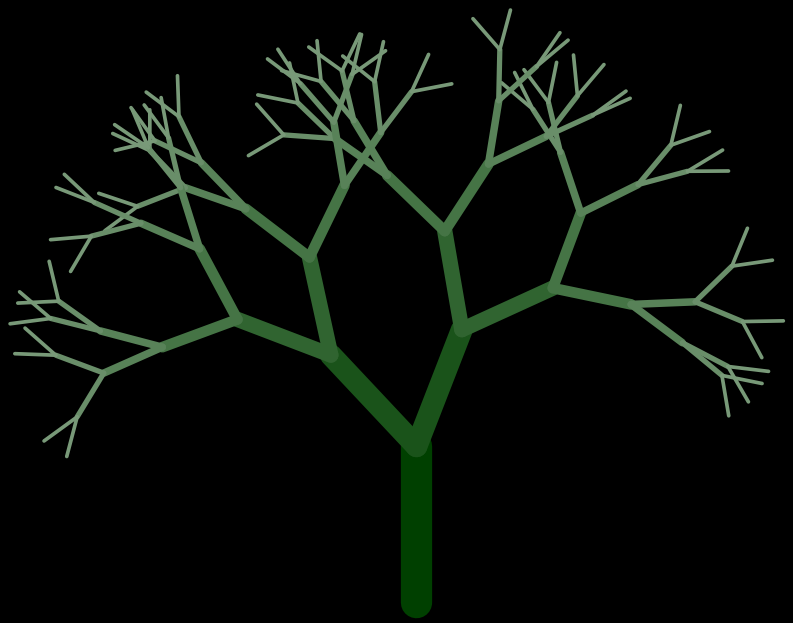
Рост дерева: 4-й порядок



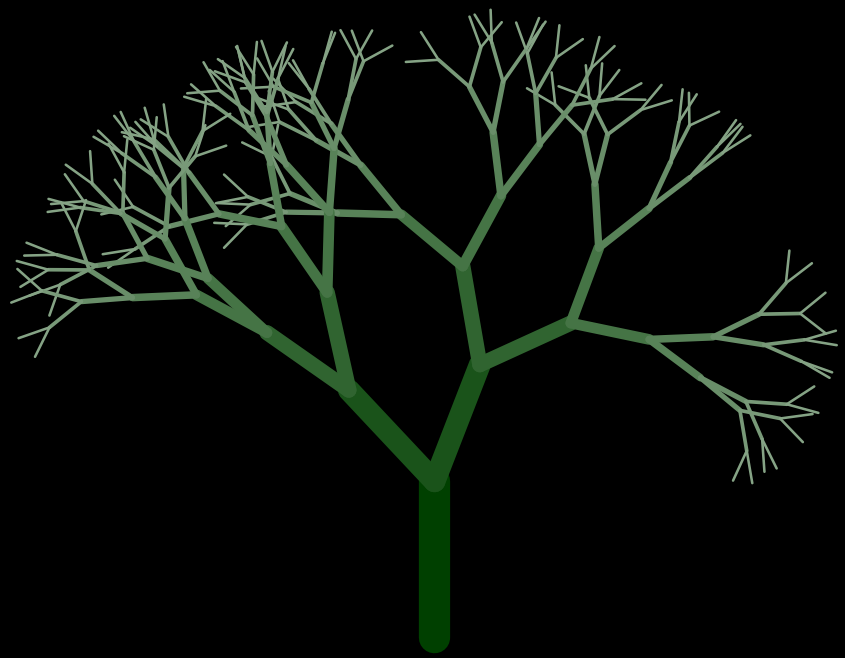
Рост дерева: 5-й порядок



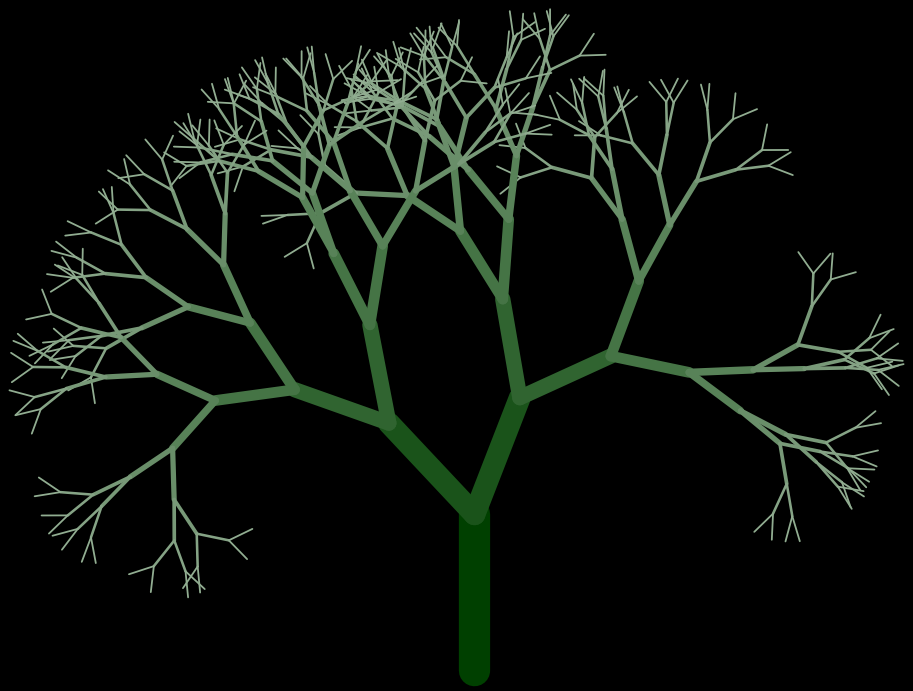
Рост дерева: 6-й порядок



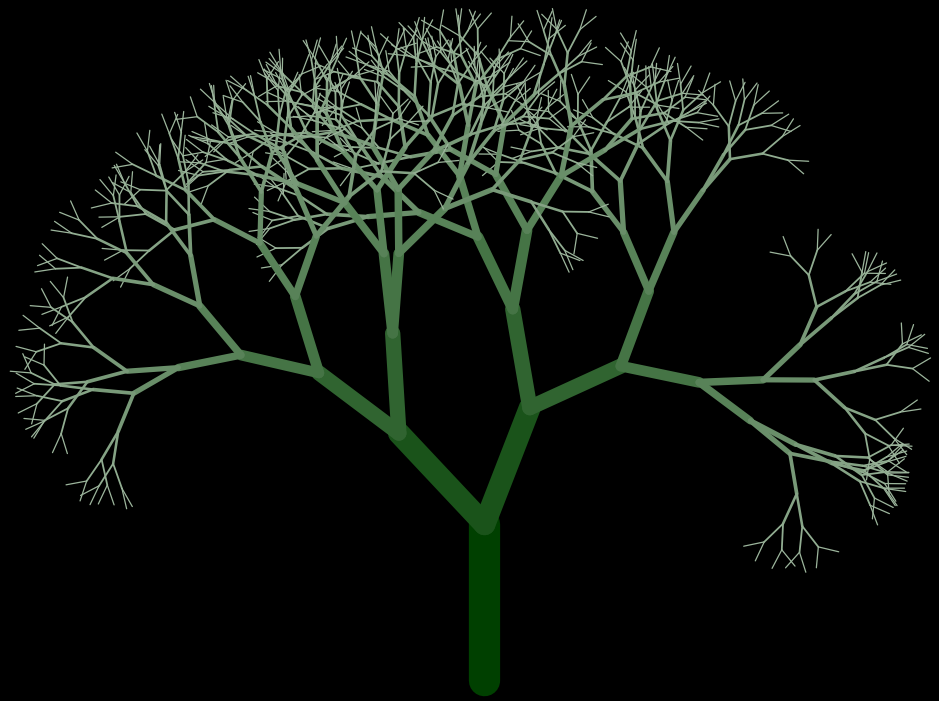
Рост дерева: 7-й порядок



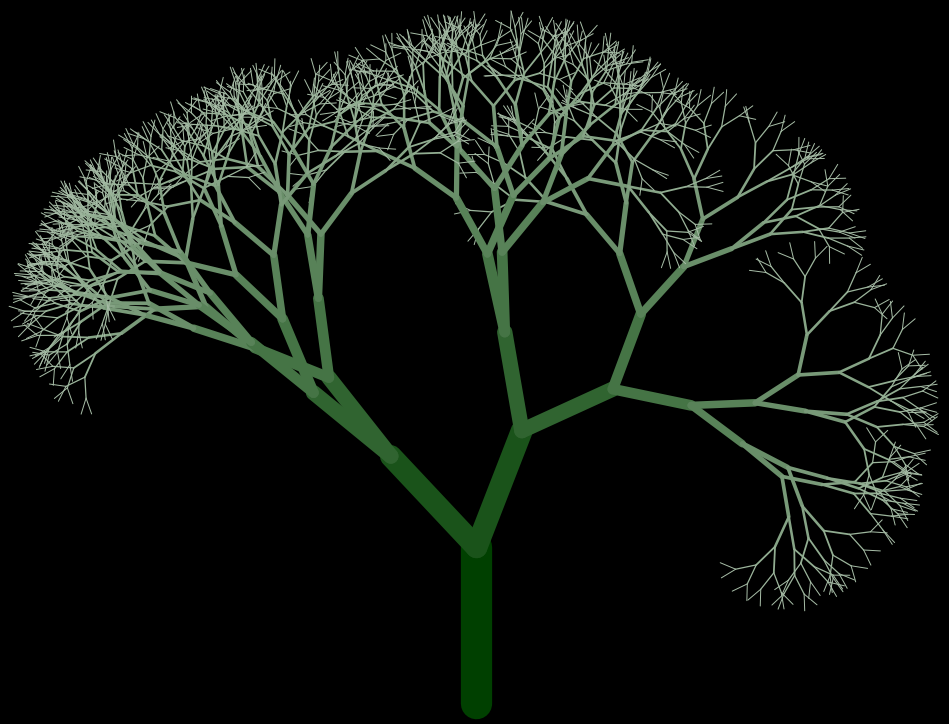
Рост дерева: 8-й порядок



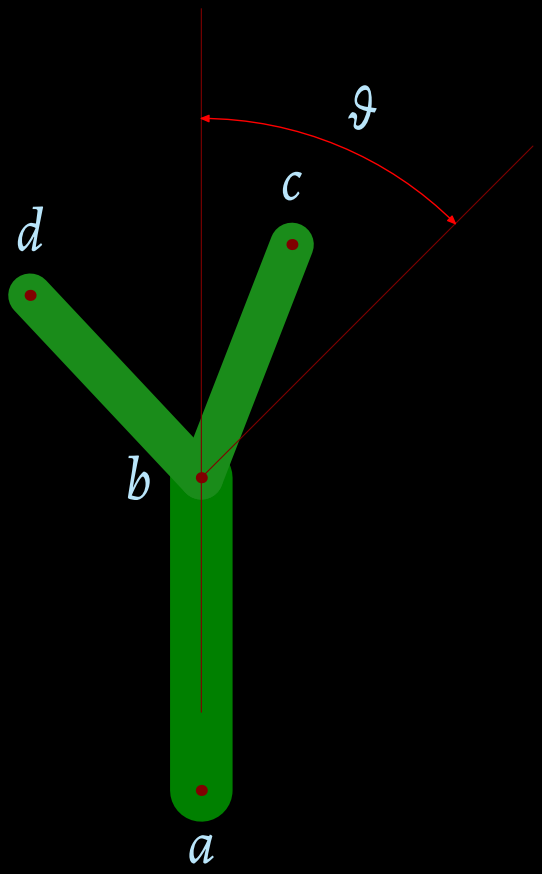
Рост дерева: 9-й порядок



Рост дерева: 10-й порядок



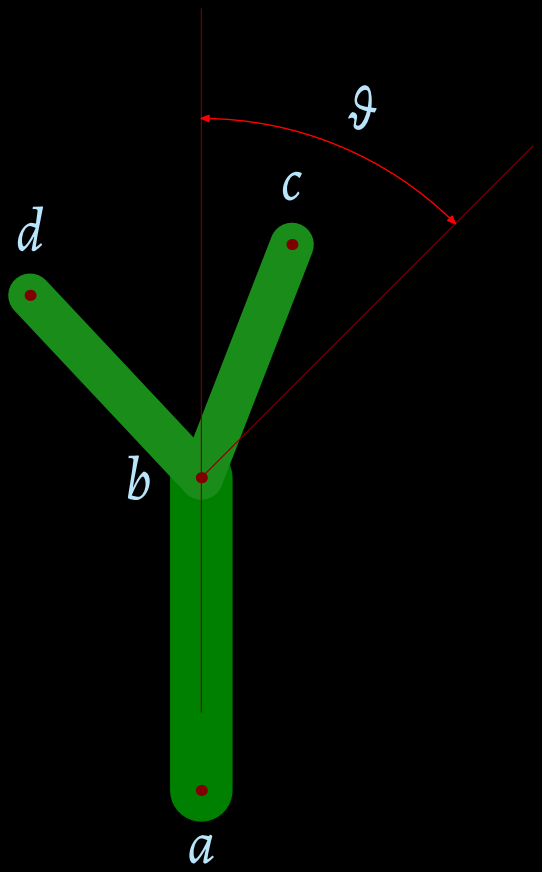
Построение дерева



Правая ветка дерева отклоняется от вертикали в пределах закрашенного сектора. Угол отклонения случайный и равномерно распределённый в пределах от 0 до ϑ (в нашем примере 45°.

Угол ϑ будет в программе храниться в переменной `theta`.

Построение дерева



Длина веток получается умножением параметра `shortening` на длину ствола дерева.

Толщина веток получается умножением параметра `thinning` на толщину ствола дерева `thickness`.

Ветки светлее ствола, и параметр `lightening` отвечает за осветление.

Дерево: код программы

Определяем параметры...

```
theta:=45;
thinning:=.7;
shortening:=.8;
lightening:=.1;
```

Дерево: код программы

Определяем макрокоманду `tree` с параметрами `a` и `b` (основание и верхушка ствола), `n` (порядок дерева), `thickness` (толщина ствола), `clr` (цвет ствола).

```
vardef tree(expr a, b, n, thickness, clr)=
    …
enddef;
```

(код макрокоманды на следующем слайде).

Вызываем макрокоманду `tree`:

```
tree(origin, (0, cm), 10, 2mm, .25green);
```

Рисунок готов.

Тело макрокоманды tree

Объявляем параметры — координаты верхушек веток...

```
save c, d;
pair c, d;
```

Определяем координаты верхушки правой ветки...

```
c:=shortening[b,
    a rotatedaround(b, 180-theta+uniformdeviate(theta))];
```

и левой...

```
d:=shortening[b,
    a rotatedaround(b, theta-180+uniformdeviate(theta))];
```

Тело макрокоманды tree

Рисуем ствол...

```
draw a--b
    withpen pencircle scaled thickness
    withcolor clr;
```

Для деревьев порядка > 0 рисуем ветки...

```
if n>0:
    tree(b, c, n-1, thinning*thickness,
        lightening[clr, white]);
    tree(b, d, n-1, thinning*thickness,
        lightening[clr, white]);
fi
```

Макрокоманды vardef … @#

В библиотеке plain.mp даётся следующее определение:

```
vardef z@#=(x@#, y@#) enddef;
```

Такое определение создаёт видимость того, что переменные, чьи имена начинаются с токена z, объявлены как переменные типа pair.

Таким образом, можно пользоваться «переменными» с именами z1, z13a, z.p.q[25] как будто они объявлены как пары.

На самом деле при употреблении таких имён происходит подстановка тела макрокоманды z@#: вместо z.p.q[25] подставляется (x.p.q[25], y.p.q[25]). Переменные x.p.q[25] и y.p.q[25], если они не объявлялись, воспринимаются в программе как числовые (numeric), что и требуется.

Макрокоманды vardef … @#

Кроме всего прочего, в определении макрокоманды `beginfig` имеется фрагмент

```
begingroup
save x, y;
```

а в определении команды `endfig` —

```
endgroup
```

Макрокоманды vardef … @#

Так что в начале каждого рисунка переменные, чьи имена начинаются с токенов x или y, становятся независимыми от таких же имён внутри кода других рисунков.

Код многих рисунков, как мы видели в примерах, начинается сразу с определений пар z.⟨*суффикс*⟩ (а их объявление пропускается):

```
z1=origin;
z2=(3cm, 0);
z3=(0, 4cm);

draw z1--z2--z3--cycle;
```

Применение определений vardef … @#

Эта особенность определений vardef позволяет передавать в макрокоманды не только выражения, но и имена.

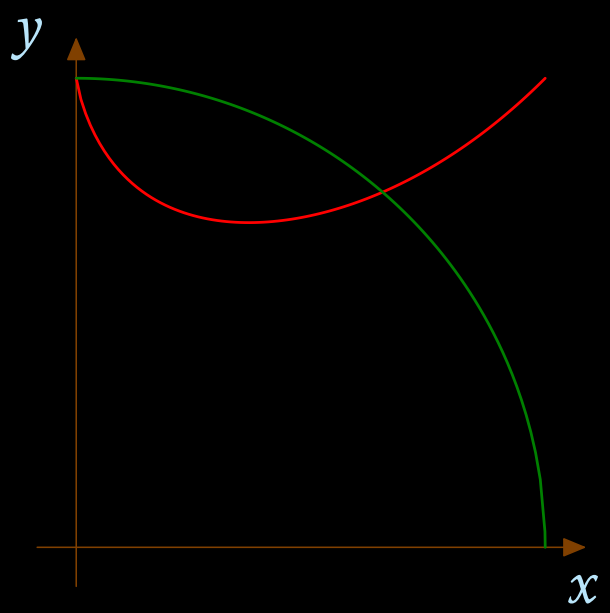
Например, можно определить макрокоманду, которая построит и возвратит график функции (путь) на заданном отрезке. Она в качестве параметров примет числа — концы заданного отрезка (a и b), шаг delta, а также имя функции (@#), чей график следует построить.

Определение будет таким:

```
vardef graph@#(expr a, b, delta)=
    for i:=a step delta until b:
        (i, @#(i))--
    endfor (b, @#(b))
enddef;
```

Применение макрокоманды graph

В качестве примера применения макрокоманды graph построим графики функций $f(x) = x^x$ и $g(x) = \sqrt{1 - x^2}$ на отрезке $[0; 1]$ с шагом 0,01 (полученная ломаная будет иметь сто звеньев). Код для рисования координатных осей не приводится.




```
vardef f(expr x)=x**x enddef;
vardef g(expr x)=1+-+x enddef;

draw graph.f(0, 1, .01)
    scaled 3cm
    withcolor red;
draw graph.g(0, 1, .01)
    scaled 3cm
    withcolor .5green;
```

Нелинейные уравнения

Некоторые построения требуют решения нелинейных уравнений.

В METAPOST отсутствуют встроенные средства для этого.

Процедура solve

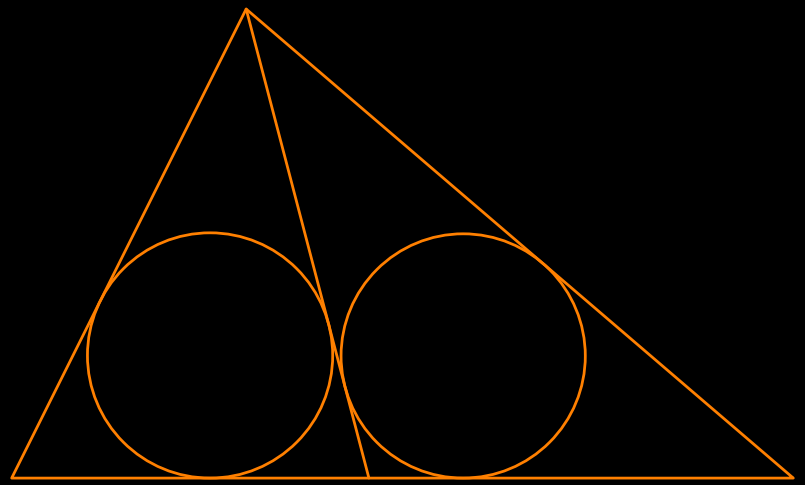
Процедура solve из библиотеки plain.mp позволяет численно решать уравнения с одной неизвестной методом бисекций (бинарным поиском).

$$\texttt{solve}\ f(t_+, t_-)$$

Здесь f — имя процедуры с одним числовым параметром, возвращающей значение типа boolean. Её мы должны запрограммировать сами с помощью vardef.

Процедура solve возвращает число $t \in [t_+, t_-]$, близко к которому значение $f(t)$ меняются с истинного на ложное. Предполагается, что $f(t_+)$ истинно, а $f(t_-)$ — ложно.

Пример с нелинейным уравнением



Требуется провести чевиану в треугольнике так, чтобы она разделила его на два треугольника с равными радиусами вписанных кругов.

Задача сводится к решению системы рациональных уравнений. Общее решение можно найти вручную (очень сложно) или с помощью систем компьютерной алгебры. Для наших целей подойдёт численное решение.

Пример с нелинейным уравнением *(продолжение)*

Определим вершины треугольника:

```
z1=origin;
z2=(5cm, 0);
z3=(1.5cm, 3cm);
```

Искомое основание чевианы будем искать как точку $t[z_1, z_2]$, где t найдём с помощью `solve`.

Пример с нелинейным уравнением *(продолжение)*

Определим процедуру `f`, используя ранее определённую процедуру `inradius`:

```
vardef f(expr t)=
    inradius(z1, t[z1, z2], z3)<inradius(t[z1, z2], z2, z3)
enddef;
```

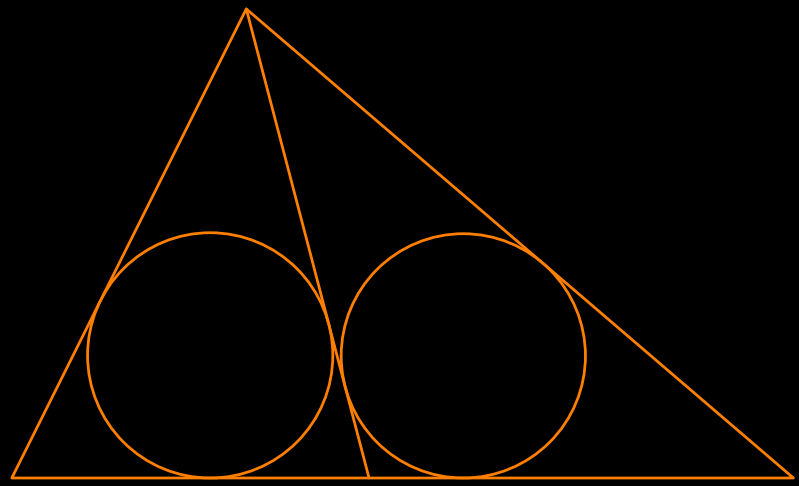
При $t_+ = 0$ она возвращает истину, при $t_- = 1$ — ложь. Где-то между этим значениями истинность $f(t)$ меняется. Теперь находим основание чевианы:

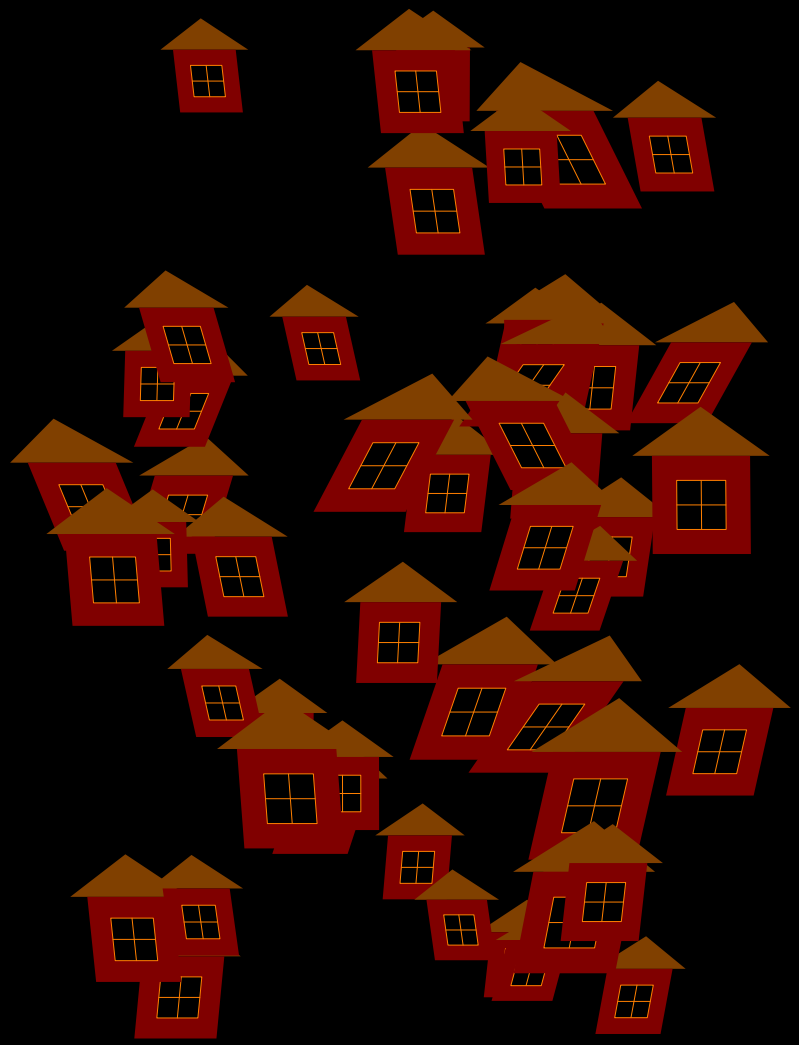
```
z4=(solve f(0, 1))[z1, z2];
```

Пример с нелинейным уравнением *(продолжение)*

Рисуем:

```
draw z1--z2--z3--cycle;
draw z4--z3;
draw incircle(z1, z4, z3);
draw incircle(z4, z2, z3);
```





Переменная currentpicture

В процессе рисования с помощью команд draw и fill текущее изображение сохраняется в переменной currentpicture типа picture.

Команда endfig, помимо прочих действий, выводит содержимое переменной currentpicture в файл в виде последовательности команд на языке PostScript. Команда beginfig «обнуляет» эту переменную, а также присваивает номер будущей картинки (обычно этот номер включается в имя PostScript-файла).

Внутренняя структура переменных типа picture сложна. Она позволяет хранить элементы изображения (пути) вместе с атрибутами рисования (цвет линии, её толщина, цвет заливки циклического контура), а также последовательность, в которой эти элементы рисовались.

Рисование на картинках и команда addto

Имеется возможность заготавливать картинки впрок для многократного использования. Можно создавать библиотеки готовых картинок.

Для рисования на картинках, отличных от currentpicture, в METAPOST имеется команда addto:

addto ⟨*картинка*⟩ doublepath ⟨*путь*⟩ ⟨*опции*⟩

(аналог draw ⟨*путь*⟩ ⟨*опции*⟩)

Рисование на картинках и команда addto

Имеется возможность заготавливать картинки впрок для многократного использования. Можно создавать библиотеки готовых картинок.

Для рисования на картинках, отличных от currentpicture, в METAPOST имеется команда addto:

addto ⟨*картинка*⟩ contour ⟨*циклический путь*⟩ ⟨*опции*⟩

(аналог fill ⟨*циклический путь*⟩ ⟨*опции*⟩)

Рисование на картинках и команда addto

Имеется возможность заготавливать картинки впрок для многократного использования. Можно создавать библиотеки готовых картинок.

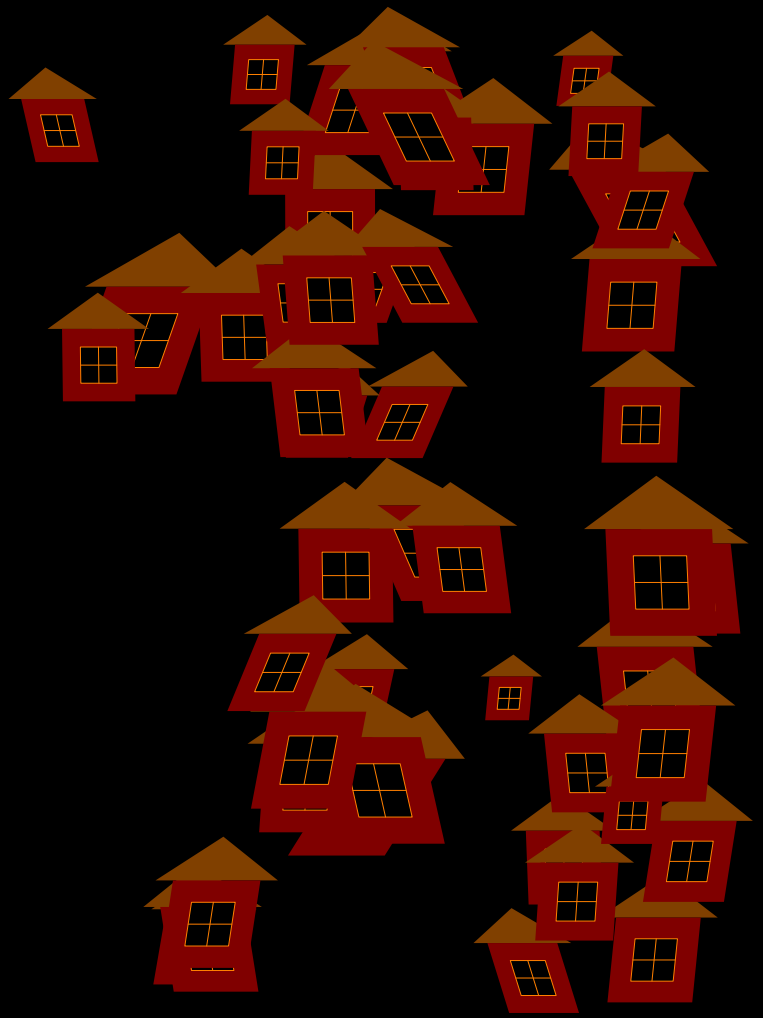
Для рисования на картинках, отличных от currentpicture, в METAPOST имеется команда addto:

addto ⟨*картинка*⟩ also ⟨*картинка*⟩

(аналог draw ⟨*картинка*⟩)

Пример: сумасшедшие домики

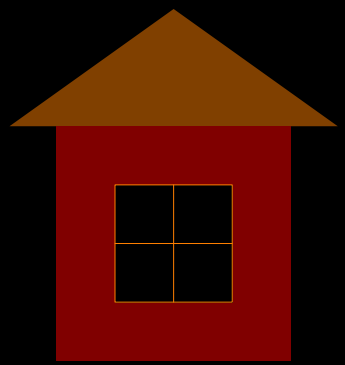
Проиллюстрируем сказанное примером — рисованием посёлка из сумасшедших домиков (это рисунок, открывающий настоящий раздел).



Пример: сумасшедшие домики

Все домики на рисунке отличаются друг от друга разными аффинными преобразованиями, выбранными случайно. Они равномерно разбросаны по полю рисунка, имеют размеры, немного и случайно отклоняющиеся от некоторого среднего размера, а также небольшие и случайные наклоны.

Все эти сумасшедшие домики получаются нанесением на поле рисунка одной и той же картинки (на ней изображён один совершенно нормальный домик), подвергнутой различным аффинным преобразованиям. Картинка заранее заготовлена в переменной `house`, имеющей тип `picture`.



Пример: сумасшедшие домики

Объявляем новую переменную-картинку house. Инициализируем её пустой картинкой...

```
picture house;
house=nullpicture;
```

Пример: сумасшедшие домики

Рисуем в переменную house крышу...

```
addto house contour
    (-.2, 1)--(1.2, 1)--(.5, 1.5)--cycle
    withcolor .5orange;
```

Пример: сумасшедшие домики

Стену...

```
addto house contour unitsquare
    withcolor .5red;
```

Пример: сумасшедшие домики

Чёрный проём окна...

```
addto house contour
    unitsquare scaled .5 shifted (.25, .25)
    withcolor black;
```

Пример: сумасшедшие домики

Оконную раму...

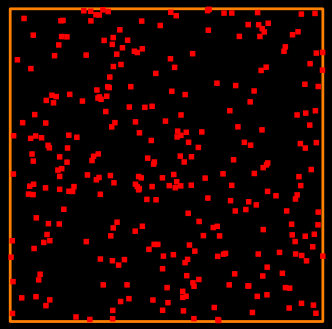
```
addto house doublepath
    unitsquare scaled .5 shifted (.25, .25)
    withcolor orange;
addto house doublepath
    (.25, .5)--(.75, .5)
    withcolor orange;
addto house doublepath
    (.5, .25)--(.5, .75)
    withcolor orange;
```

Пример: сумасшедшие домики

И, наконец, наносим на наш рисунок (то есть в переменную currentpicture) 50 копий картинки house, подвергнутых сумасшедшим аффинным преобразованиям. Пояснения последуют.

```
for i:=1 upto 50:
    draw house
        slanted (normaldeviate*.005cm)
        scaled (.25cm+normaldeviate*.05cm)
        shifted (uniformdeviate(3cm), uniformdeviate(4cm));
endfor
```

Команда uniformdeviate

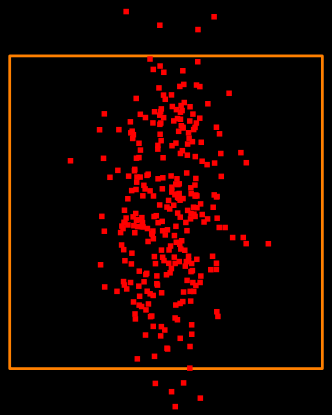


По площади квадратика размером 2 см равномерно разбросаны 300 точек, чьи координаты (и абсцисса, и ордината) получены с помощью команды `uniformdeviate(2cm)`.

Примитивная команда `uniformdeviate(`⟨*число*⟩`)` возвращает (псевдо)случайное число, равномерно распределённое на промежутке от `0` до ⟨*числа*⟩.

```
for i:=1 upto 300:
    draw (uniformdeviate(2cm), uniformdeviate(2cm))
        withpen pensquare scaled bp
        withcolor red;
endfor
```

Команда normaldeviate

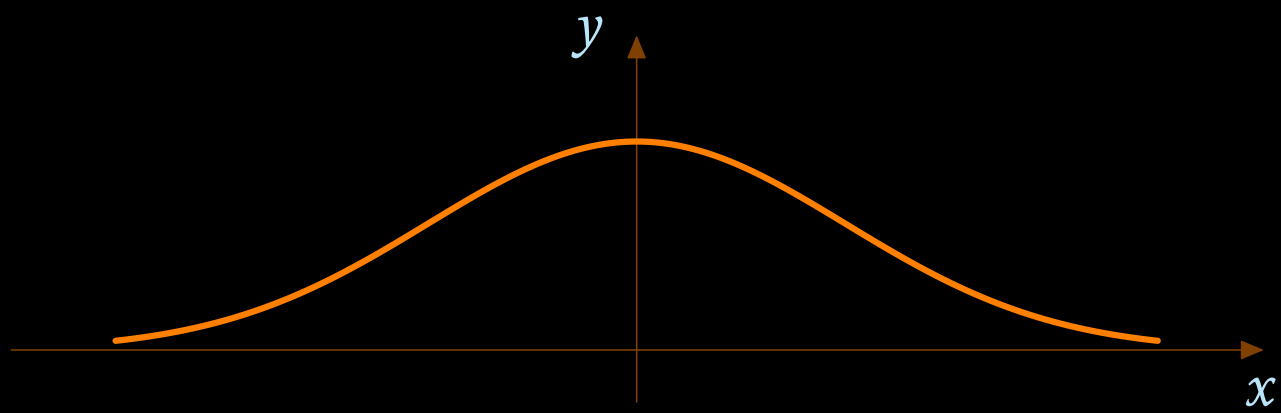


А теперь разбросаны 300 точек, чьи координаты получены с помощью команды normaldeviate. Это похоже на прицельную стрельбу в центр квадрата, при которой кучность по горизонтали несколько выше, чем по вертикали.

```
for i:=1 upto 300:
    draw (cm+normaldeviate*2mm, cm+normaldeviate*5mm)
        withpen pensquare scaled bp
        withcolor red;
endfor
```

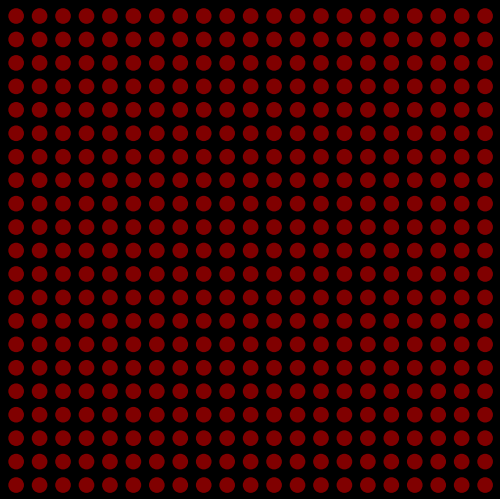
Команда normaldeviate

Примитивная команда normaldeviate возвращает *нормально-распределённое* (псевдо)случайное число. Плотность распределения выше для чисел вблизи нуля и убывает до нуля на отдалении. Бóльшие по модулю числа имеют меньшую вероятность появления. Эта плотность описывается формулой $e^{-x^2/2}$, её график имеет колоколообразную форму:



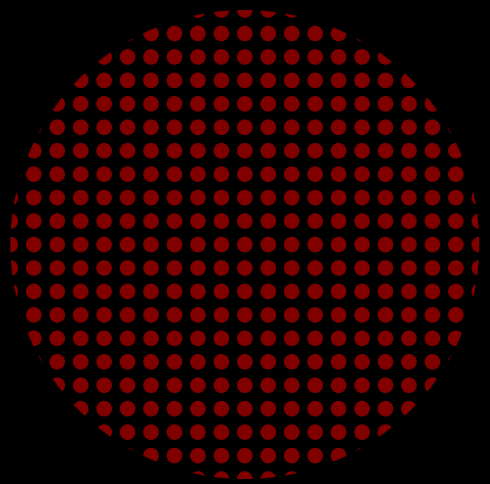
Такое распределение ещё называют *гауссовским*.

Вырезание



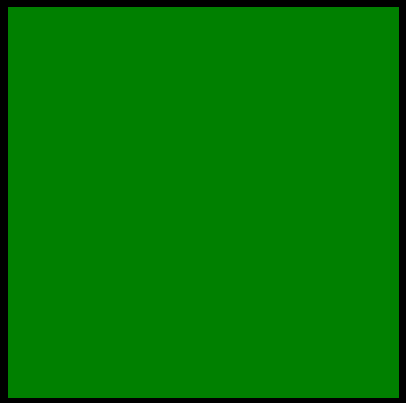
```
for j:=-10 upto 10:
    for i:=-10 upto 10:
        fill fullcircle
            scaled mm
            shifted ((i, j)*1.5mm)
            withcolor .5red;
    endfor
endfor
```

Вырезание



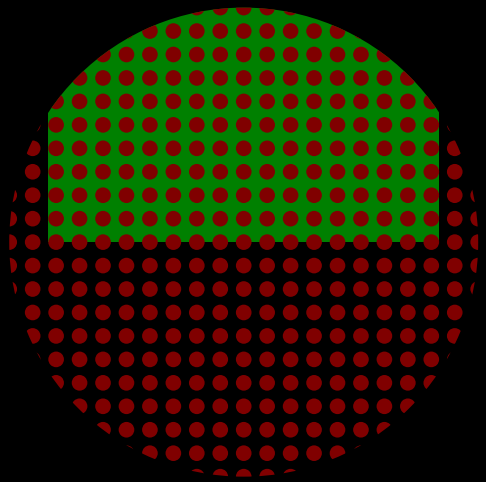
```
for j:=-10 upto 10:
    for i:=-10 upto 10:
        fill fullcircle
            scaled mm
            shifted ((i, j)*1.5mm)
            withcolor .5red;
    endfor
endfor

clip currentpicture
    to fullcircle scaled 3cm;
```

Текстуры



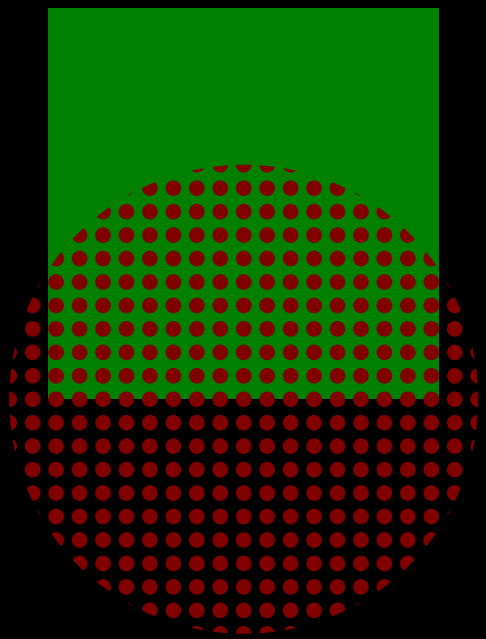
Если требуется наложить описанное изображение на рисунок, где уже что-то изображено, описанный приём не сработает.
Дело в том, что последняя команда `clip currentpicture to …` ограничит всё ранее нарисованное внутри пути `fullcircle scaled 3cm`.

Текстуры



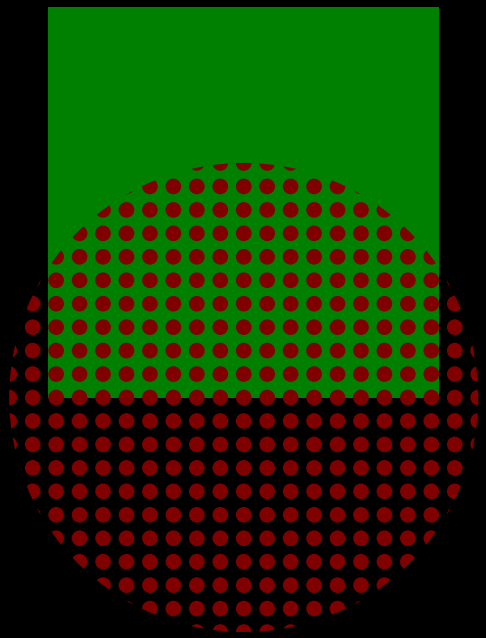
Если требуется наложить описанное изображение на рисунок, где уже что-то изображено, описанный приём не сработает.
Дело в том, что последняя команда `clip currentpicture to …` ограничит всё ранее нарисованное внутри пути `fullcircle scaled 3cm`.

Текстуры



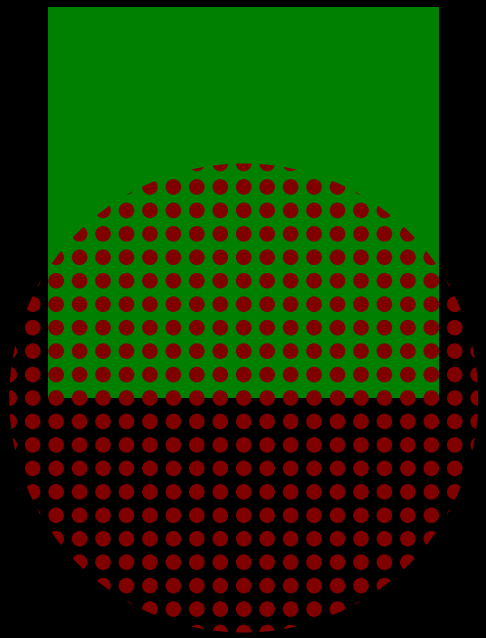
Выход из положения заключается в том, чтобы наносить текстуру не прямо на текущий рисунок в переменной `currentpicture`, а на вспомогательную картинку. Затем нужно применить команду `clip` к этой вспомогательной картинке. Наконец, следует наложить вспомогательную картинку на `currentpicture`.

Текстуры: код



```
fill unitsquare scaled 2.5cm
    shifted (-1.25cm, 0)
    withcolor .5green;

picture tmpPic;
tmpPic:=nullpicture;

for j:=-10 upto 10:
    for i:=-10 upto 10:
        addto tmpPic contour
            fullcircle scaled mm
            shifted ((i, j)*1.5mm)
            withcolor .5red;
    endfor
endfor
```

Текстуры: код *(продолжение)*



```
clip tmpPic to fullcircle
    scaled 3cm;

addto currentpicture also tmpPic;
```

Подготовка надписей

METAPOST позволяет размещать на картинке надписи.

METAPOST может использовать внешние программы для подготовки надписей — troff и TEX.

Изучение всех возможностей систем troff и TEX не входит в наши планы. Мы совершенно обойдём вниманием troff, но обсудим простейшие и важнейшие возможности TEX.

Введение в TEX

TEX — система компьютерной вёрстки, созданная уже упомянутым профессором Кнутом.

TEX произносится отнюдь не как [текс], а как [тех].

Особенностью TEX является превосходная поддержка набора математических формул.

Между прочим, это руководство подготовлено в TEX.

Назначение и особенности TEX

Назначение системы TEX примерно то же самое, что и у известных программ Microsoft Word, OpenOffice Writer, Adobe Pagemaker, QuarkXPress и подобных.

Однако, подобно системе METAPOST, TEX не является системой WYSIWYG. Будущая вёрстка готовится как программа, в которой обычный текст перемежается с управляющими командами (кстати, макрокомандами), которые влияют на вид документа. Для получения документа, который можно печатать, требуется компиляция.

Это обстоятельство несколько затрудняет работу, однако именно оно является причиной великолепного полиграфического качества документов, подготовленных в TEX, и стилевого их единообразия.

Простейшие TEXнические решения

`Просто текст`	Просто текст
`\textit{Курсивный} текст`	*Курсивный* текст
`\textbf{Жирный} текст`	**Жирный** текст
`\textit{\textbf{Жирный курсив}}`	***Жирный курсив***
`\texttt{Моноширинный} текст`	`Моноширинный текст`
`Формула $x$`	Формула x

Простейшие TEXнические решения

`$a+b$`	$a+b$
`$a-b=2$`	$a-b=2$
`$\frac12$`	$\frac{1}{2}$
`$\frac1{12}$`	$\frac{1}{12}$
`$\frac{123}{456}$`	$\frac{123}{456}$
`$\sqrt2$`	$\sqrt{2}$
`$\root3\of2$`	$\sqrt[3]{2}$
`$\alpha\beta\gamma\ldots\omega$`	$\alpha\beta\gamma\ldots\omega$
`$z_2$`	z_2
`$z^{12}$`	z^{12}
`$\cos^2\theta+\sin^2\theta=1$`	$\cos^2\vartheta+\sin^2\vartheta=1$

Простейшие TEXнические решения

`$x\in[-\pi;\pi)$`	$x \in [-\pi;\pi)$
`$x\in\{2,3,5,7,11,13,17,\ldots\}$`	$x \in \{2,3,5,7,11,13,17,\ldots\}$
`$\mathbb N\subset\mathbb Z$`	$\mathbb{N} \subset \mathbb{Z}$
`$\|\tg x\|>\|\sin x\|$ при $x>0$`	$\|\operatorname{tg} x\| > \|\sin x\|$ при $x > 0$
`$\sqrt{ab}\leqslant\frac{a+b}2$`	$\sqrt{ab} \leqslant \frac{a+b}{2}$
`$\sqrt{ab}\geqslant\frac{2ab}{a+b}$`	$\sqrt{ab} \geqslant \frac{2ab}{a+b}$

Более сложный пример: формула Коши

$$f(z) = \frac{1}{2\pi i} \oint\limits_{\partial U_\varepsilon(z)} \frac{f(\zeta)\,d\zeta}{\zeta - z}$$

```
$f(z)=\frac1{2\pi i}
    \oint\limits_{\partial U_\varepsilon(z)}
    \frac{f(\zeta)\,d\zeta}{\zeta-z}$
```

Более сложный пример: формула Коши

$$f(z) = \frac{1}{2\pi i} \oint\limits_{\partial U_\varepsilon(z)} \frac{f(\zeta)\,d\zeta}{\zeta - z}$$

```
$f(z)=\frac1{2\pi i}
    \oint\limits_{\partial U_\varepsilon(z)}
    \frac{f(\zeta)\,d\zeta}{\zeta-z}$
```

Более сложный пример: формула Коши

$$f(z) = \frac{1}{2\pi i} \oint\limits_{\partial U_\varepsilon(z)} \frac{f(\zeta)\,d\zeta}{\zeta - z}$$

```
$f(z)=\frac1{2\pi i}
    \oint\limits_{\partial U_\varepsilon(z)}
    \frac{f(\zeta)\,d\zeta}{\zeta-z}$
```

Более сложный пример: формула Коши

$$f(z) = \frac{1}{2\pi i} \oint\limits_{\partial U_\varepsilon(z)} \frac{f(\zeta)\,d\zeta}{\zeta - z}$$

```
$f(z)=\frac1{2\pi i}
    \oint\limits_{\partial U_\varepsilon(z)}
    \frac{f(\zeta)\,d\zeta}{\zeta-z}$
```

Более сложный пример: формула Коши

$$f(z) = \frac{1}{2\pi i} \oint\limits_{\partial U_\varepsilon(z)} \frac{f(\zeta)\,d\zeta}{\zeta - z}$$

```
$f(z)=\frac1{2\pi i}
    \oint\limits_{\partial U_\varepsilon(z)}
    \frac{f(\zeta)\,d\zeta}{\zeta-z}$
```

Где почитать про TEX

Конечно, всех этих примеров недостаточно, чтобы освоить все TEXнические тонкости.

Для более подробного знакомства с системой TEX предлагаем другие источники:

- Всё про TEX
  Кнут, Д. Е.
  Протвино : АО RDTEX, 1993. — ISBN 5-900614-01-8
- Всё про TEX
  Кнут, Дональд Э.
  Москва : Вильямс, 2003. — ISBN 5-8459-0382-3 (рус.)

Где почитать про TEX *(продолжение)*

- Набор и вёрстка в системе LATEX
  Львовский, С. М.
  Москва : МЦНМО, 2003. — ISBN 5-94057-091-7

TEXовские фрагменты как объекты picture

TEXовские вставки с точки зрения METAPOST являются объектами типа picture. Код на языке TEX следует поместить между словами btex и etex:

btex ⟨*TEXовский код*⟩ etex

Например:

```
btex $\frac1{\sqrt3}$ etex
```

Препроцессор

Перед тем как METAPOST приступает к компиляции исходного текста, этот текст подвергается обработке препроцессором.

Препроцессор отыскивает фрагменты `btex … etex`, извлекает их в TEXовский файл, затем файл компилируется TEXом. Результат этой компиляции обрабатывается специальной программой, преобразующей его в код на языке METAPOST. Этот код вставляется в исходный текст рисунка как выражение типа `picture`. Компилятор METAPOST видит только лишь результат этой закулисной деятельности.

TEXовские ошибки

Нет необходимости вникать в эти подробности, если TEXовский код в порядке. Если же он содержит ошибки с точки зрения TEX, METAPOST выдаёт ошибку

```
! Unable to make mpx file.
```

Для исправления таких ошибок необходимы определённые TEXнические познания.

TEXовские ошибки

Нет необходимости вникать в эти подробности, если TEXовский код в порядке. Если же он содержит ошибки с точки зрения TEX, METAPOST выдаёт ошибку

```
! Невозможно создать mpx-файл.
```

Для исправления таких ошибок необходимы определённые TEXнические познания.

Вставка TEXовских фрагментов в METAPOST

Если картинку `btex … etex` нужно включить в рисунок, можно использовать команду `draw`:

```
draw btex $\frac1{\sqrt3}$ etex;
```

Однако такая команда размещает картинку таким образом, что левый нижний угол картинки попадает в начало координат. Для более точного размещения картинку нужно сдвинуть.

Команда label

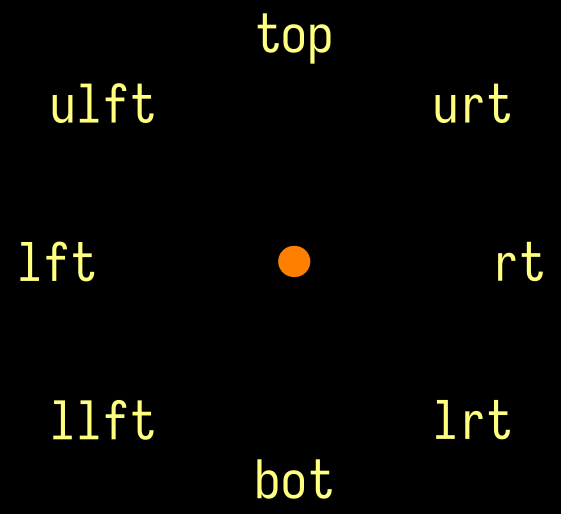
В библиотеке plain.mp определена макрокоманда label, которая позволяет разместить TEXовскую надпись либо по центру в данной точке рисунка, либо поблизости в стороне от данной точки. Поддерживаются восемь положений: справа, сверху, слева и снизу, и ещё четыре промежуточных.

Синтаксис команды:

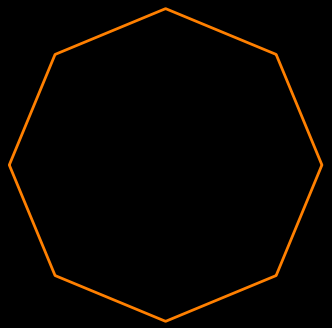
label(⟨*картинка*⟩, ⟨*точка*⟩)
label.⟨*суффикс*⟩(⟨*картинка*⟩, ⟨*точка*⟩)

Команда `label.`⟨*суффикс*⟩

В команде `label` за положение ⟨*картинки*⟩ по отношению к ⟨*точке*⟩ отвечает ⟨*суффикс*⟩. Допустимые ⟨*суффиксы*⟩:



Пример размещения надписей

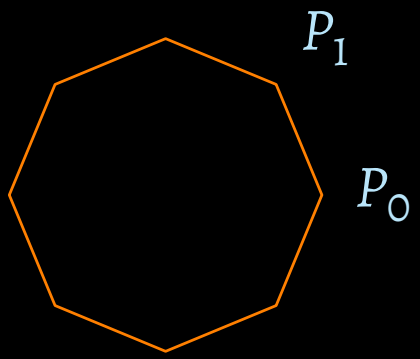


```
draw
    for d:=0 upto 7:
        cm*dir(45d)--
    endfor
    cycle;
```

Пример размещения надписей

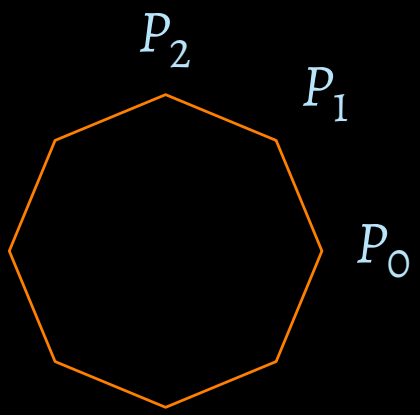

```
label.rt(
        btex $P_0$ etex,
        cm*right
    );
```

Пример размещения надписей



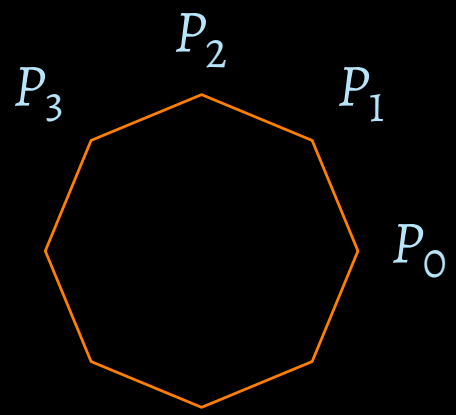

```
label.urt(
        btex $P_1$ etex,
        cm*dir 45
    );
```

Пример размещения надписей



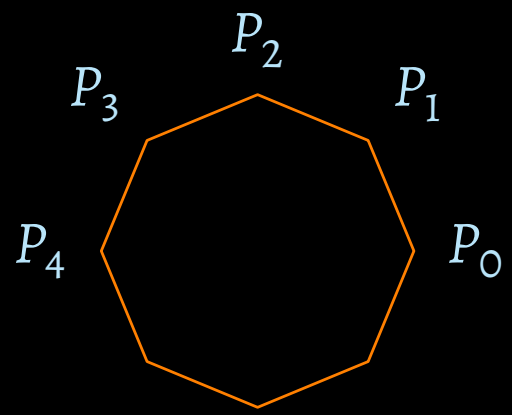

```
label.top(
        btex $P_2$ etex,
        cm*up
    );
```

Пример размещения надписей



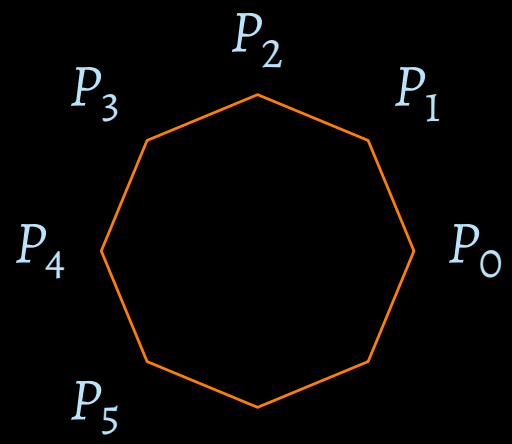

```
label.ulft(
        btex $P_3$ etex,
        cm*dir 135
    );
```

Пример размещения надписей



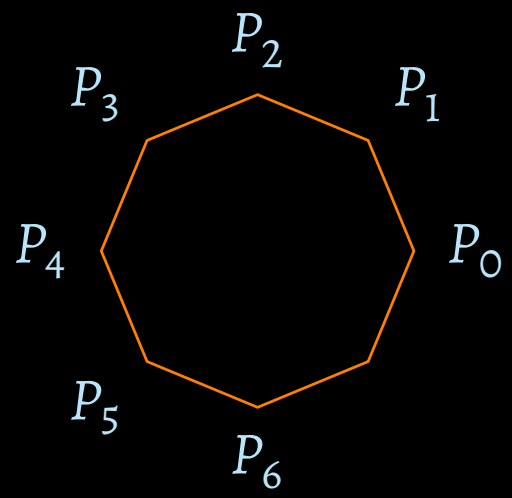

```
label.lft(
        btex $P_4$ etex,
        cm*left
    );
```

Пример размещения надписей



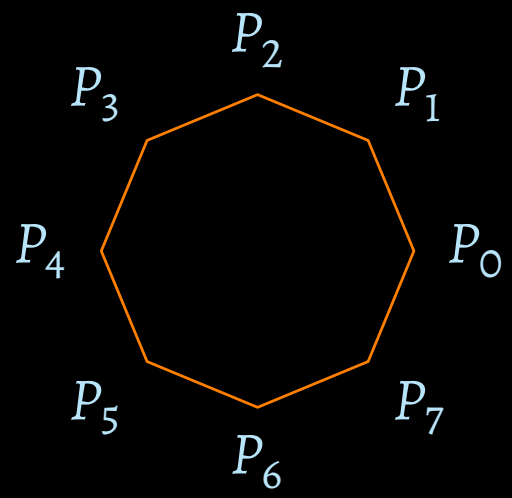

```
label.llft(
        btex $P_5$ etex,
        cm*dir 225
    );
```

Пример размещения надписей



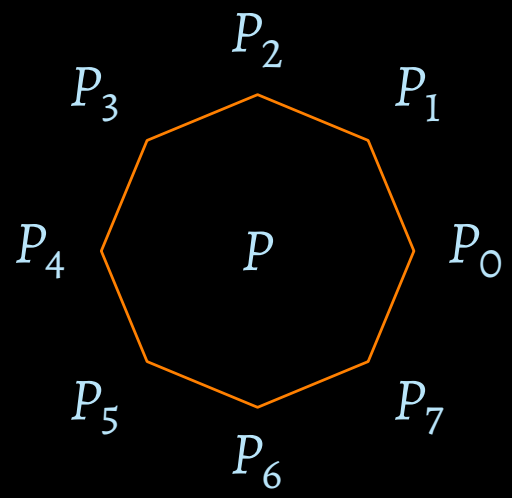

```
label.bot(
        btex $P_6$ etex,
        cm*down
    );
```

Пример размещения надписей




```
label.lrt(
        btex $P_7$ etex,
        cm*dir 315
    );
```

Пример размещения надписей




```
label(
        btex $P$ etex,
        origin
    );
```

Заключение

Работа над этой презентацией не окончена. Актуальная версия расположена по адресу http://mech.math.msu.su/~shvetz/54/inf/metapost/mpshort.pdf.

При создании настоящего руководства использовались следующие технологии: LuaLaTeX, METAPOST.

Изображение капеллы Notre Dame du Haut в Роншане архитектора Ле Корбюзье (Le Corbusier) взято по адресу http://home.broadpark.no/~lhellemo/gallery/architecture/Le_Corbusier_front.jpg

Логотип механико-математического факультета МГУ имени М. В. Ломоносова первоначально был создан С. В. Голованем в системе METAFONT и адаптирован автором для METAPOST.

Пиктограммы взяты с сайта http://aiga.org.

Фотография В. И. Арнольда взята на Википедии: https://commons.wikimedia.org/wiki/File:Vladimir_Arnold-1.jpg.

Спасибо за внимание!

http://mech.math.msu.su/~shvetz/54/inf/metapost/mpshort.pdf

https://github.com/urbic/mpshort

8cdcbe24c8ccd240f19306d18526e4ecfe2e3897

2022-12-21T12:16:52+03:00

shvetz.anton@gmail.com