Лекция 2. Графика: основные принципы

В *Mathematica* существует всего два основных типа графических объектов: **Graphics** и **Graphics3D**. Имеется ДВА ОСНОВНЫХ СПОСОБА создавать графические объекты. **Первый способ** (и, на самом деле, основной) предназначен для создания графики, состоящей из набора простых графических элементов (так называемых **примитивов**), которые описываются явно (скажем, точки задаются своими координатами, сфера - своими центром и радиусом). При использовании **второго способа**, соответстующие примитивы порождаются самим пакетом исходя из заданной функции, списка или другого объекта, подлежащего визуализации, с помощью специальных команд, таких как Plot, ParametricPlot и т.п.

Второй способ

Рассмотрим сначала второй способ. *Mathematica* располагает большим арсеналом средств для построения графиков, поверхностей, линий уровня, графического представления разнообразных данных и т.п. Познакомимся с основными командами.

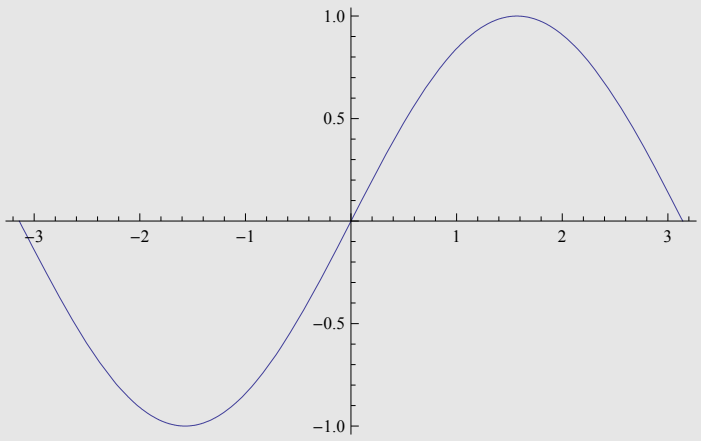
Двумерная графика

- **Команда Plot**

График функции строится с помощью команды **Plot**.

In[1]:=
```
Plot[Sin[x], {x, -π, π}]
```

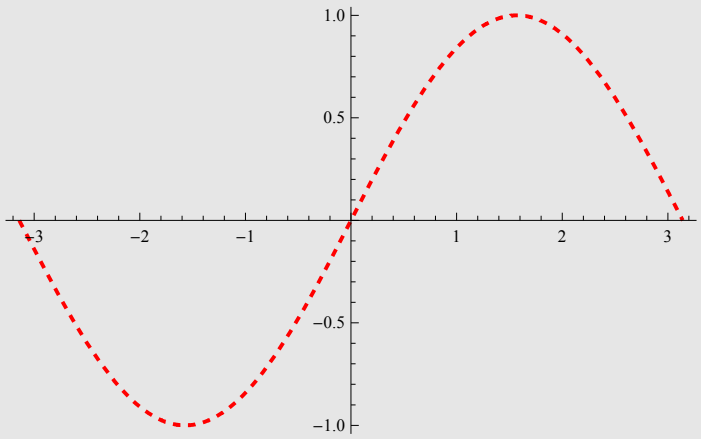
Out[1]=



В командах типа **Plot** можно использовать опции обычной графики (мы познакомимся с соответствующими командами позже). Однако есть еще опции специфические для команды **Plot:** специальные способы управления, главные из которых **PlotStyle** и **ColorFunction**.

In[2]:=
```
Plot[Sin[x], {x, -π, π}, PlotStyle → {RGBColor[1, 0, 0], Dashed, Thick}]
```

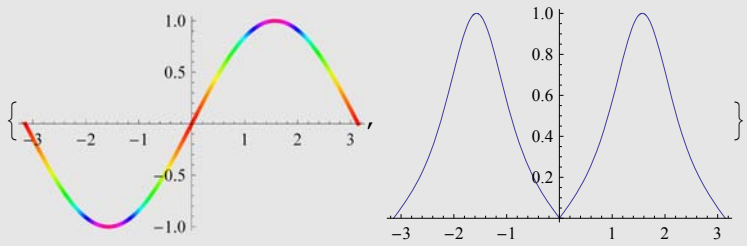
Out[2]=



Опция **ColorFunction** служит для управления цветом изображаемого графика. Переменные этой функции фиксированы (для каждого типа графики - свои, для обычного **Plot**, например, это **x, y**). В качестве примера, раскрасим график в соответствие с кривизной соответствующей плоской кривой.
Здесь вам придется вспомнить формулу для кривизны плоской кривой, заданной в виде графика функции. Обратите внимание, что **cur** в качестве параметра имеет еще и имя функции!

In[3]:=
```
ClearAll[g];
cur[f_][t_] := Abs[D[f[tt], tt, tt]] / (1 + D[f[tt], tt]^2)^(3/2) /. tt → t;
g[t_] := cur[Sin][t];
{Plot[Sin[x], {x, -π, π}, PlotStyle → {Thick},
  ColorFunction → Function[{x, y}, Hue[g[x] / 1.1]], ColorFunctionScaling → False],
 Plot[g[t], {t, -π, π}]}
```

Out[6]=

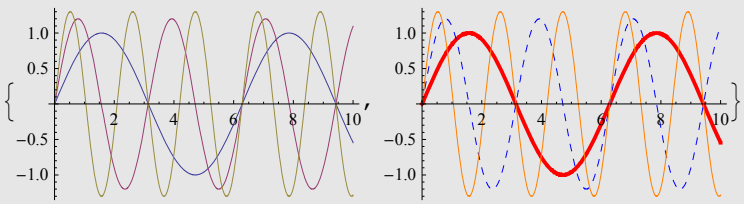


Замечание. На самом деле, чтобы картина была реалистичной, надо отмасштабировать кривизну, чтобы она менялась между нулем 0 и 1 и не доходила до 1. Здесь мы пользуемся тем, что кривизна графика синуса меняется он нуля (красный) до единицы (синий). Опция **ColorFunctionScaling → False** отключает автоматическое масштабирование функции, отвечающей за цвет, которое затемняет дело.

Перечислим теперь встоенные возможности функции **Plot**. Во-первых, можно рисовать сразу несколько графиков, причем они автоматически получаются разноцветными. Можно и самим регулировать их вид (с помощью **PlotStyle**).

In[7]:=
```
{Plot[{Sin[x], 1.2 Sin[2 x], 1.3 Sin[3 x]}, {x, 0, 10}],
 Plot[{Sin[x], 1.2 Sin[2 x], 1.3 Sin[3 x]}, {x, 0, 10},
  PlotStyle → {{Red, Thick}, {Dashed, Blue}, {Orange}}]}
```

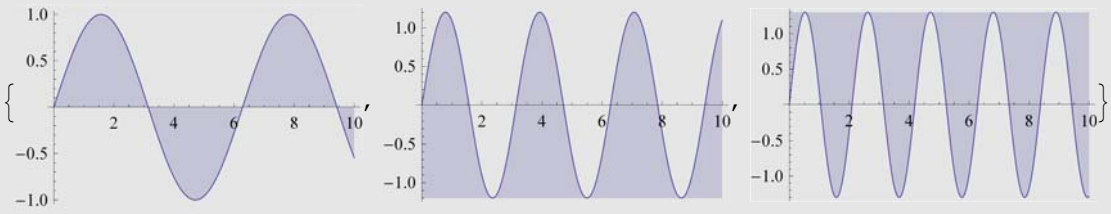
Out[7]=



Можно также закрашивать области, ограниченные графиком (графиками). Это исполняется с помощью опции **Filling**, значение которой устанавливается в тип заполнения (до оси, подграфик, надграфик).

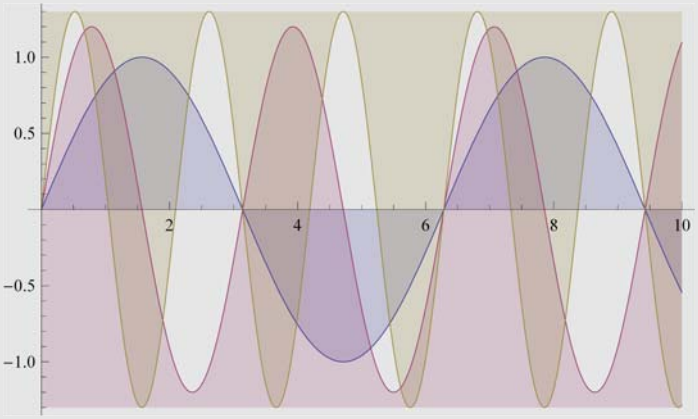
In[8]:=
```
{Plot[Sin[x], {x, 0, 10}, Filling → {1 → Axis}],
 Plot[1.2 Sin[2 x], {x, 0, 10}, Filling → {1 → Bottom}],
 Plot[1.3 Sin[3 x], {x, 0, 10}, Filling → {1 → Top}]}
```

Out[8]=



In[9]:=
```
Plot[{Sin[x], 1.2 Sin[2 x], 1.3 Sin[3 x]},
 {x, 0, 10}, Filling → {1 → Axis, 2 → Bottom, 3 → Top}]
```

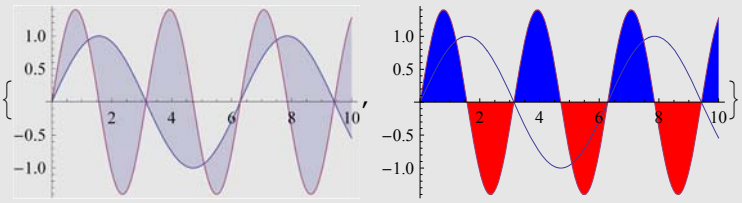
Out[9]=



Более хитрый вид заполнения : для 2-го объекта выбрано заполнение до 1-го. Можно также добавить директивы, как именно заполнять.

In[10]:=
```
{Plot[{Sin[x], 1.4 Sin[2 x]}, {x, 0, 10}, Filling → {2 → {1}}],
 Plot[{Sin[x], 1.4 Sin[2 x]}, {x, 0, 10}, Filling → {2 → {0, {Red, Blue}}}]}
```

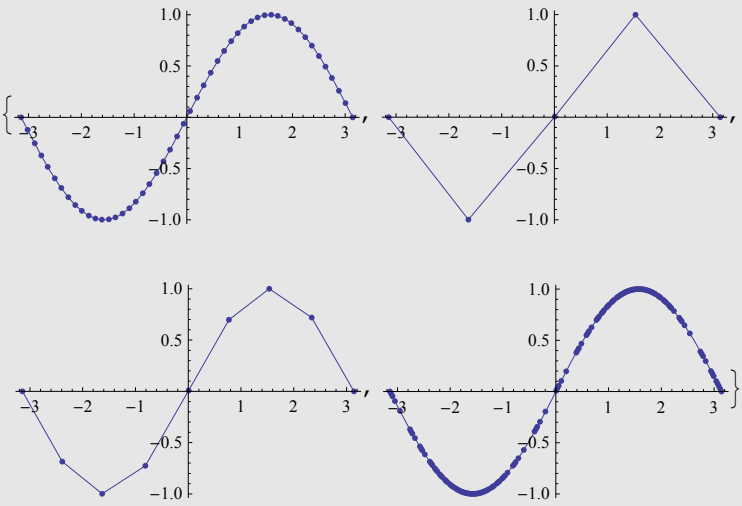
Out[10]=



Следует понимать, что **Plot** строит ломаную, при этом *Mathematica* следит за тем, чтобы на участках высокой кривизны количество точек аппроксимации было выбрано достаточно большим. Вместе с тем, параметрами, влияющими на точность картинки, можно управлять.
Можно явно указать сколько точек участвуют в построении ломаной, изображающей кривую. Современные версии *Mathematica* сами следят за качеством графика, поэтому **PlotPoints** в двумерной графике - это начальное колическтво точек апрокисмации (по умолчанию равен 50). Если картинка *Mathematica* не нравится, она сама подразобьет кривую столько раз, сколько ей захочется. Чтобы запретить это, нужно ограничить рекурсию с помощью опции **MaxRecursion** (которая устанавливается в количество итераций). Чтобы увидеть точки, нужно использовать опцию **Mesh**.

In[11]:=
```
{Plot[Sin[x], {x, -π, π}, Mesh → All, MaxRecursion → 0],
 Plot[Sin[x], {x, -π, π}, PlotPoints → 5, Mesh → All, MaxRecursion → 0],
 Plot[Sin[x], {x, -π, π}, PlotPoints → 5, Mesh → All, MaxRecursion → 1],
 Plot[Sin[x], {x, -π, π}, PlotPoints → 5, Mesh → All]}
```

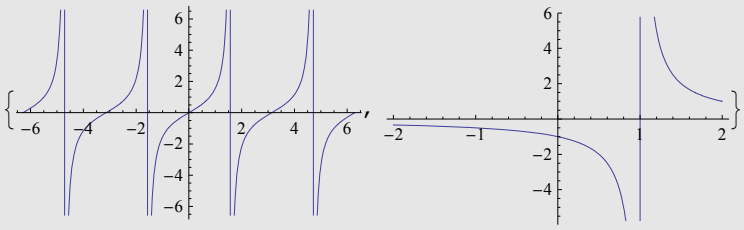
Out[11]=



Mathematica последовательно соединяет отрезками соседние (по x-координате) точки, в результате чего могут получиться неожиданные эффекты :

In[12]:=
```
{Plot[Tan[x], {x, -2 π, 2 π}], Plot[1 / (x - 1), {x, -2, 2}]}
```

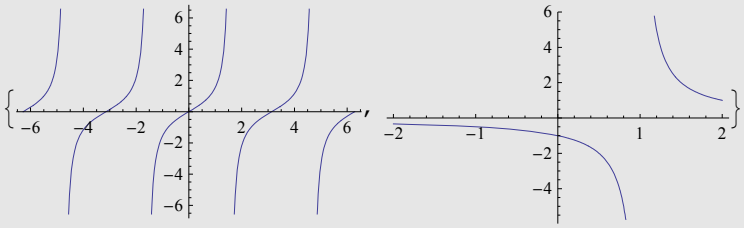
Out[12]=



В этом случае особые точки можно исключить из рассмотрения. Для этого служит опция **Exclusions,** которая задает условие на исключительные точки.

In[13]:=
```
{Plot[Tan[x], {x, -2 π, 2 π}, Exclusions → {Cos[x] == 0}],
  Plot[1 / (x - 1), {x, -2, 2}, Exclusions → {x == 1}]}
```

Out[13]=

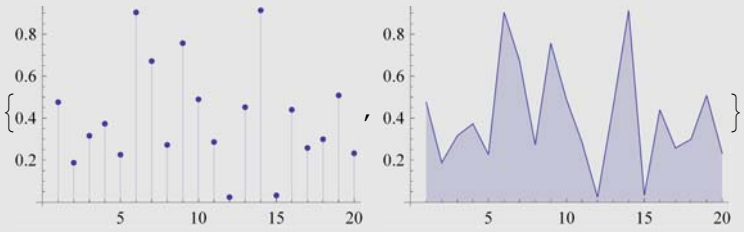


■ Другие способы представления числовых данных

К двумерным способам представления числовых данных относятся **ListPlot**, **ListLinePlot**, **ArrayPlot**, **PolarPlot**, **ParametricPlot**, **ContourPlot**, **DensityPlot** и **RegionPlot**.
Первые три служат для изображения дискретных данных (одномерных или двумерных). Вот пример графического отображения одномерных данных.

In[14]:=
```
data = Table[Random[], {20}];
 {ListPlot[data, Filling → Axis], ListLinePlot[data, Filling → Axis]}
```

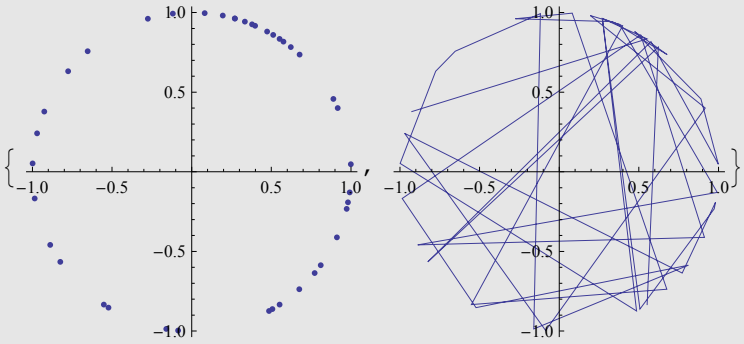
Out[15]=



А вот пример для двумерных данных. Обратите внимание, что ListLinePlot просто соединяет последовательные точки линией.

In[16]:=
```
data = Table[{Cos[#], Sin[#]} &[Random[Real, 2 π]], {i, 1, 40}];
{ListPlot[data, AspectRatio → Automatic],
 ListLinePlot[data, AspectRatio → Automatic]}
```

Out[17]=



ArrayPlot изображает числа строки в виде раскрашенных квадратиков. По умолчанию они черно-белые, но можно их и раскрасить.

In[18]:=
```
ArrayPlot[Transpose[data]]
```

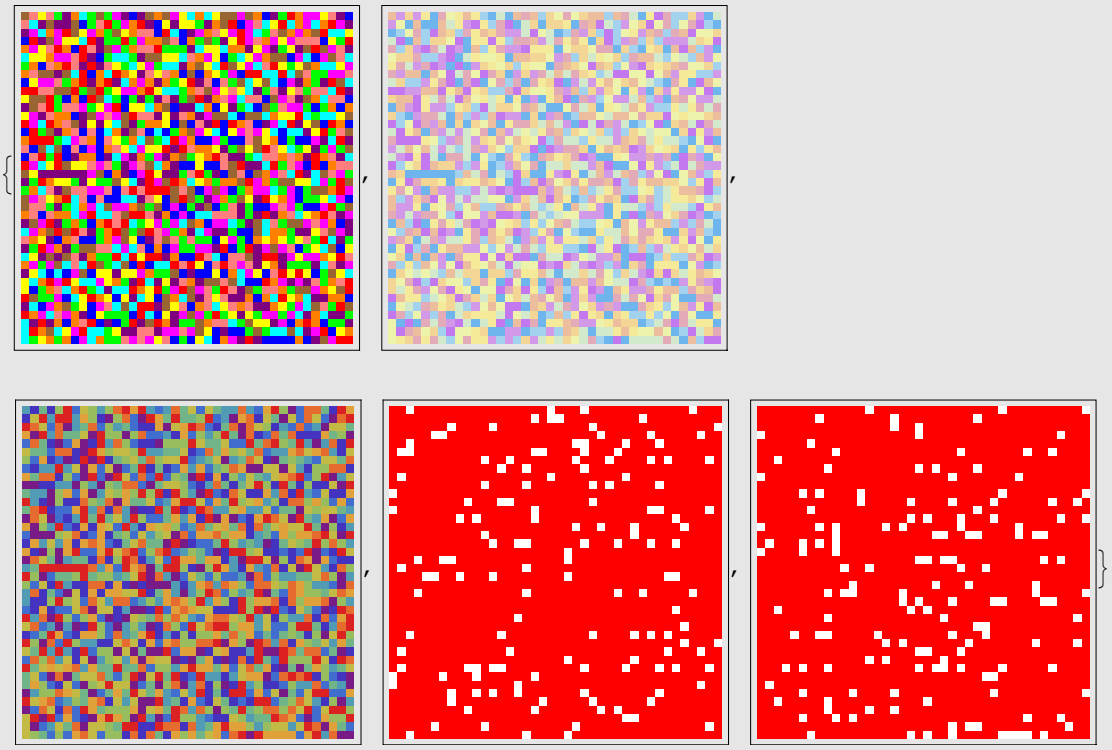
Out[18]=

Можно задать цвета явно с помощью правил **ColorRules**, можно использовать **ColorFunction**. Мы нарисуем число π.
Здесь нужно обратить внимание на синтаксис подпрограммы, оформленной с помощью Module.
Посмотрите в Help, что делает функция RealDigits с четырьмя аргументами, это очень важно для нашего примера.

In[19]:=
```
carpet[x_, m_, n_] := Module[{res, xx, i},
    (* Эта программа вычисляет цифры числа x и организовывает их в массив*)
    xx = N[x, m n];
    res = {RealDigits[xx, 10, m][[1]]};
    xx = xx / 10^RealDigits[xx,10,m][[2]];
    For[i = 1, i ≤ n - 1, i++,
     res = res ~Join~ {RealDigits[xx, 10, m, -m i - 1][[1]]}
    ];
    res
   ];
rules = {0 → Red, 1 → Pink, 2 → Orange, 3 → Brown,
    4 -> Yellow, 5 → Green, 6 → Cyan, 7 → Blue, 8 → Magenta, 9 → Purple};
rules1 = {0 → Red, 1 → Red, 2 → Red, 3 → Red, 4 → White,
    5 → Red, 6 → Red, 7 → Red, 8 → Red, 9 → Red};
rules2 = {0 → Red, 1 → Red, 2 → Red, 3 → Red, 4 → Red, 5 → Red,
    6 → Red, 7 → White, 8 → Red, 9 → Red};
```

In[23]:=
```
{ArrayPlot[carpet[π, 40, 40], ColorRules → rules],
 ArrayPlot[carpet[π, 40, 40], ColorFunction → "Pastel"],
 ArrayPlot[carpet[π, 40, 40], ColorFunction → "Rainbow"],
 ArrayPlot[carpet[π, 40, 40], ColorRules → rules1],
 ArrayPlot[carpet[π, 40, 40], ColorRules → rules2]}
```

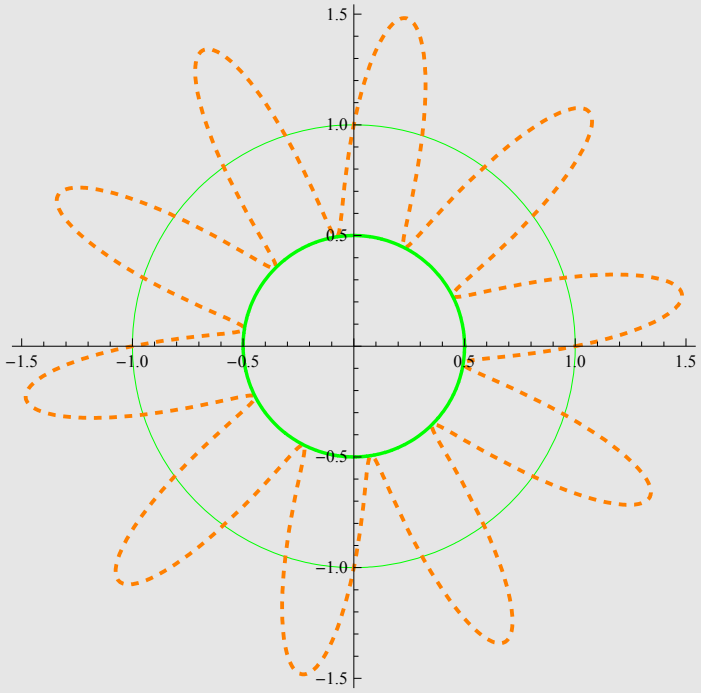
Out[23]=



PolarPlot удобен для рисования кривых, заданных в полярных координатах. Имеется также его модификация **ListPolarPlot**.

In[24]:=
```
PolarPlot[{1, 1 + 1 / 2 Sin[10 t], 1 / 2}, {t, 0, 2 Pi},
 PlotStyle → {Green, {Dashed, Thick, Orange}, {Thickness[0.005], Green}}]
```

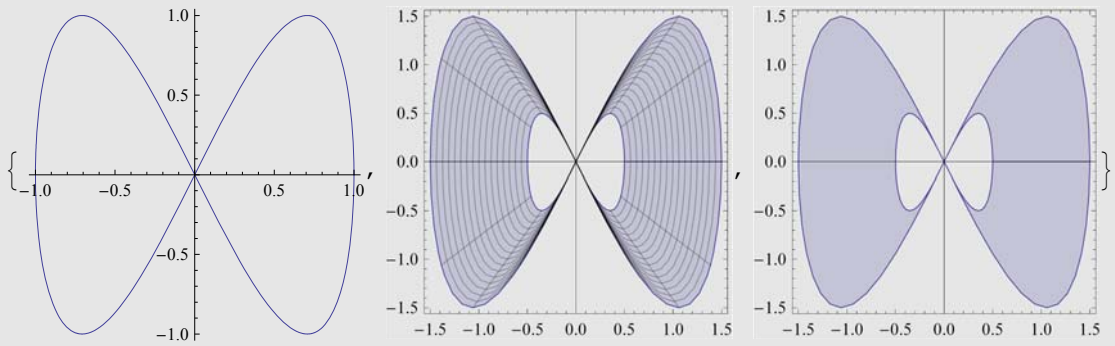
Out[24]=



Универсальный инструмент для рисования кривых - это, конечно, **ParametricPlot**. Эта фукнкция позволяет рисовать однопараметрические семейства или области.

In[25]:=
```
{ParametricPlot[{Cos[x], Sin[2 x]}, {x, 0, 2 π}],
 ParametricPlot[r {Cos[x], Sin[2 x]}, {x, 0, 2 π}, {r, 0.5, 1.5}],
 ParametricPlot[r {Cos[x], Sin[2 x]}, {x, 0, 2 π}, {r, 0.5, 1.5}, Mesh → None]}
```

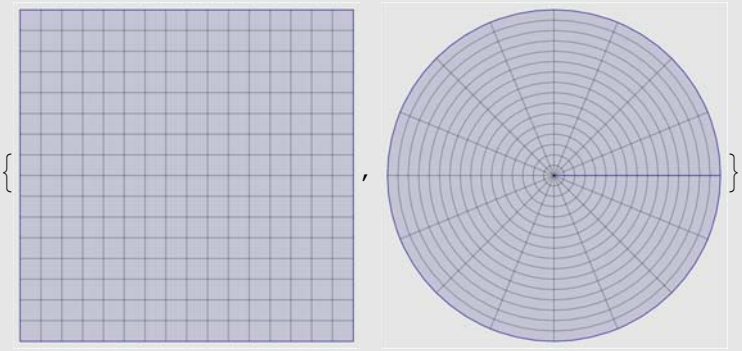
Out[25]=



Можно использовать это для рисования, например, координатных линий. Опции **Axes** и **Frame** позволяют убрать декартовы оси и рамку вокруг картинки, которые в данном примере могут сбить с толка.

In[26]:=
```
{ParametricPlot[{x, y}, {x, -1, 1}, {y, -1, 1}, Axes → False, Frame → False],
 ParametricPlot[{r Cos[φ], r Sin[φ]},
  {φ, 0, 2 π}, {r, 0, 1}, Axes → False, Frame → False]}
```

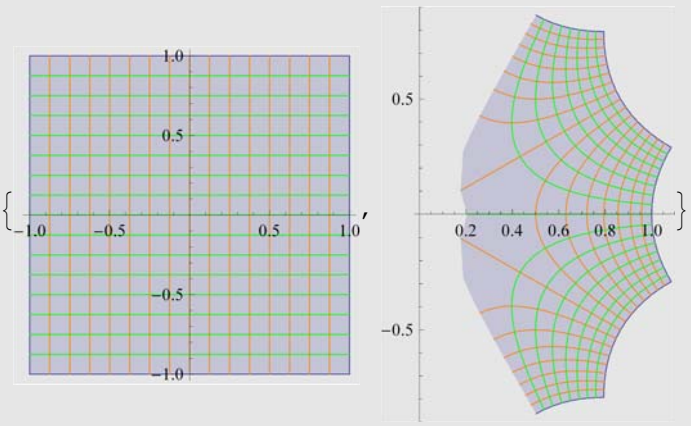
Out[26]=



Или для наглядного изображения отображений из плоскости в плоскость.

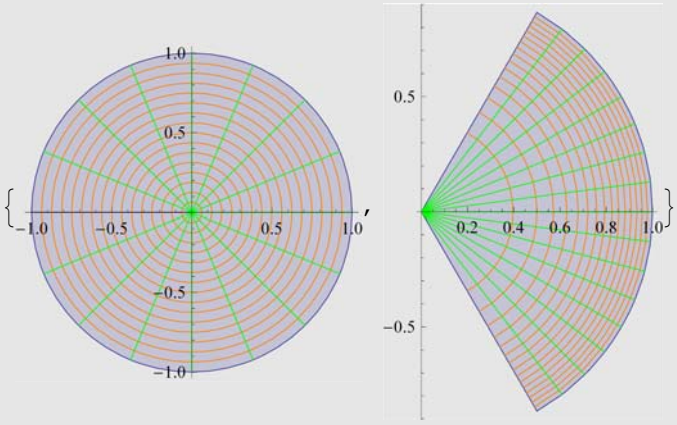
In[27]:=
```
F[u_, v_] := {Re[(u + i v)^(1/3)], Im[(u + i v)^(1/3)]};
{ParametricPlot[{x, y}, {x, -1, 1}, {y, -1, 1}, Frame → False,
  MeshStyle → {Orange, Green}], ParametricPlot[F[u, v], {u, -1, 1},
  {v, -1, 1}, Frame → False, MeshStyle → {Orange, Green}, AxesOrigin → {0, 0}]}
```

Out[28]=



In[29]:=
```
FF[r_, φ_] := {Re[r^(1/3) e^(i φ/3)], Im[r^(1/3) e^(i φ/3)]};
{ParametricPlot[{r Cos[φ], r Sin[φ]}, {r, 0, 1}, {φ, -π, π}, Frame → False,
  MeshStyle → {Orange, Green}], ParametricPlot[FF[r, φ], {r, 0, 1},
  {φ, -π, π}, Frame → False, MeshStyle → {Orange, Green}, AxesOrigin → {0, 0}]}
```

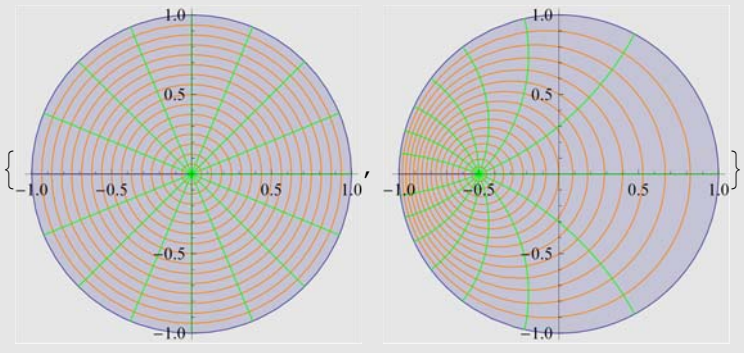
Out[30]=



Следующий пример показывает, что делают дробно-линейные преобразования комплексной плоскости

In[31]:=
```
DLP[r_, φ_] := Module[{z, z0, w},
   z = r e^(i φ);
   z0 = 1 / 2;
   w = (z - z0) / (1 - z Conjugate[z0]);
   {Re[w], Im[w]}
  ];
{ParametricPlot[{r Cos[φ], r Sin[φ]}, {r, 0, 1},
  {φ, -π, π}, Frame → False, MeshStyle → {Orange, Green}],
 ParametricPlot[DLP[r, φ], {r, 0, 1}, {φ, -π, π}, Frame → False,
  PlotPoints → 50, MeshStyle → {Orange, Green}, AxesOrigin → {0, 0}]}
```

Out[32]=



Снова можно задавать осмысленные функции для раскраски кривой. Например, кривизну. Напомним, что красный соответствует нулю, и цвет меняется в синюю сторону при возрастании.

In[33]:=
```
cur[x_, y_][t_] := Abs[D[x[tt], tt, tt] D[y[tt], tt] - D[y[tt], tt, tt] D[x[tt], tt]] /
    (D[x[tt], tt]^2 + D[y[tt], tt]^2)^(3/2) /. tt → t;
```

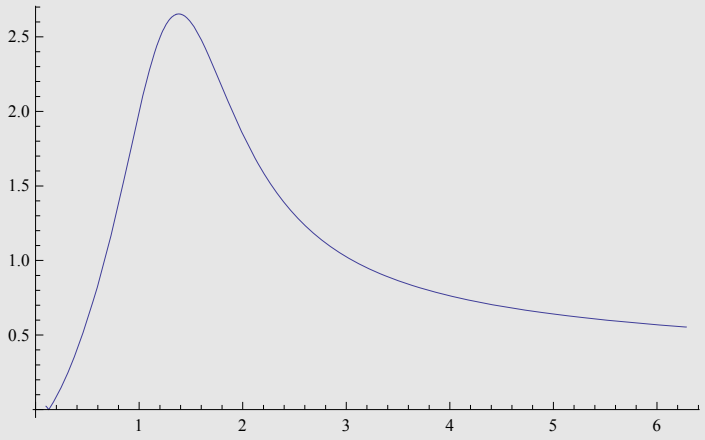
In[34]:=
```
ClearAll[gg];
xx[t_] := Log[t] Cos[t];
yy[t_] := Log[t] Sin[t];
gg[t_] := cur[xx, yy][t] ;
```

In[38]:=
```
mm = NMaximize[{gg[t], t ≥ 1/2}, t][[1]]
```

Out[38]=
```
2.65323
```

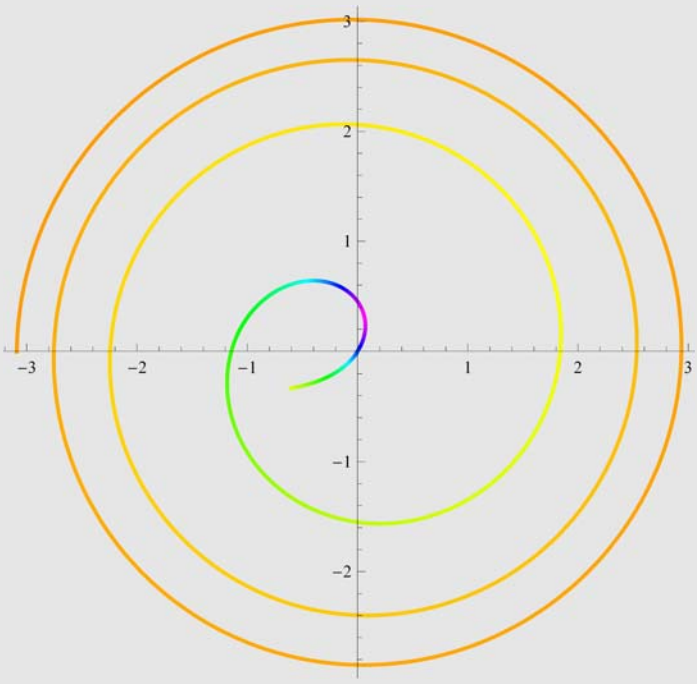
In[39]:=
```
Plot[gg[t], {t, 0.1, 2 π}]
```

Out[39]=



In[40]:=
```
ParametricPlot[{xx[t], yy[t]}, {t, 1 / 2, 7 π}, PlotStyle → Thick,
 ColorFunction → Function[{x, y, u}, Hue[gg[u] / 1.2 mm]], ColorFunctionScaling → False]
```

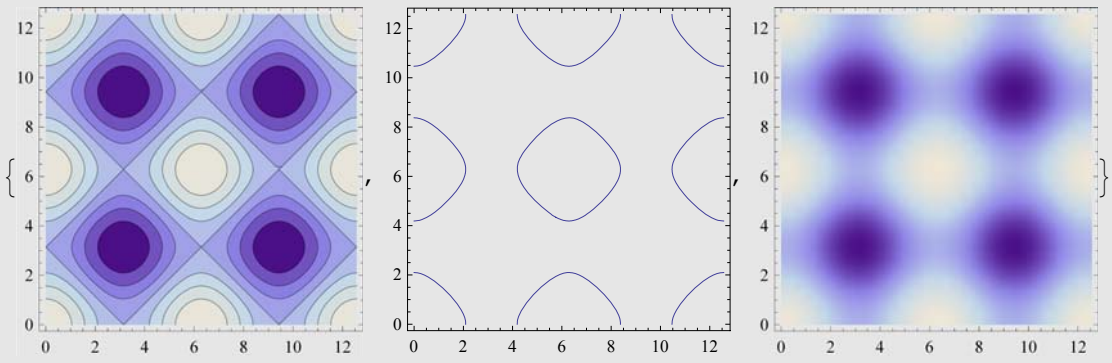
Out[40]=



Команды **ContourPlot** и **DensityPlot** служат для рисования неявно заданных кривых и функций.

In[41]:=
```
{ContourPlot[Cos[x] + Cos[y], {x, 0, 4 Pi}, {y, 0, 4 Pi}],
 ContourPlot[Cos[x] + Cos[y] == 1 / 2, {x, 0, 4 Pi}, {y, 0, 4 Pi}],
 DensityPlot[Cos[x] + Cos[y], {x, 0, 4 Pi}, {y, 0, 4 Pi}]}
```

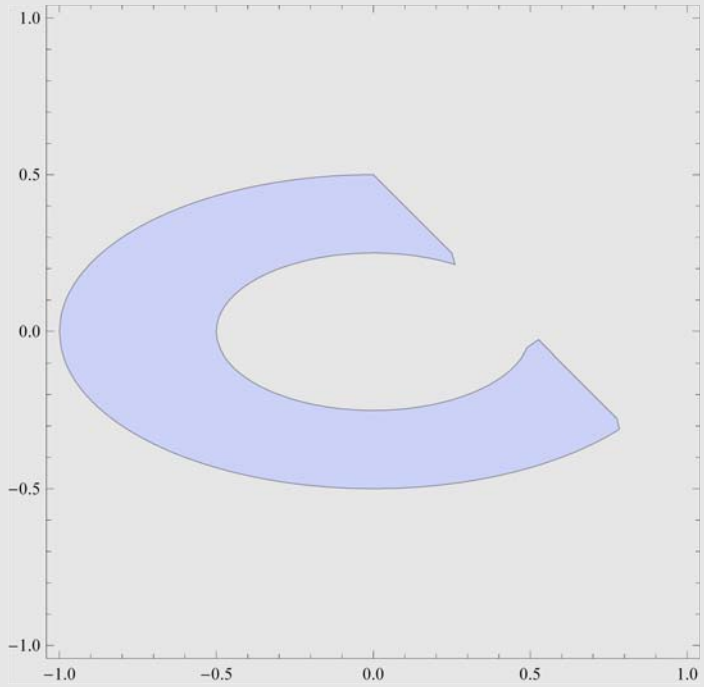
Out[41]=



Наконец, **RegionPlot** изображает области, заданные неравенствами.

In[42]:=
```
RegionPlot[1 / 4 ≤ x^2 + (2 y)^2 ≤ 1 && x + y ≤ 1 / 2,
 {x, -1, 1}, {y, -1, 1}, AspectRatio → Automatic]
```

Out[42]=



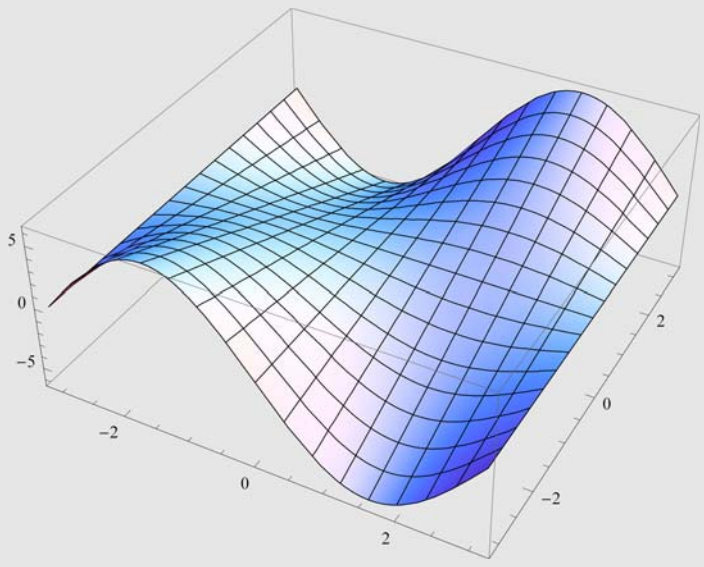
Трехмерная графика

Основные команды

Практически все команды двумерной графики имеют аналоги в трехмерной. Снова расстorим сначала простейшую команду рисования графика функции двух переменных **Plot3D**.

In[43]:=
```
Plot3D[ y Sin[x] + y Sin[x], {x, -π, π}, {y, -π, π}]
```

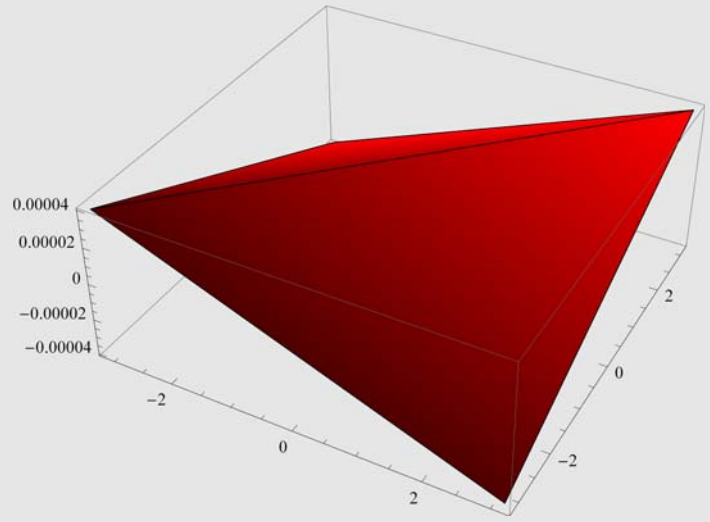
Out[43]=



Точнее

In[44]:=
```
Plot3D[ y Sin[x] + y Sin[x], {x, -π, π}, {y, -π, π},
 PlotStyle → FaceForm[Red, Blue], PlotPoints → 2, Mesh → All, MaxRecursion → 0]
```

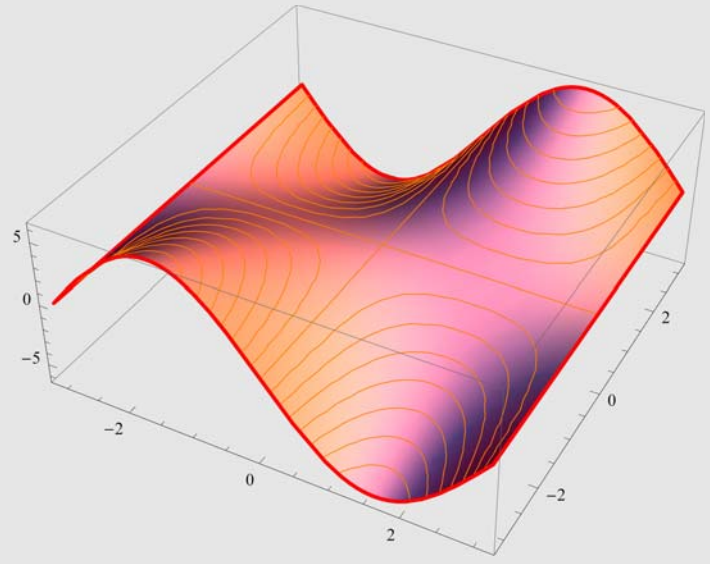
Out[44]=



Снова можно пользоваться стандартными опциями трехмерной графики, снова есть **PlotStyle** для директив. Из новых опций обратим внимание на **BoundaryStyle** (приписывается графическая директива, описывающая граничную линию), **Exclusions** и **ExclusionStyle** (исключения из рисунка условие и способ прорисовки ограничивающей линии) и **RegionFunction** (функция от стандартного набора переменных), **MeshFunction** (приписывается функция или функции, линии уровня которых задают линии на поверхности), **Mesh** приписывается количество линий.

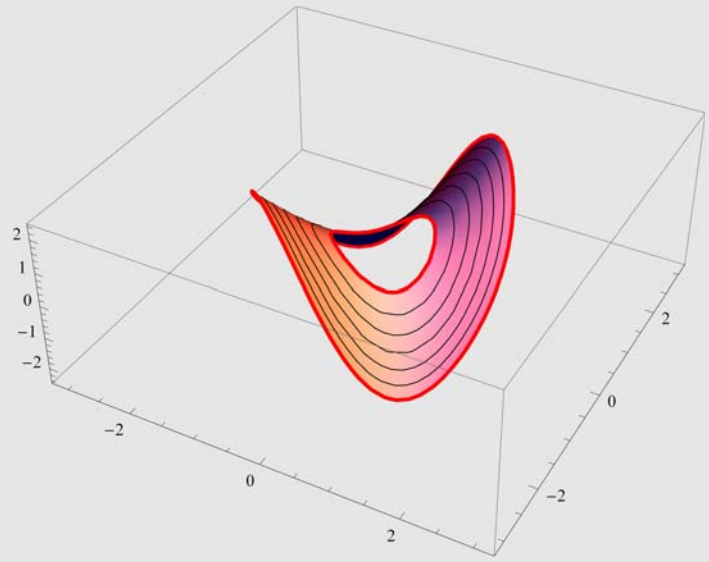
In[45]:=
```
Plot3D[y Sin[x] + y Sin[x], {x, -π, π},
 {y, -π, π}, MeshFunctions → {#3 &}, MeshStyle → {Orange},
 PlotStyle → FaceForm[Directive[Red, Specularity[White, 2]], Blue],
 BoundaryStyle → {Thick, Red}]
```

Out[45]=



In[46]:=
```
Plot3D[y Sin[x] + y Sin[x], {x, -π, π}, {y, -π, π},
 PlotStyle → FaceForm[Directive[Red, Specularity[White, 2]], Blue],
 BoundaryStyle → {Thick, Red}, MeshFunctions → {(#1^2 + #2^2) &},
 Mesh → 5, RegionFunction → Function[{x, y, z}, 1 / 2 < x^2 + y^2 < 3]]
```

Out[46]=



Из других способов рисования отметим, прежде всего, **ParametricPlot3D**, позволяющий рисовать параметрически заданные поверхности и кривые, его частные случаи **RevolutionPlot3D** и **SphericalPlot3D**, неявно заданные поверхности **ContourPlot3D**, трехмерные области заданные неравенствами **RegionPlot3D**, а также трехмерные рисовалки дискретных данных, такие как **ListPlot3D**, **ListPointPlot3D**, **ListSurfacePlot3D**.

Параметризуем тор

In[47]:=
```
R = 3; r = 1;
torus[φ_, ψ_] := {-(R + r Cos[ψ]) Sin[φ], Cos[φ] (R + r Cos[ψ]), r Sin[ψ]};
```

Масштабируем кривизну так, чтобы она принимала значения от 1 до 1/2. Сначала вычислим гауссову кривизну.

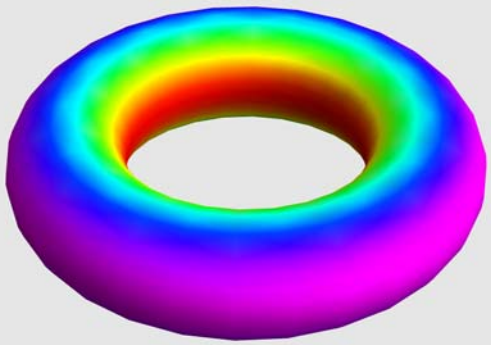
In[49]:=
```
iForm[r_][u_, v_] := Module[{ru, rv},
   ru = D[r[u, v], u]; rv = D[r[u, v], v];
   ( ru.ru  ru.rv
     ru.rv  rv.rv )
  ];
norm[r_][u_, v_] := Module[{ru, rv, nn},
   ru = D[r[u, v], u]; rv = D[r[u, v], v]; nn = Cross[ru, rv];
   nn / Sqrt[nn.nn]
  ];
iiForm[r_][u_, v_] := Module[{ruu, ruv, rvv, n},
   n = norm[r][u, v];
   ruu = D[r[u, v], u, u]; ruv = D[r[u, v], u, v]; rvv = D[r[u, v], v, v];
   ( ruu.n  ruv.n
     ruv.n  rvv.n )
  ];
k[r_][u_, v_] := Det[iiForm[r][u, v]] / Det[iiForm[r][u, v]];
```

In[53]:=
```
g[u_, v_] = k[torus][u, v] // Simplify;
kMax = Maximize[g[u, v], {u, v}][[1]];
kMin = Minimize[g[u, v], {u, v}][[1]];
kRescaled[u_, v_] := (g[u, v] - kMin) / (1.2 (kMax - kMin));
```

Теперь раскрашиваем тор с помощью **ColorFunction**.

In[57]:=
```
ParametricPlot3D[torus[φ, ψ], {φ, 0, 2 π}, {ψ, 0, 2 π}, Boxed → False,
 Axes → False, ColorFunction → Function[{x, y, z, u, v}, Hue[kRescaled[u, v]]] ,
 ColorFunctionScaling → False, Mesh → None]
```

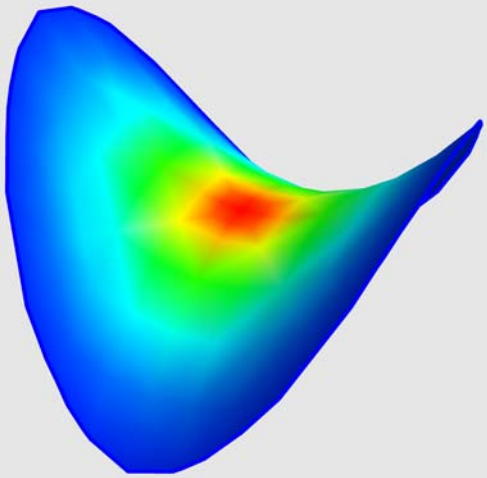
Out[57]=



Параметризуем седловую поверхность

In[58]:=
```
sadd[x_, y_] := {x, y, x^2 - y^2};
kM[x_, y_] := k[sadd][x, y] // Simplify;
kMax = Maximize[kM[u, v], u^2 + v^2 ≤ 2, {u, v}][[1]];
kMin = Minimize[kM[u, v], {u, v}][[1]];
kRes[u_, v_] = (kM[u, v] - kMin) / (1.4 (kMax - kMin));
ParametricPlot3D[sadd[x, y], {x, -2, 2}, {y, -2, 2}, Boxed → False,
 Axes → False, RegionFunction → Function[{x, y, z, u, v}, (x^2 + y^2 < 1 / 2)],
 ColorFunction → Function[{x, y, z, u, v}, Hue[kRes[x, y]]],
 ColorFunctionScaling → False, Mesh → False, BoundaryStyle → {Thick, Blue}]
```

Out[63]=

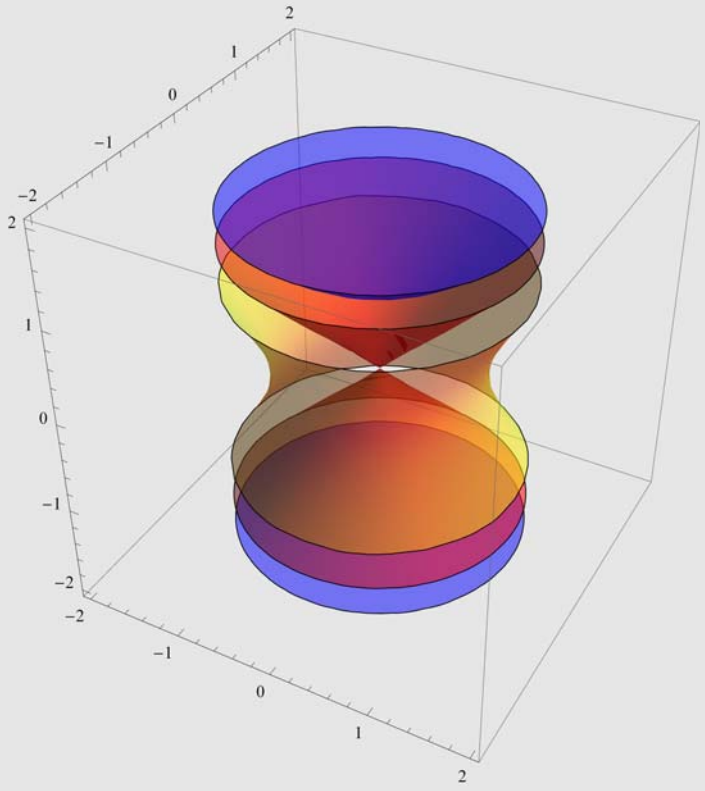


Поверхности уровня

Поверхности уровня удобно рисовать с помощью **ContourPlot3D** (функция, интервалы значения переменных, какие (или сколько именно) значения функции рисовать (**Contours** приписывается количество значений или их список), директивы, описывающие как именно рисовать поверхности (**ContourStyle** приписывается список директив))

In[64]:=
```
ContourPlot3D[x^2 + y^2 - z^2, {x, -2, 2}, {y, -2, 2}, {z, -2, 2}, Contours → {-1, 0, 1},
 ContourStyle → Map[{#, Opacity[0.5]} &, {Blue, Red, Yellow}],
 Mesh → None, RegionFunction → (#1^2 + #2^2 < 2 &)]
```

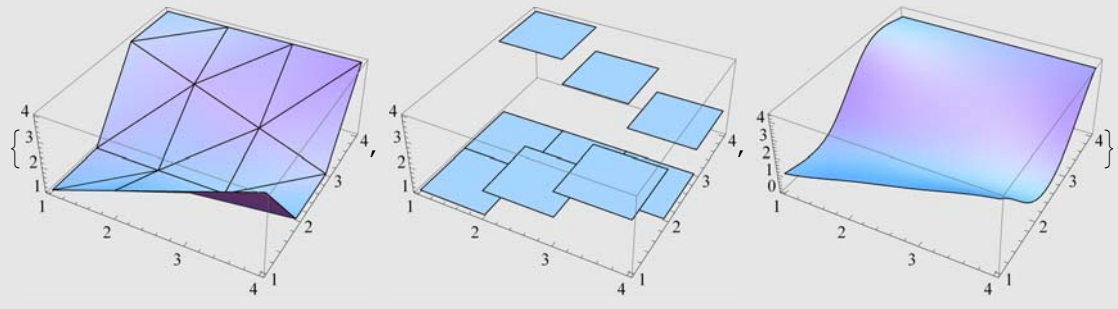
Out[64]=



Следующие команды **ListPlot3D**, **ListPointPlot3D**, **ListSurfacePlot3D** служат для визуализации дискретных наборов чисел.

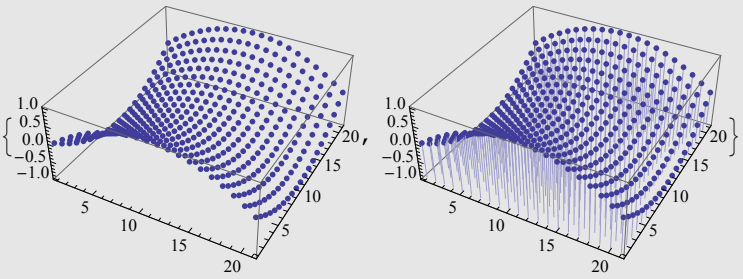
In[65]:=
```
{ListPlot3D[{{1, 2, 3, 4}, {1, 1, 1, 1}, {4, 3, 2, 1}, {4, 4, 4, 4}}, Mesh → All],
 ListPlot3D[{{1, 2, 3, 4}, {1, 1, 1, 1}, {4, 3, 2, 1}, {4, 4, 4, 4}},
  Mesh → None, InterpolationOrder → 0],
 ListPlot3D[{{1, 2, 3, 4}, {1, 1, 1, 1}, {4, 3, 2, 1}, {4, 4, 4, 4}},
  Mesh → None, InterpolationOrder → 3]}
```

Out[65]=



In[66]:=
```
{ListPointPlot3D[Table[i^2 - j^2, {i, -1, 1, 0.1}, {j, -1, 1, 0.1}]],
 ListPointPlot3D[Table[i^2 - j^2, {i, -1, 1, 0.1}, {j, -1, 1, 0.1}], Filling → Bottom]}
```

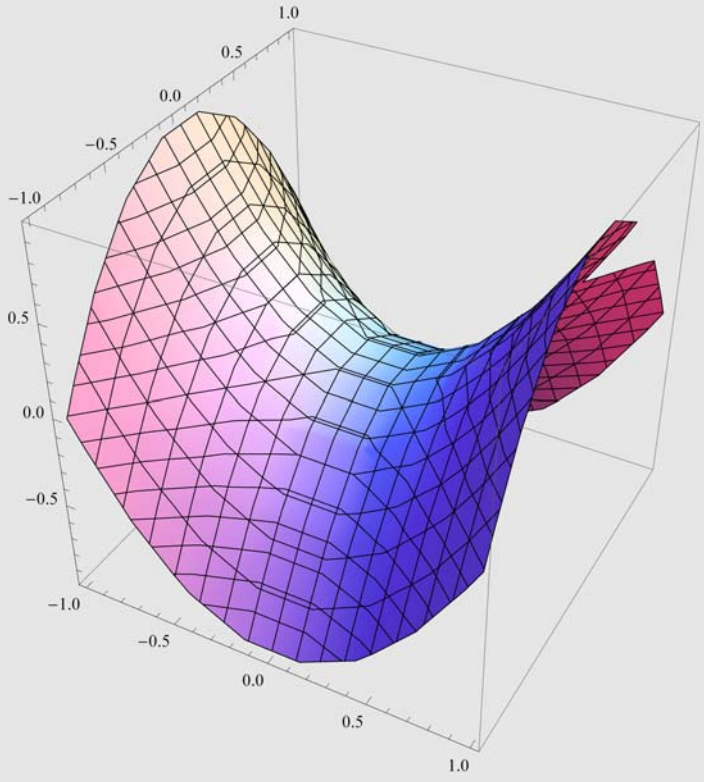
Out[66]=



Наконец, **ListSurfacePlot3D** служит для визуализации набора точек в пространстве.

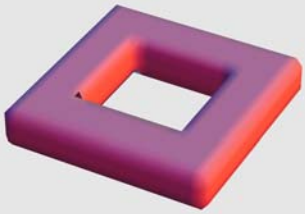
In[67]:=
```
ListSurfacePlot3D[Flatten[Table[{i, j, i^2 - j^2}, {i, -1, 1, 0.1}, {j, -1, 1, 0.1}], 1]]
```

Out[67]=



In[68]:=
```
RegionPlot3D[-1 ≤ x ≤ 1 && -1 ≤ y ≤ 1 && -0.2 ≤ z ≤ 0.2 &&
  Not[-1 / 2 ≤ x ≤ 1 / 2 && -1 / 2 ≤ y ≤ 1 / 2], {x, -2, 2}, {y, -2, 2}, {z, -2, 2},
 Mesh → None, PlotPoints → 50, Axes → None, Boxed → False, PlotStyle → {Pink}]
```

Out[68]=



Первый способ

Примитивы

- **Введение**

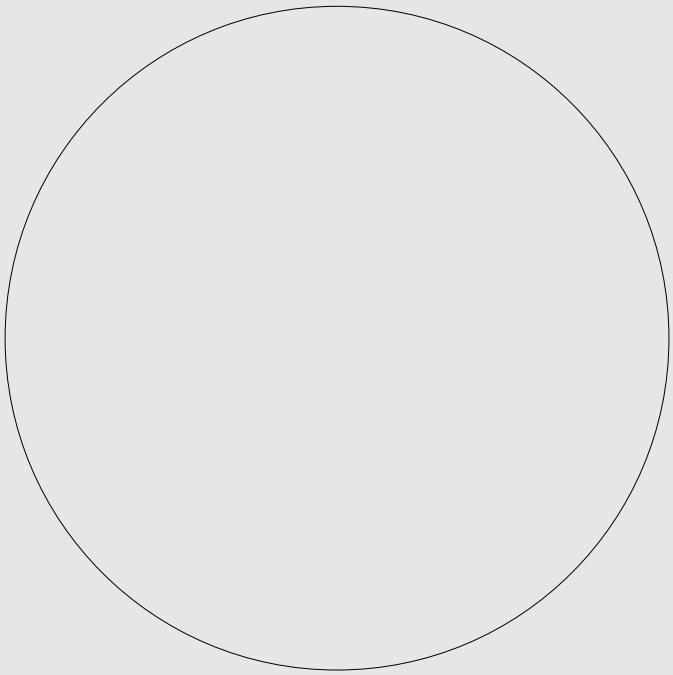
  Общий формат команды, изображающей на экране графические объекты, составленные из примитивов таков:

  Graphics[< primitives >, < options >] (* для двумерной графики*)
  Graphics3D[< primitives >, < options >](* для трехмерной графики*)

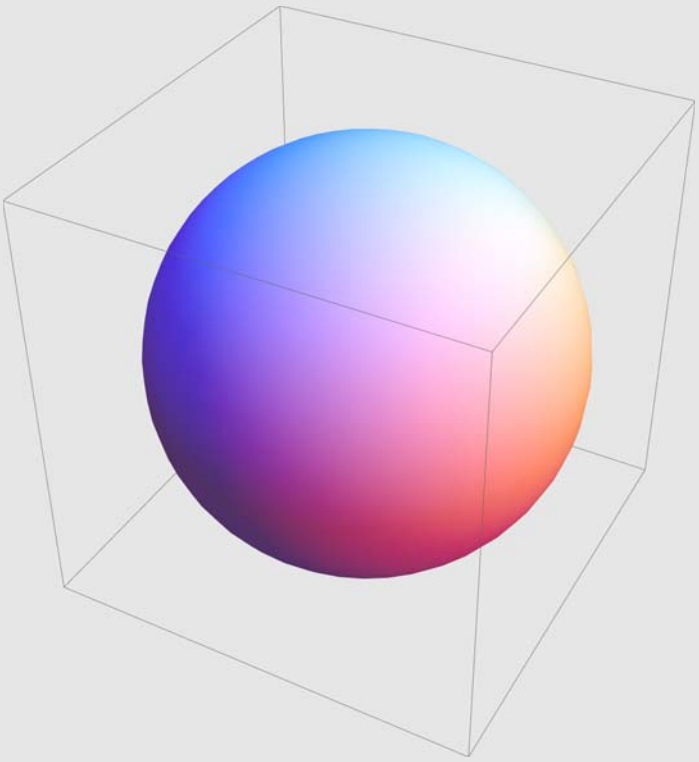
  Например,

In[69]:=
```
Graphics[Circle[]]
Graphics3D[Sphere[]]
g = Graphics[Disk[]];
```

Out[69]=



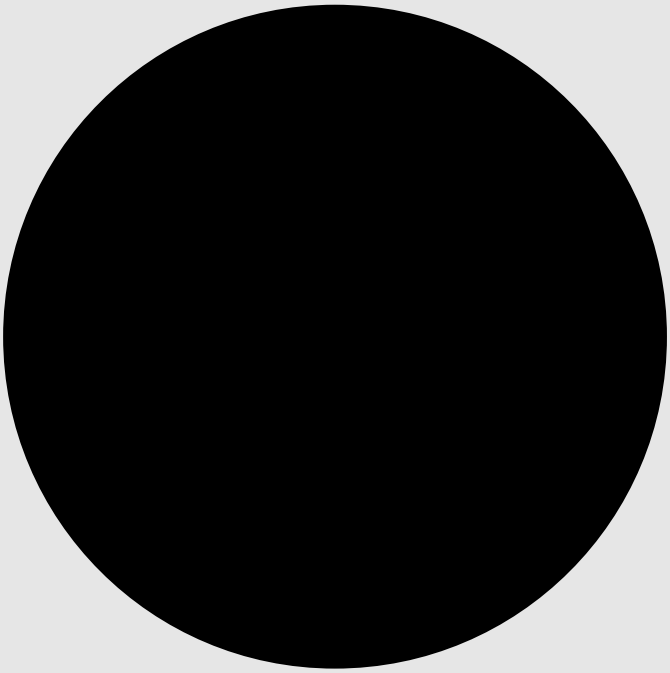
Out[70]=



Отметим, что точка с запятой после команды, как и положено, подавляет вывод на экран, хотя оерация все равно выполняется и ее результат сохраняется. Увидеть полученное можно, обратившись к результату операции:

In[72]:=
```
g
```

Out[72]=

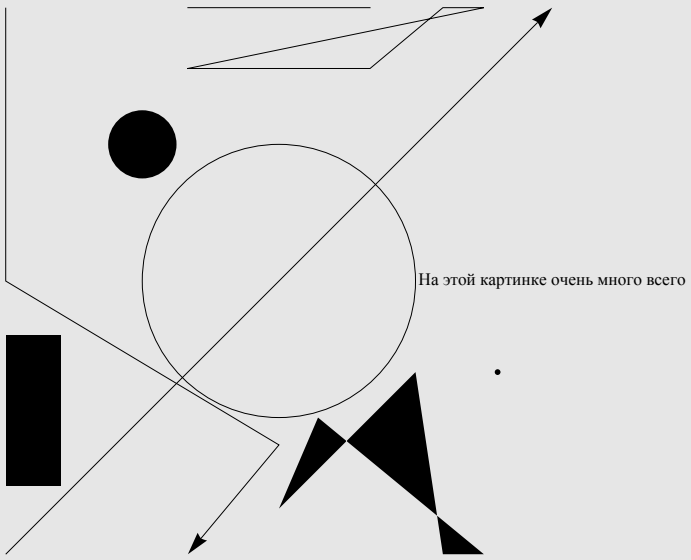


Вот список основных **двумерных графических примитивов** (элементарных кирпичиков, из которых складывается любая картинка) в алфавитном порядке (напомним, что комментарии обрамляются скобками со звездочками: **(* комметарии*)**)

In[73]:=

```
Graphics[{Arrow[{{0, 0}, {1, 1}}],
          Arrow[{{0, 1}, {0, 1 / 2}, {1 / 2, 1 / 5}, {1 / 3, 0}}],
  (* Стрелка, прямая или ломаная,
  заданная набором своих последовательных вершин (точнее - их координат) *)
          Circle[{1 / 2, 1 / 2}, 1 / 4], (*окружность,
  заданная своим центром и радиусом *)
          Disk[{1 / 4, 3 / 4}, 1 / 16], (* заполненный диск,
  заданный своим центром и радиусом *)
          Line[{{1 / 3, 1}, {2 / 3, 1}}],
          Line[{{1 / 3, 8 / 9}, {2 / 3, 8 / 9}, {4 / 5, 1}, {7 / 8, 1}, {1 / 3, 8 / 9}}],
  (* Отрезок или ломаная линия,
  заданные координатами своих последовательных вершин, Listable *)
          Point[{0.9, 1 / 3}], (* точка, заданная своими координатами, Listable *)
          Polygon[{{1 / 2, 1 / 12}, {3 / 4, 1 / 3}, {4 / 5, 0}, {7 / 8, 0}, {4 / 7, 1 / 4}}],
  (*заполненный многоугольник, заданный координатами последовательных вершин*)
         Rectangle[{0, 1 / 8}, {1 / 10, 4 / 10}], (* прямоугольник,
  заданный левой нижней и правой верхней вершинами*)
          Text["На этой картинке очень много всего", {1, 1 / 2}]
  (* текст, центровнный в заданной точке *)
 }]
```

Out[73]=

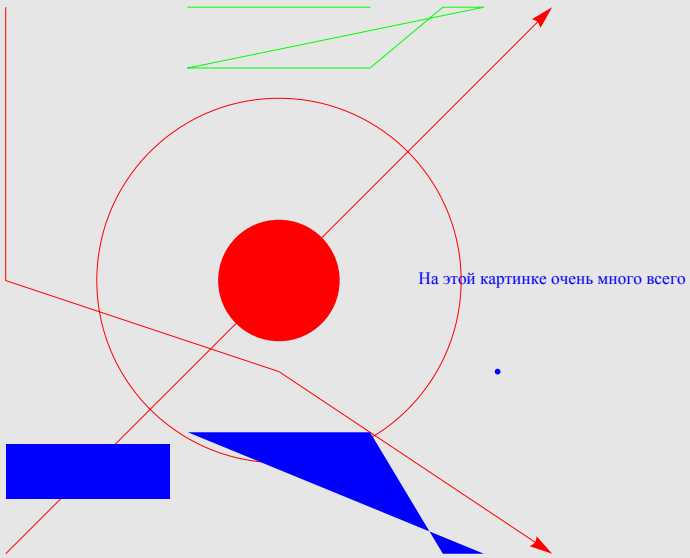


Если на этой картинке элементы будут перекрываться, то, конечно, она может оказаться не очень простой для восприятия. Научимся использовать цветы и менять толщину линий. Имеется две возможности менять вид рисунка вообще и графических примитивов в частности. Первый способ связан с использованием **опций** (**Options**), которые влияют на рисунок в целом. Мы поговорим о них потом, а сейчас займемся вторым способом, использующим так называемые графические **директивы** (**Directives**). Каждая директива оказывает действие на следующие за ней примитивы в списке.

In[74]:=

```
Graphics[{Red, Arrow[{{0, 0}, {1, 1}}],
  (* Red, очевидно, делает последующий вывод красным *)
         Arrow[{{0, 1}, {0, 1 / 2}, {1 / 2, 1 / 3}, {1, 0}}],
         Circle[{1 / 2, 1 / 2}, 1 / 3],
         Disk[{1 / 2, 1 / 2}, 1 / 9],
         Green, Line[{{1 / 3, 1}, {2 / 3, 1}}], (*Green - зелёным*)
         Line[{{1 / 3, 8 / 9}, {2 / 3, 8 / 9}, {4 / 5, 1}, {7 / 8, 1}, {1 / 3, 8 / 9}}],
         Blue, Point[{0.9, 1 / 3}],
         Polygon[{{1 / 3, 2 / 9}, {2 / 3, 2 / 9}, {4 / 5, 0}, {7 / 8, 0}, {1 / 3, 2 / 9}}],
        Rectangle[{0, 1 / 10}, {3 / 10, 2 / 10}],
         Text["На этой картинке очень много всего", {1, 1 / 2}]
 }]
```

Out[74]=



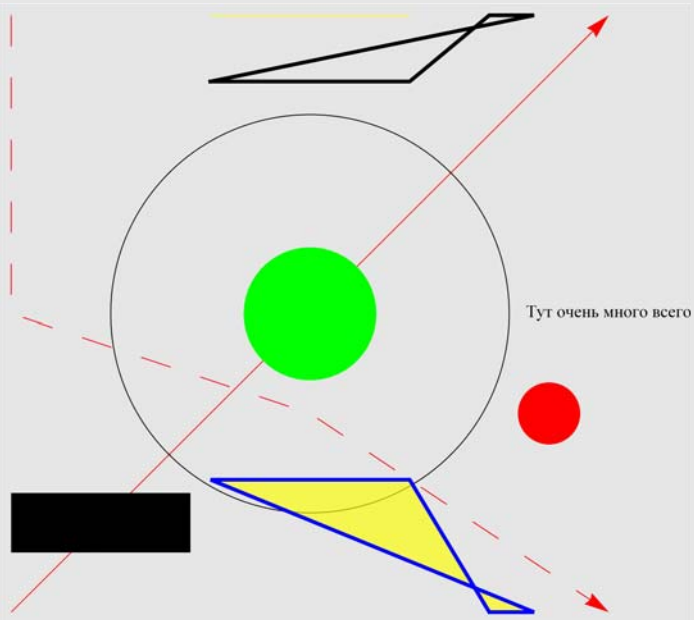
Можно ограничивать действие директивы, организуя примитивы в несколько подсписков.
Обратите внимание на "многоугольник" - у него теперь граница отличается цветом от внутренности, кроме того, внутренность полупрозрачна.

In[75]:=

```
Graphics[{
  {Red, Arrow[{{0, 0}, {1, 1}}], Dashing[0.05],
   Arrow[{{0, 1}, {0, 1 / 2}, {1 / 2, 1 / 3}, {1, 0}}]},
          Circle[{1 / 2, 1 / 2}, 1 / 3],
          {Green, Disk[{1 / 2, 1 / 2}, 1 / 9]},
          {Yellow, Line[{{1 / 3, 1}, {2 / 3, 1}}]},

  {Thick, Line[{{1 / 3, 8 / 9}, {2 / 3, 8 / 9}, {4 / 5, 1}, {7 / 8, 1}, {1 / 3, 8 / 9}}]},
  (*Thick указывает на толщину линии*)
          {PointSize[0.1], Red, Point[{0.9, 1 / 3}]},
  (* PointSize на относительный размер точки *)
          {Opacity[0.65], EdgeForm[{Blue, Thick}], FaceForm[Yellow],
   (* Прозрачность, св-ва граничной ломаной и заполнения*)
           Polygon[{{1 / 3, 2 / 9}, {2 / 3, 2 / 9}, {4 / 5, 0}, {7 / 8, 0}, {1 / 3, 2 / 9}}]},
           Rectangle[{0, 1 / 10}, {3 / 10, 2 / 10}],
           {Text["Тут очень много всего", {1, 1 / 2}] }
 }]
```

Out[75]=

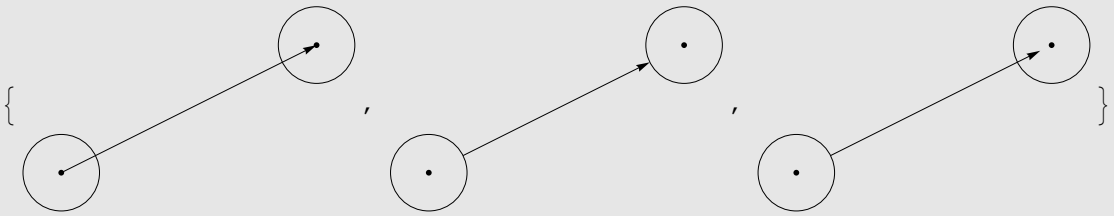


Более подробное описание основных примитивов

1. Задавая стрелку **Arrow** можно, кроме концов, указывать отступы от них (**второй аргумент**), которые могут быть как одинаковыми, так и разными (окружности нарисованы для наглядности).

In[76]:=
```
{Graphics[{Arrow[{{0, 0}, {2, 1}}],
   Circle[{0, 0}, 0.3], Circle[{2, 1}, 0.3], Point[{{0, 0}, {2, 1}}]}],
 Graphics[{Arrow[{{0, 0}, {2, 1}}, .3], Circle[{0, 0}, 0.3],
   Circle[{2, 1}, 0.3], Point[{{0, 0}, {2, 1}}]}],
 Graphics[{Arrow[{{0, 0}, {2, 1}}, {.3, .1}], Circle[{0, 0}, 0.3],
   Circle[{2, 1}, .3], Point[{{0, 0}, {2, 1}}]}]}
```

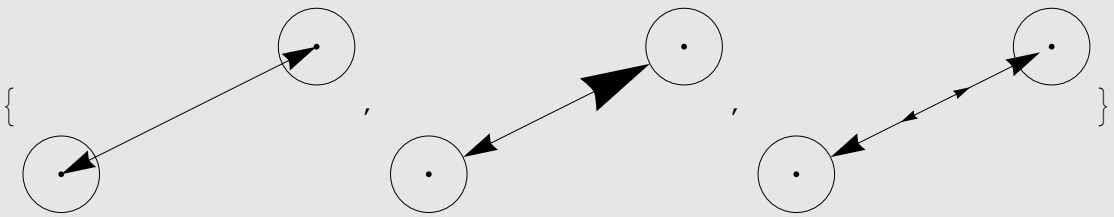
Out[76]=



Можно также управлять размером и расположением стрелочек, используя директиву **Arrowheads.** Элементы списка задают направление (знак) и относительный размер стрелки (число). Абсолютный размер можно задавать командами **Tiny, Small, Medium, Large.**

In[77]:=
```
{Graphics[{Arrowheads[{-.1, .1}], Arrow[{{0, 0}, {2, 1}}],
   Circle[{0, 0}, 0.3], Circle[{2, 1}, 0.3], Point[{{0, 0}, {2, 1}}]}],
 Graphics[{Arrowheads[{-.1, .2}], Arrow[{{0, 0}, {2, 1}}, .3],
   Circle[{0, 0}, 0.3], Circle[{2, 1}, 0.3], Point[{{0, 0}, {2, 1}}]}],
 Graphics[{Arrowheads[{-0.1, -.05, .05, .1}], Arrow[{{0, 0}, {2, 1}}, {.3, .1}],
   Circle[{0, 0}, 0.3], Circle[{2, 1}, .3], Point[{{0, 0}, {2, 1}}]}]}
```

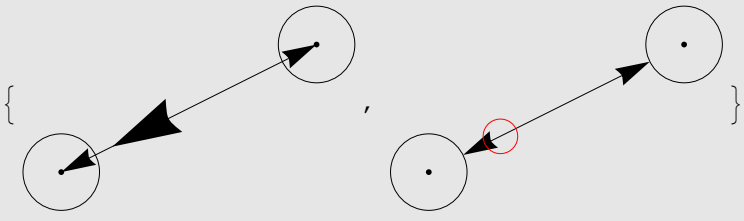
Out[77]=



Элемент списка директивы Arrowheads сам может быть списком, второй элемент которого отвечает за относительное положение стрелки (число от нуля до единицы), а третий - за форму (Graphics).

In[78]:=
```
{Graphics[{Arrowheads[{{-.1, 0}, {-.2, .2}, .1}], Arrow[{{0, 0}, {2, 1}}],
   Circle[{0, 0}, 0.3], Circle[{2, 1}, 0.3], Point[{{0, 0}, {2, 1}}]}],
 Graphics[{Arrowheads[{{-.1, 0}, {-.05, .2, Graphics[{Red, Circle[]}]}, .1}],
   Arrow[{{0, 0}, {2, 1}}, .3], Circle[{0, 0}, 0.3],
   Circle[{2, 1}, 0.3], Point[{{0, 0}, {2, 1}}]}]}
```

Out[78]=

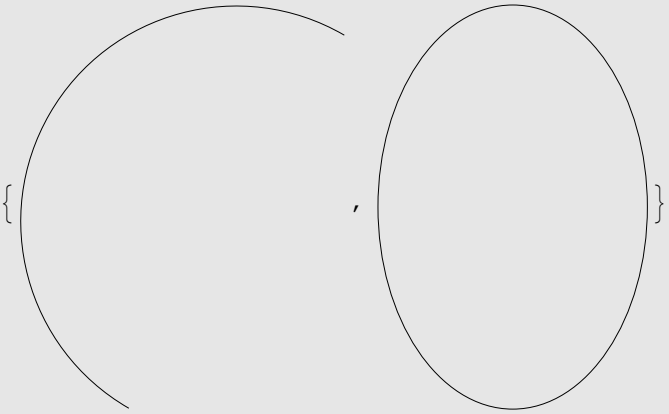


2. Примитив **Circle** может быть использован также для рисования дуг (третий аргумент задает начало и конец дуги в радианах) и эллипсов (второй аргумент в этом случае не радиус, а список

длин полуосей). Примитив **Cycle[]** без аргументов дает единичную окружность с ценром в начале координат.

In[79]:=
```
{Graphics[Circle[{0, 0}, 1, {π/3, 4π/3}]], Graphics[Circle[{0, 0}, {2, 3}]]}
```

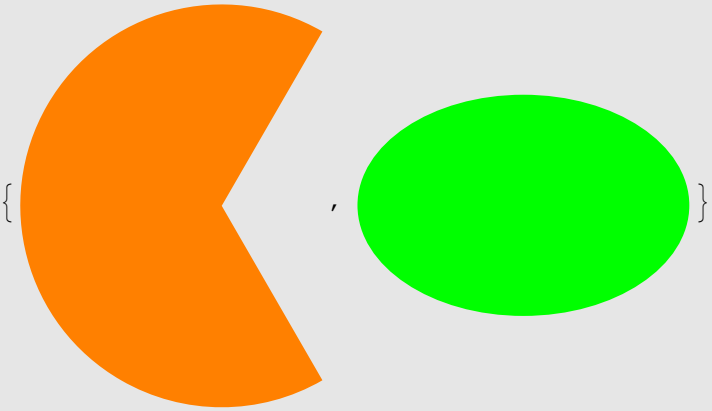
Out[79]=



3. Аналогичные модификации имеются у примитива **Disk**

In[80]:=
```
{Graphics[{Orange, Disk[{0, 0}, 1, {π / 3, 5 π / 3}]}],
 Graphics[{Green, Disk[{0, 0}, {3, 2}]}]}
```

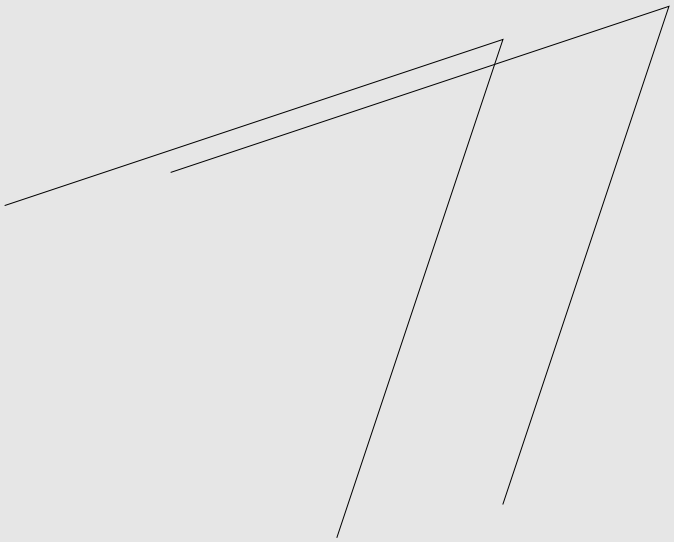
Out[80]=



4. Примитив **Line** позволяет рисовать сразу несколько ломаных. Вот, скажем, две ломаных, отличающиеся на параллельный перенос на вектор **a**

In[81]:=
```
ll = {{1, 0}, {2, 3}, {-1, 2}};
a = {1, .2};
Graphics[Line[{ll, Map[a + # &, ll]}]]
```

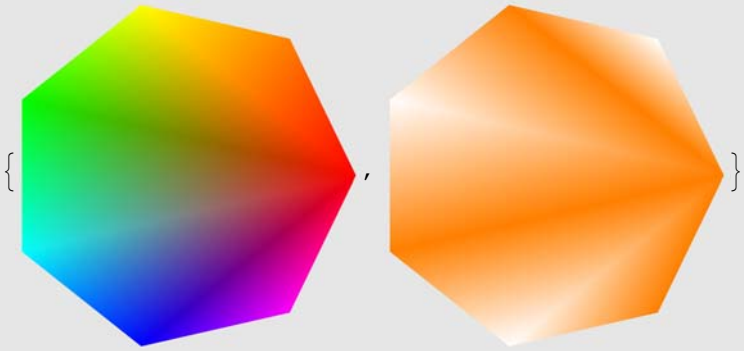
Out[83]=



5. Точно так же **Polygon** может рисовать сразу несколько многоугольников. Кроме того, цвет многоугольника можно задавать "от вершин, по градиенту". Для этого используется опция **VertexColor**

In[84]:=
```
{Graphics[Polygon[Table[{Cos[i 2 π/7], Sin[i 2 π/7]}, {i, 0, 6}],
    VertexColors → {Red, Orange, Yellow, Green, Cyan, Blue, Magenta}]],
 Graphics[Polygon[Table[{Cos[i 2 π/7], Sin[i 2 π/7]}, {i, 0, 6}],
    VertexColors → {Orange, White, Orange, White, Orange, White, Orange}]]}
```

Out[84]=

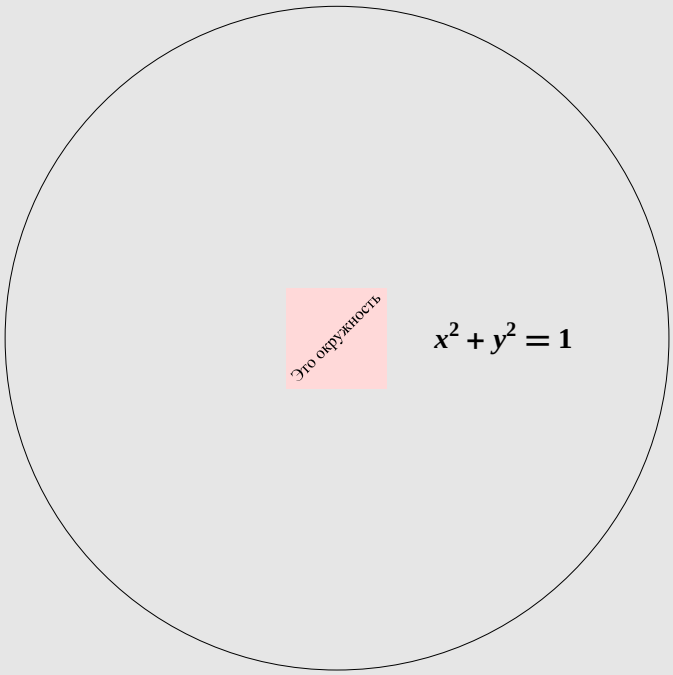


6. Если примитив **Rectangle** использовать с одним аргументом, то получится единичный квадрат с заданным левым нижним углом. Если вовсе без аргументов - то с левым нижним углом в начале координат.

7. Наконец, примитив **Text** располагает средствами для позиционирования текста (второй аргумент), выбора отступа (третий аргумент), направления (четвертый), фона (пятый) и т.п. Мы не будем на этом останавливаться подробно, приведем только один пример.

In[85]:=
```
Graphics[
 {Circle[], Text["Это окружность", {0, 0}, Automatic, {1, 1}, Background → LightRed],
  Text[Style[x^2 + y^2 == 1, 15, Bold], {1 / 2, 0}, FormatType → TraditionalForm]}]
```

Out[85]=



Прежде чем перейти к полному списку графических директив, перечислим оставшиеся примитивы. Это

8. Inset: служащий для вставки одного объекта внутрь другого, например, подписи к рисункам;

9. Raster: изображающий прямоугольник, разбитый на раскрашеные квадратики заданных цветов;

10. Locator: служит для создания динамического элемента графики, позволяющего вводить координаты текущей точки экрана;

11. GraphicsGroup: служит для объединения объектов в группу, которая может быть отредоктирована как единое целое;

12. GraphicsComplex: важный элемент графики, позволяющий отдельно задавать структуру одномерного, двумерного или трехмерного комплекса (набора примитивов), и отдельно задавать координаты определяющих их точек. Формат: **GraphicsComplex[координаты вершин, структура]**. При этом в структуре вершины задаются уже только своими номерами в списке вершин. Приведем пример. Здесь список вершин состоит из 6 элементов (первые три - вершины треугольника, оставшиеся три - середины его сторон).

Примеры

Теорема о медианах треугольника

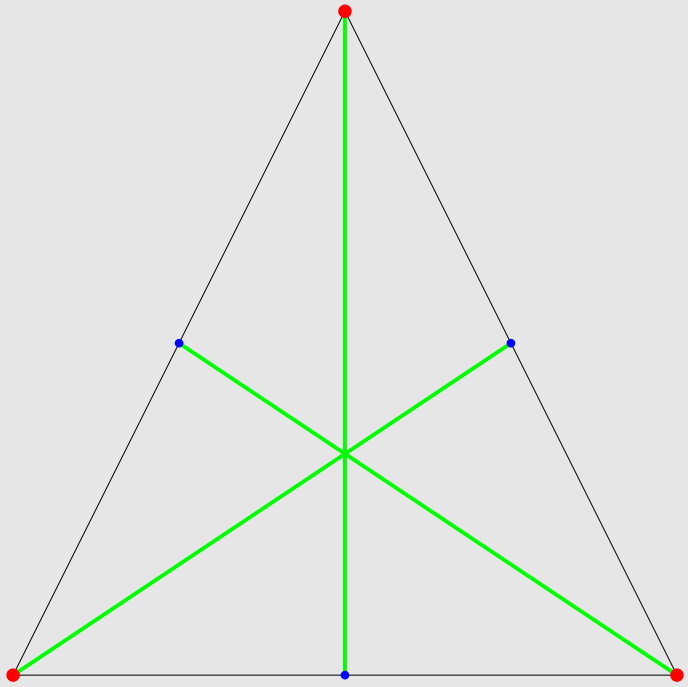
GraphicComlex - очень удобная команда.

В следующем примере мы создали список из трех вершин треугольника, добавили к нему середина сторон, потом с помощью GraphicComplex указали какие точки следует соединить и заодно указали цвета для линий: стороны треугольника показаны тонкими черными линиями, а медианы - толстыми зелеными.

In[86]:=

```
a = {-1, -1}; b = {1, -1}; c = {0, 1};
v = {a, b, c} ~Join~ {(b + c)/2, (a + c)/2, (b + a)/2};
Graphics[GraphicsComplex[v,
  {Thin, Line[{1, 2, 3, 1}], Green, Thick, Line[{1, 4}], Line[{2, 5}], Line[{3, 6}], Red,
   PointSize[Large], Point[{1, 2, 3}], Blue, PointSize[Medium], Point[{4, 5, 6}]}]]
```

Out[88]=



У нарисованного треугольника медианы, судя по всему, персекаются в одной точке. Конечно же, мы знаем теорему о том, что медианы пересекаются в одной точке для любого треугольника. Но представим на мгновение, что эта теорема нам неизвестна. А нам, прежде чем пробовать искать ее доказательство, естественно, хочется проверить ее на примерах.
Замечательный способ сделать это дают динамические возможности *Mathematica.*
В следующем примере мы получим возможность передвигать вершины треугольника с помощью мышки. Все что нужно сделать - это вставить код предыдущего примера
в команду **Manipulate**

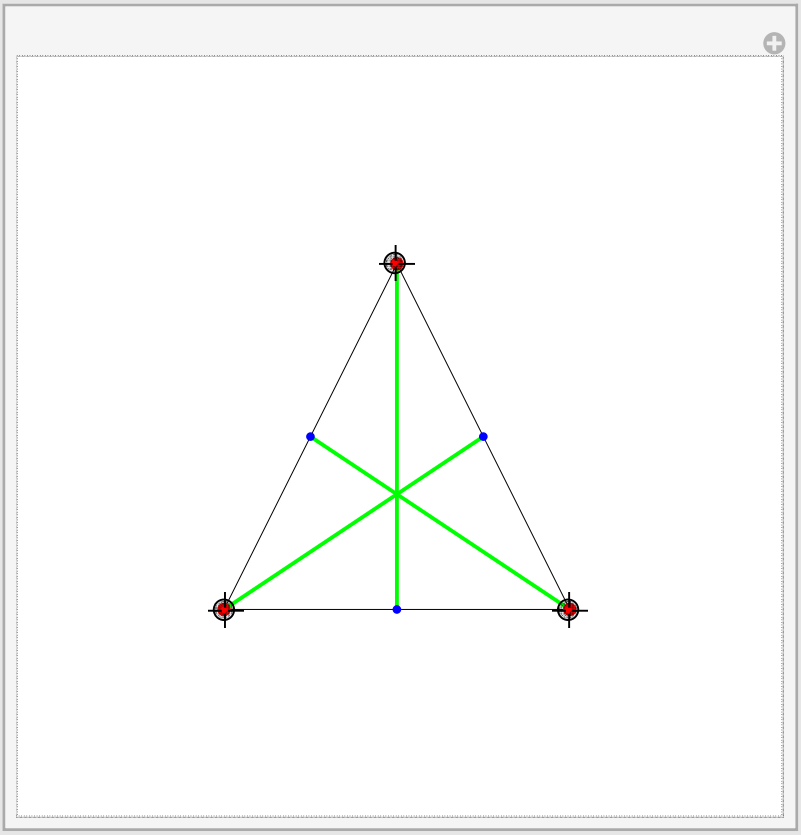
```
Manipulate[  ЗДЕСЬ КОД ДЛЯ РИСОВАНИЯ ТРЕУГОЛЬНИКА И МЕДИАН   ,
 {{a, {{-1, -1}, {1, -1}, {0, 1}}}, Locator}]
```

Второй аргумент в этой команде - указание, что координаты трех точек из списка **a** задаются с помощью примитива **Locator**, дающего возможность вводить положение точки мышью).

In[89]:=

```
Manipulate[
 Graphics[GraphicsComplex[a ~Join~ {(a[[2]] + a[[3]])/2, (a[[1]] + a[[3]])/2, (a[[2]] + a[[1]])/2}, {Thin,
    Line[{1, 2, 3, 1}], Green, Thick, Line[{1, 4}], Line[{2, 5}], Line[{3, 6}], Red,
    PointSize[Large], Point[{1, 2, 3}], Blue, PointSize[Medium], Point[{4, 5, 6}]}],
  PlotRange → 2.0], {{a, {{-1, -1}, {1, -1}, {0, 1}}}, Locator}]
```

Out[89]=



Поставив этот математический эксперимент, мы убеждаемся, что медианы треугольника должны пересекаться в одной точке. Но не забудем, что эксперимент, хоть и математический - еще не доказательство.

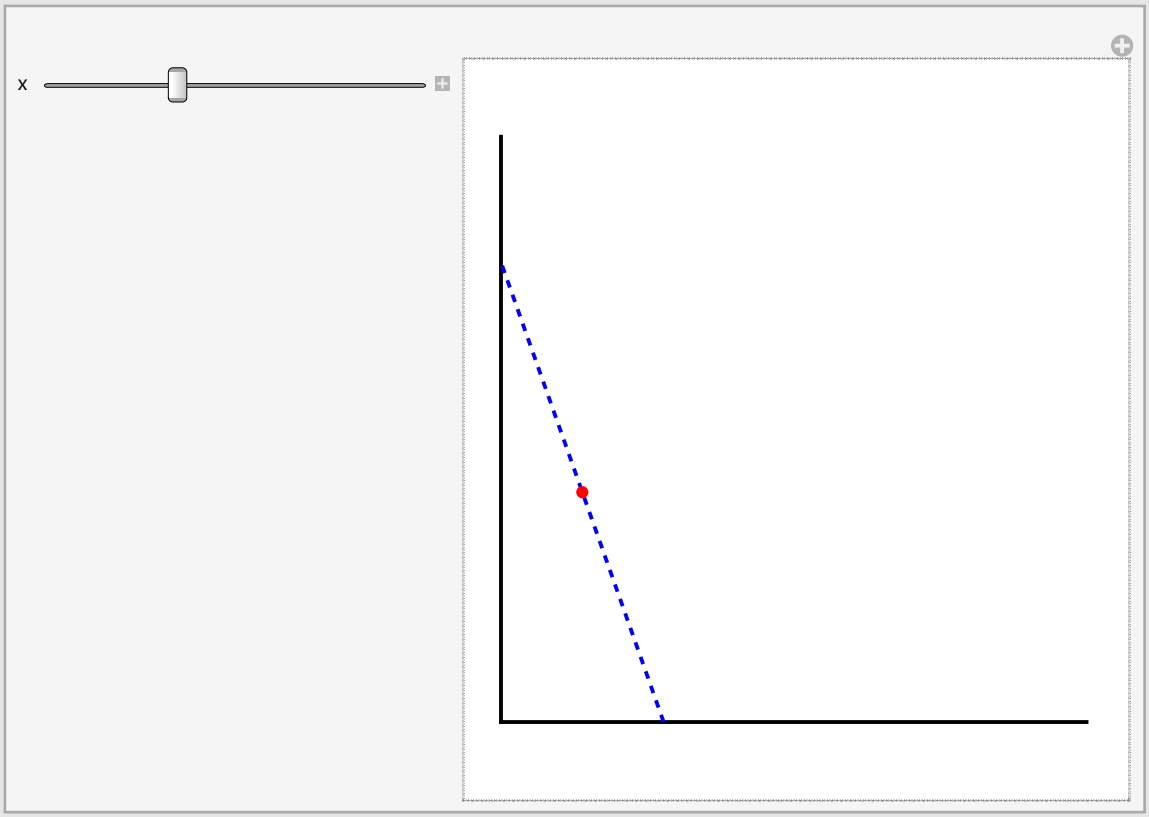
Кошка не лестнице

ЗАДАЧА: На середине прислоненной к стене лестницы сидит кошка. Лестница начала скользить вниз, оставаясь прислоненной к стене. По какой кривой перемещается кошка?

In[90]:=

```
Manipulate[
 Graphics[{
   {Thick, Black , Line[{{0, 0}, {1.2, 0}}], Line[{{0, 0}, {0, 1.2}}]}, (*пол и стена*)
   {Blue, Thick, Dashed, Line[{{x, 0}, {0, Sqrt[1 - x^2]}}]} , (*лестниц*)
   {Red, PointSize[0.02], Point[{x / 2, Sqrt[1 - x^2] / 2}]}(*кошка*)}
 ] , {{x, 1 / 3}, 0, 1} (*координата нижнего конца лестницы*)]
```

Out[90]=

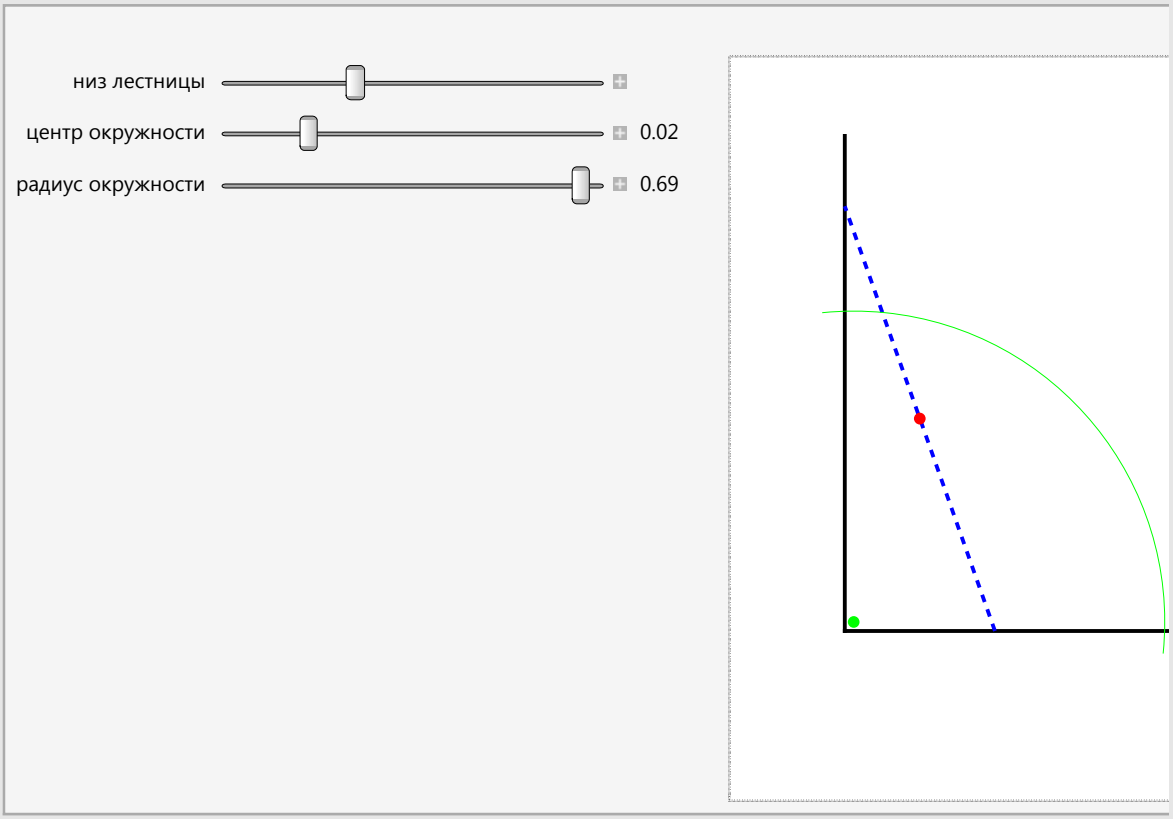


Похоже на выпуклую кривую, т.е. это не прямая, не гипербола. Она симметрична относительно биссектрисы x = y, поэтому эллипсом с РАЗНЫМИ полуосями она быть не может. Может быть это окружность? По соображениям симметрии у нее центр должен лежать на биссектрисе x = y. Добавим в наш код окружность с центром в переменной точке (t,t), радиусом r, причем параметры t и r мы можем изменять. Затем поэкспериментируем.

In[91]:=

```
Manipulate[
 Graphics[{
   {Thick, Black , Line[{{0, 0}, {1.2, 0}}], Line[{{0, 0}, {0, 1.2}}]}, (*пол и стена*)
   {Blue, Thick, Dashed, Line[{{x, 0}, {0, Sqrt[1 - x^2]}}]} , (*лестниц*)
   {Red, PointSize[0.02], Point[{x / 2, Sqrt[1 - x^2] / 2}]}(*кошка*),
   {Green, PointSize[0.02], Point[{t, t}]}(*центр окружности*),
   {Green, Thin, Circle[{t, t}, r, {-0.1, Pi / 2 + 0.1}]} (*окружность*)
  },
  PlotRange → {{-0.2, 1.1}, {-0.2, 1.1}}
 ] ,
 {{x, 1 / 3, "низ лестницы"}, 0, 1} (*координата нижнего конца лестницы*),
 {{t, 0.02, "центр окружности"}, -0.1, 0.5, Appearance → "Labeled"},
 {{r, 0.69, "радиус окружности"}, 0, 0.7, Appearance → "Labeled"}
]
```

Out[91]=

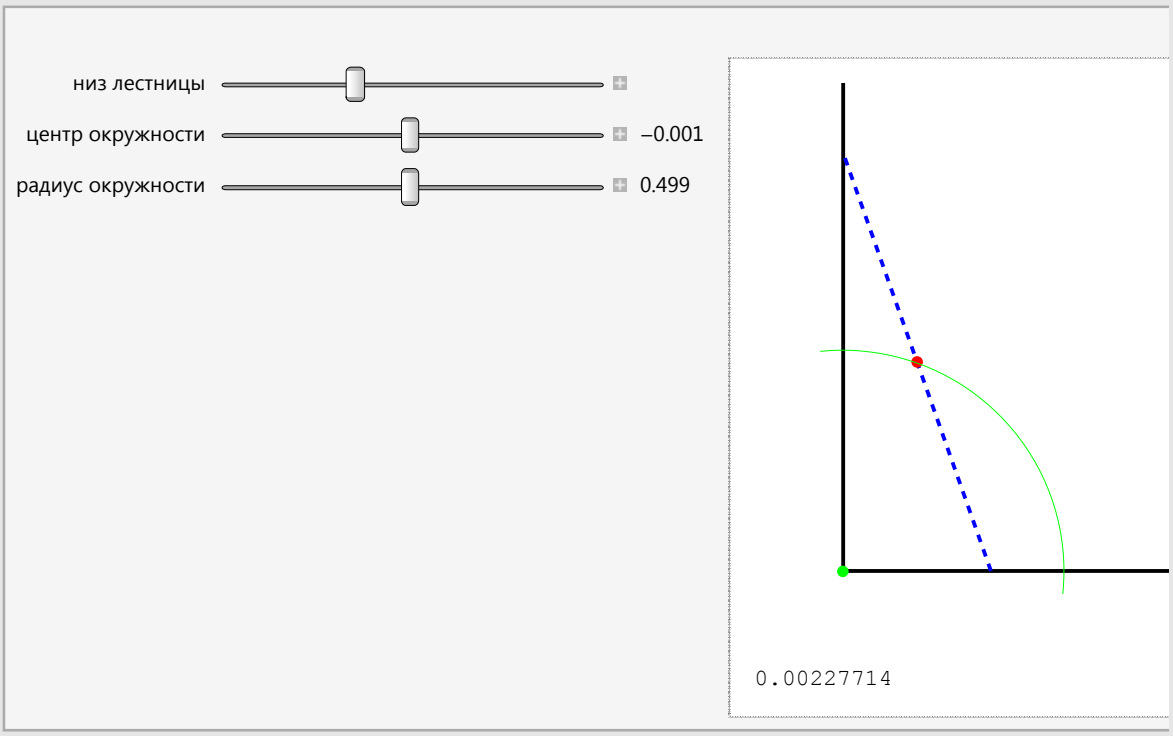


Обнаружив, что эта кривая похожа на окружность с центром в начале координат и радиусом 1/2, добавим к нашему графическому математическому эксперименту вычисления.
А именно, будем подставлять координаты красной точки = кошки в уравнение зеленой прямой.
Но на этот раз сильно ограничим области изменения координат центра окружности и ее радиуса.
Обратите внимание, что координаты центра и радиус теперь меняются дискретно - с шагом 0.01.
Заметьте, как используется новая команда Column для вывода картинки и числа в столбик.
У нее есть аналог - Row, предназначенный для вывода результатов рядом друг с другом в строку.

In[92]:=

```
Manipulate[  Column[{
    Graphics[{
      {Thick, Black , Line[{{0, 0}, {1.2, 0}}],
       Line[{{0, 0}, {0, 1.2}}]}, (*пол и стена*)
      {Blue, Thick, Dashed, Line[{{x, 0}, {0, Sqrt[1 - x^2]}}]} , (*лестниц*)
      {Red, PointSize[0.02], Point[{x / 2, Sqrt[1 - x^2] / 2}]}(*кошка*),
      {Green, PointSize[0.02], Point[{t, t}]}(*центр окружности*),
      {Green, Thin, Circle[{t, t}, r, {-0.1, Pi / 2 + 0.1}]} (*окружность*)
     },
     PlotRange → {{-0.2, 1.1}, {-0.2, 1.1}}, ImageSize → 300
    ] ,
    N[(x / 2 - t) ^2 + (Sqrt[1 - x^2] / 2 - t) ^2 - r^2, 10]
   }],
 {{x, 1 / 3, "низ лестницы"}, 0, 1} (*координата нижнего конца лестницы*),
 {{t, -0.001, "центр окружности"}, -0.1, 0.1, 0.01, Appearance → "Labeled"},
 {{r, 0.499, "радиус окружности"}, 0.4, 0.6, 0.01, Appearance → "Labeled"}
]
```

Out[92]=



Наблюдая числа равные 0 или числа вида 10^{-17} (машинные нули), мы теперь понимаем, что же нужно доказывать: а именно, что кошка перемещается по окружности с центром в начале координат и радиусом равным половине лестницы.

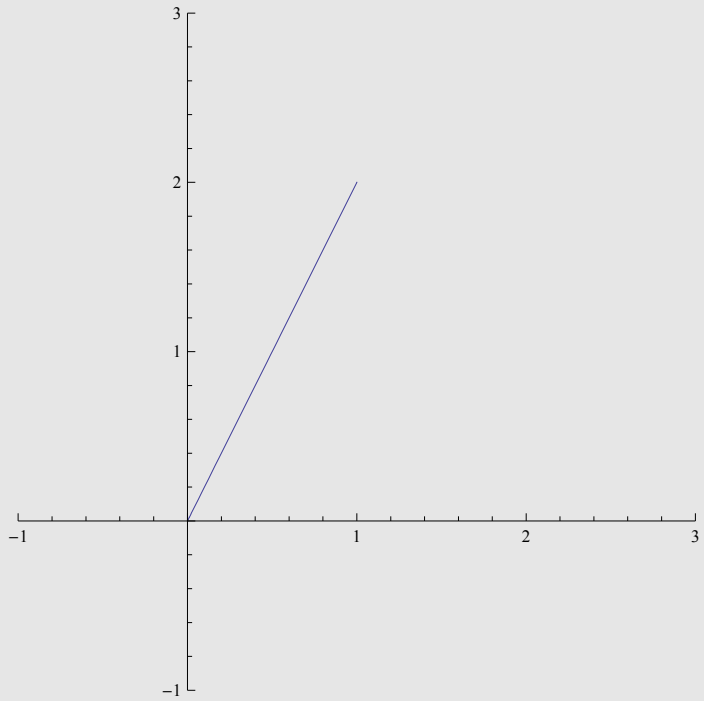
Разумеется, этот пример игрушечный, потому что всего - навсего надо заметить или вспомнить, что медиана , проведенная из вершины прямого угла прямоугольного треугольника равна половине гипотенузы.

Теорема о высотах треугольника

Для начала научимся проводить через данную точку прямую, перпендикулярную данной.

In[93]:=
```
a = {{0, 0}, {1, 2}, {0, 2}};
ParametricPlot[a[[1]] t + a[[2]] (1 - t), {t, 0, 1}, PlotRange → {{-1, 3}, {-1, 3}}]
```

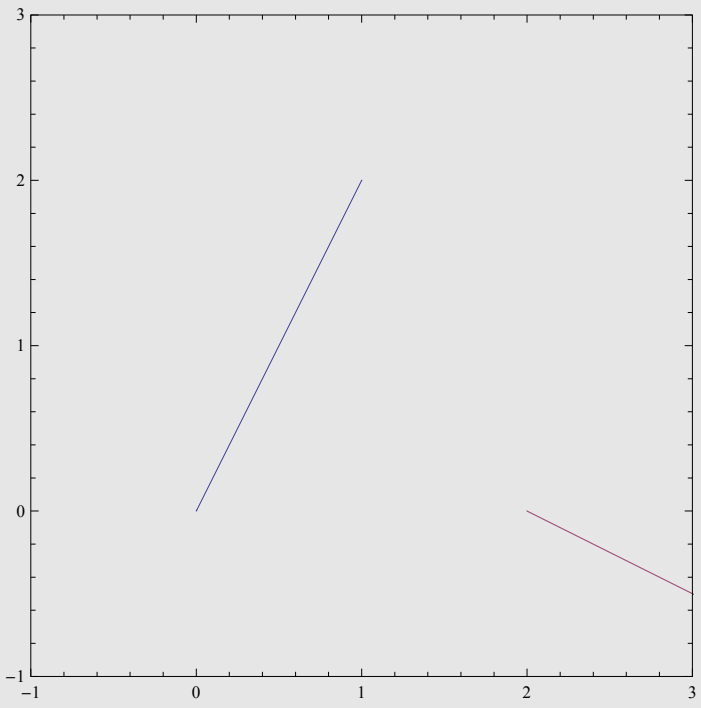
Out[94]=



Оси нам мешают - уберем их. Для удобства добавим рамку вокруг рисунка.
Рисуем прямую с направляющим вектором, ортогональным прямой, соединяющей первые две точки, и проходящую через третью.

In[95]:=
```
a = {{0, 0}, {1, 2}, {2, 0}};
ParametricPlot[
 {a[[1]] t + a[[2]] (1 - t), (*сторона треугольника*)
  a[[3]] + t ({-(a[[1]] - a[[2]])[[2]], (a[[1]] - a[[2]])[[1]] })
  (*ортогональная прямая. разберитесь,
  как реализовано ортогональное направление*)
 },
 {t, 0, 1}, PlotRange → {{-1, 3}, {-1, 3}}, Axes → False, Frame → True]
```

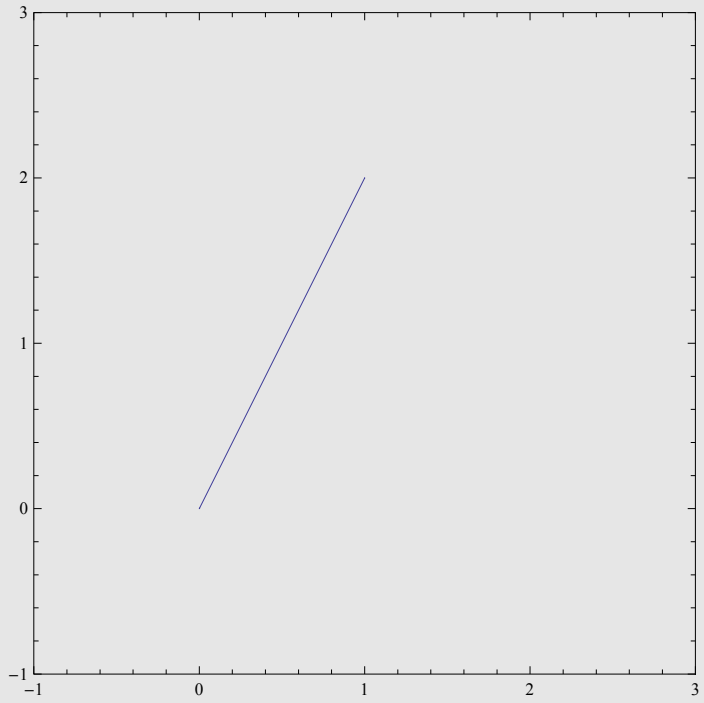
Out[96]=



Мы нарисовали часть перпендикулярной к стороне прямой, и выбрали ее неудачно. Чтобы не возиться с тем, что тут происходит, нарисуем всю прямую, благодаря фиксированной области, которая показана на рисунке, увидим только то, что попало в PlotRange

In[97]:=
```
a = {{0, 0}, {1, 2}, {2, 0}};
ParametricPlot[
 {a[[1]] t + a[[2]] (1 - t), (*сторона треугольника*)
  a[[3]] +
   Tan[0.999999 (t - 1 / 2) Pi ] ({-(a[[1]] - a[[2]])[[2]], (a[[1]] - a[[2]])[[1]] })
  (*ортогональная прямая. разберитесь,
  как реализовано ортогональное направление. Разберитесь,
  зачем тут стоит тангенс. что будет, если убрать множитель 0.999999?*)
 },
 {t, 0, 1}, PlotRange → {{-1, 3}, {-1, 3}}, Axes → False, Frame → True]
```

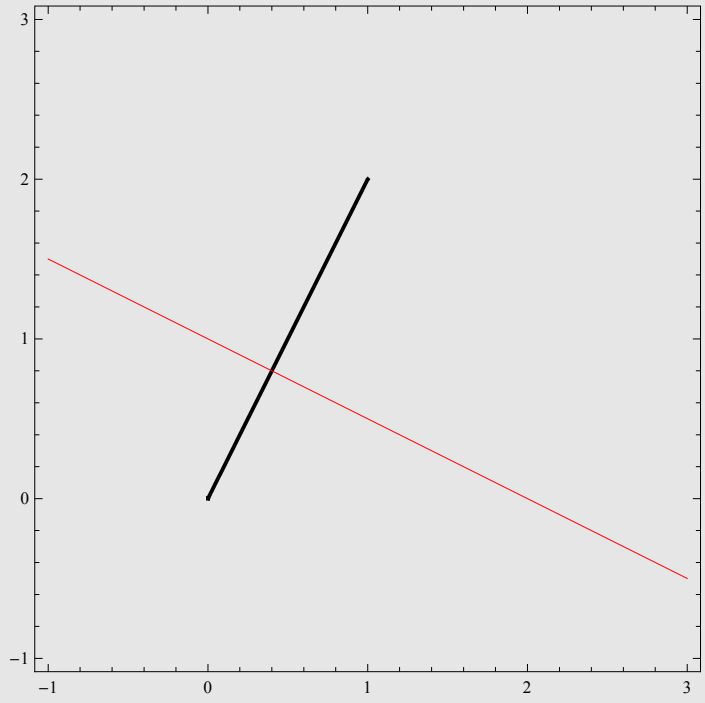
Out[98]=



Однако приличнее и удобнее воспользоваться командой Show, которая позволяет на одном рисунке изобразить неколько разных рисунков.

In[99]:=
```
MIN = -1; MAX = 3;
a = {{0, 0}, {1, 2}, {2, 0}};
Show[
 {ParametricPlot[a[[1]] t + a[[2]] (1 - t), {t, 0, 1},
   PlotStyle → Directive[Thick, Black]], (*сторона треугольника*)
  ContourPlot [ (a[[2]] - a[[1]]).{x - a[[3]][[1]], y - a[[3]][[2]]} == 0,
   {x, MIN, MAX}, {y, MIN, MAX}, ContourStyle → Directive[Thin, Red]]}
 (*ортогональная прямая. разберитесь, как реализована ортогональность*)
 , PlotRange → {{-1, 3}, {-1, 3}}, Axes → False, Frame → True, AspectRatio → 1]
```

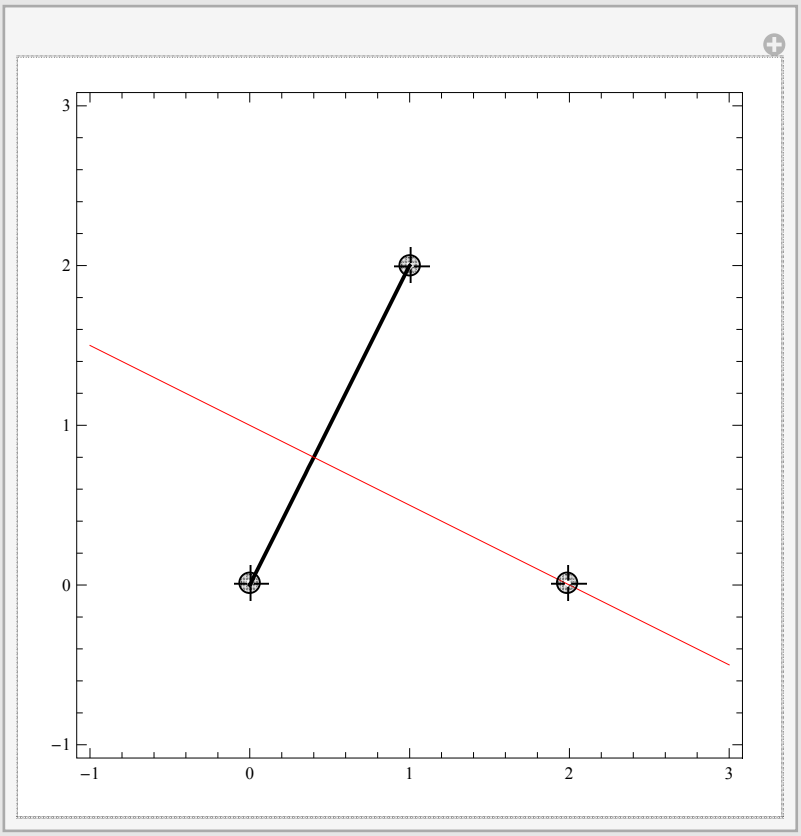
Out[101]=



Теперь добавим динамический выбор трех точек

In[102]:=
```
MIN = -1; MAX = 3;
Manipulate[
 Show[
  {ParametricPlot[a[[1]] t + a[[2]] (1 - t), {t, 0, 1},
    PlotStyle → Directive[Thick, Black]], (*сторона треугольника*)
   ContourPlot [ (a[[2]] - a[[1]]).{x - a[[3]][[1]], y - a[[3]][[2]]} == 0,
    {x, MIN, MAX}, {y, MIN, MAX}, ContourStyle → Directive[Thin, Red]]}
  (*ортогональная прямая. разберитесь, как реализована ортогональность*)
  , PlotRange → {{-1, 3}, {-1, 3}}, Axes → False, Frame → True, AspectRatio → 1]
 , {{a, {{0, 0}, {1, 2}, {2, 0}}}, Locator}]
```

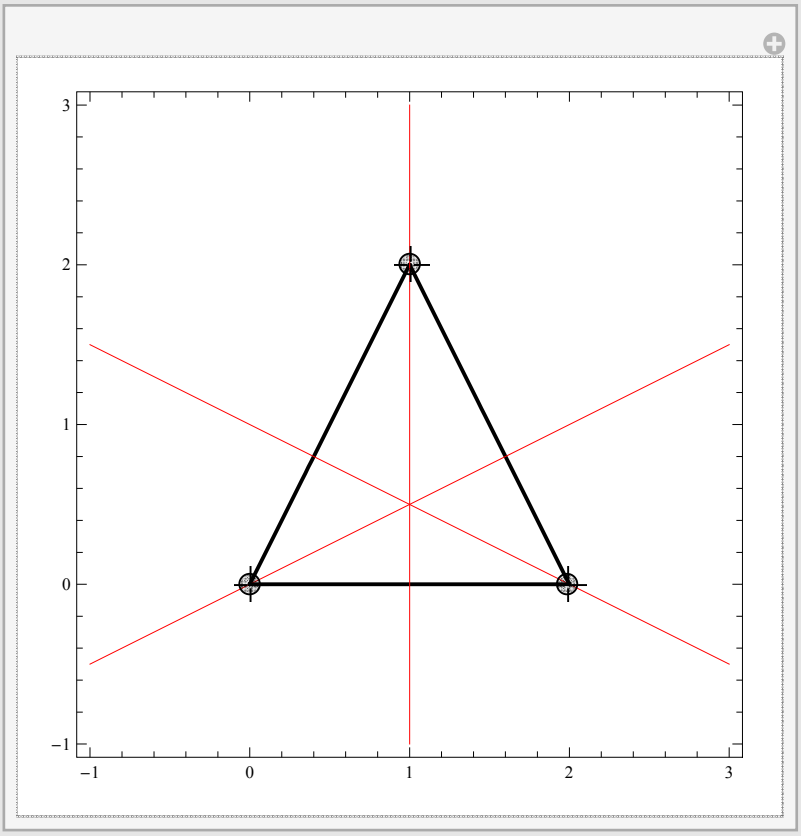
Out[103]=



Оталось так модифицировать код, чтобы на рисунке были все три высоты и все три стороны треугольника. Это можно сделать многими способами, мы поступим довольно топорно.

In[104]:=
```
MIN = -1; MAX = 3;
Manipulate[
 Show[
  Table[ (*в списке три элемента - по одной картинке на каждую сторону*)
   {ParametricPlot[a[[i]] t + a[[Mod[i + 1, 3, 1]]] (1 - t), {t, 0, 1},
     PlotStyle → Directive[Thick, Black]], (*сторона треугольника*)
    ContourPlot [ (a[[i]] - a[[Mod[i + 1, 3, 1]]]).
       {x - a[[Mod[i + 2, 3, 1]]][[1]], y - a[[Mod[i + 2, 3, 1]]][[2]]} == 0,
     {x, MIN, MAX}, {y, MIN, MAX}, ContourStyle → Directive[Thin, Red]]
    (*высота*)
   } , {i, 3}] (*  i+2 (mod 3) - это номер вершины,
  из коротой опускается высота*)
  , PlotRange → {{-1, 3}, {-1, 3}}, Axes → False, Frame → True, AspectRatio → 1
 ], {{a, {{0, 0}, {1, 2}, {2, 0}}}, Locator}]
```

Out[105]=



Этот математический эксперимент должен убедить нас в том, что теорема о высотах треугольника верна. Остается найти строгое доказательство.

Директивы

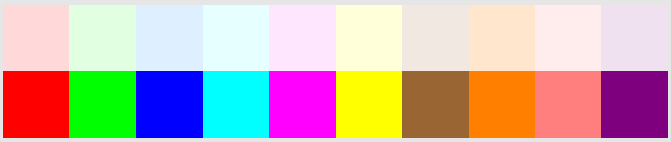
Директивы непосредственного управления цветом

В *Mathematica* цвет задается разными способами. Кроме названий стандартных цветов и четырех основных оттенков серого (тут мы пользуемся упомянутым выше примитивом **Raster**), которыми мы уже пользовались выше, есть возможность задавать цвет в форматах **RGB**, **CMYK**, **Hue** и **GrayLevel** для оттенков серого. В первых двух результирующий цвет представлен как сумма

монохромных слагаемых (**Red-красный**, **Green-зеленый**, **Blue-синий**) и (**Cyan-голубой**, **Magenta-фиолетовый**,**Yellow-жёлтый**, **blacK-чёрный**), принимающих значения от нуля до единицы, что проиллюстрировано в следующих примерах

In[106]:=
```
colors = {Red, Green, Blue, Cyan, Magenta, Yellow, Brown, Orange, Pink, Purple,
   LightRed, LightGreen, LightBlue, LightCyan, LightMagenta,
   LightYellow, LightBrown, LightOrange, LightPink, LightPurple};
cl = Partition[Map[Apply[List, #] &, colors], 10];
Graphics[Raster[cl]]
```

Out[108]=



In[109]:=
```
Graphics[Raster[{Map[#[[1]] &, {Black, White, Gray, LightGray}]}]]
```

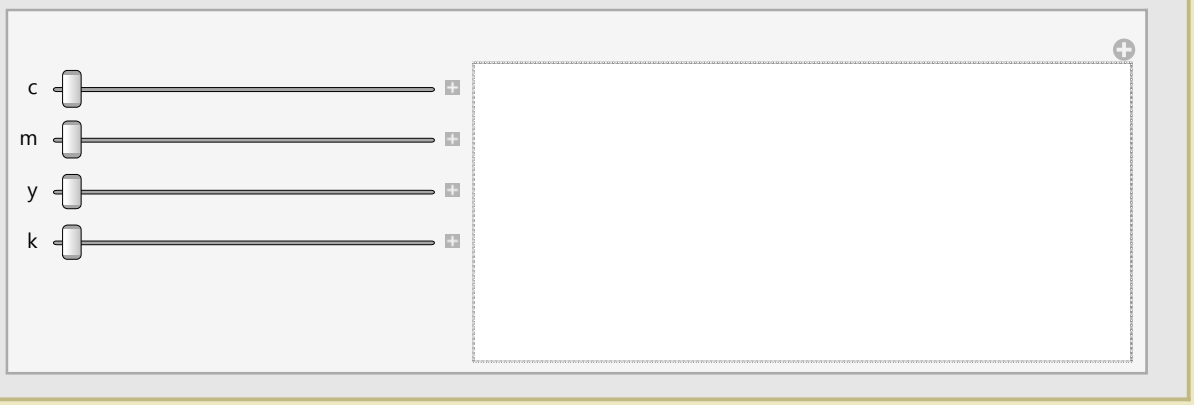
Out[109]=

In[110]:=
```
Manipulate[Graphics[{RGBColor[r, g, b], Rectangle[{0, 0}, {1, 1 / 3}]}],
 {r, 0, 1}, {g, 0, 1}, {b, 0, 1}]
```

Out[110]=

In[111]:=
```
Manipulate[Graphics[{CMYKColor[c, m, y, k], Rectangle[{0, 0}, {1, 1 / 3}]}],
 {c, 0, 1}, {m, 0, 1}, {y, 0, 1}, {k, 0, 1}]
```

Out[111]=



Управление с помощью функции **Hue** может быть как однопараметрическим (от красного к фиолетовому и обратно), так и трёх и четырех параметрическим (Hue, Saturation, Brightness,Opacity__).

In[112]:=
```
Manipulate[Graphics[{Hue[t], Rectangle[{0, 0}, {1, 1 / 3}]}], {t, 0, 2}]
```

Out[112]=

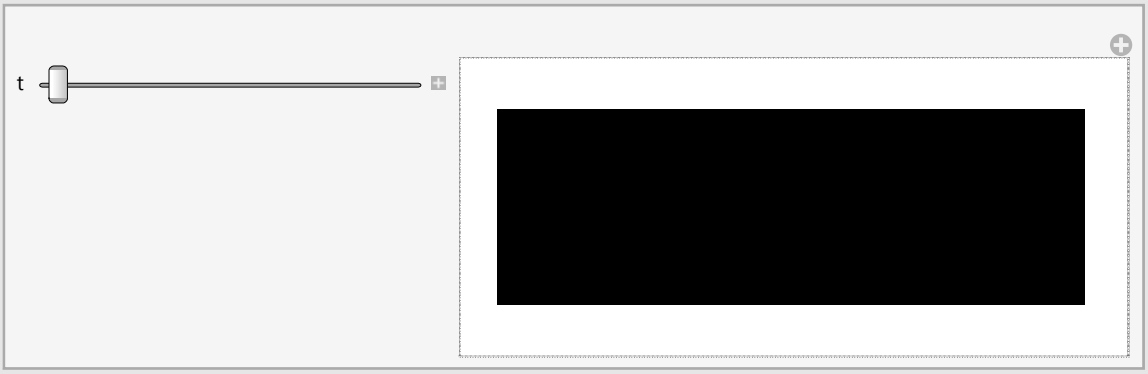
In[113]:=
```
Graphics[Table[{Hue[h], EdgeForm[Gray], Rectangle[{32 h, 0}]}, {h, 0, 1, 1 / 32}]]
```

Out[113]=

Также однопараметрически задаются оттенки серого функцией **GrayLevel**

In[114]:=
```
Manipulate[Graphics[{GrayLevel[t], Rectangle[{0, 0}, {1, 1 / 3}]}], {t, 0, 1}]
```

Out[114]=



Абсолютные и относительные характеристики

Размеры составляющих рисунок примитивов можно задавать как в абсолютных единицах, так и в относительных. Абсолютная единица размера точки равна 1/72 дюйма (т.е. та самая точка-point старого матричного принтера:). Относительная единица - это доля от размера рисунка.

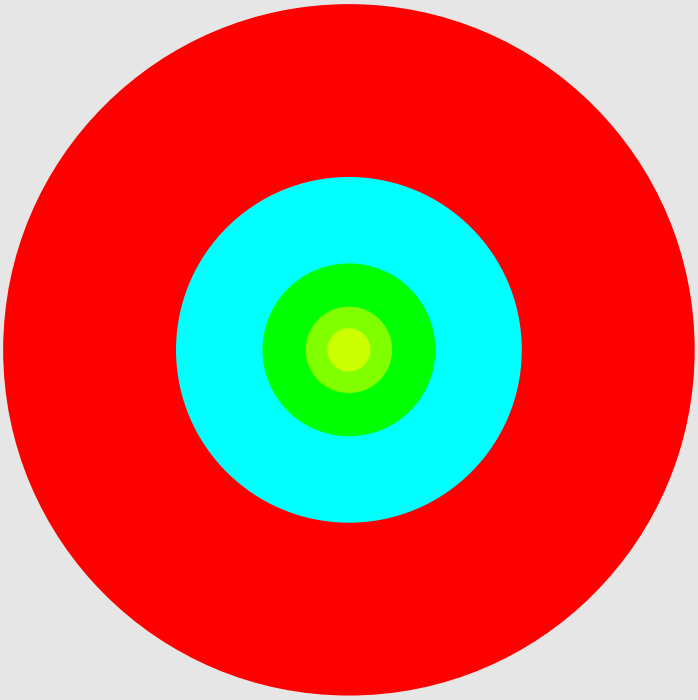
Размер точки

In[115]:=
```
Graphics[Table[{Hue[1 / i], AbsolutePointSize[i], Point[{i, 0}]}, {i, 1, 20}]]
```

Out[115]=

In[116]:=
```
Graphics[Table[{PointSize[1 / 2^(i-1)], Hue[1 / i], Point[{0, 0}]}, {i, 1, 5}]]
```

Out[116]=



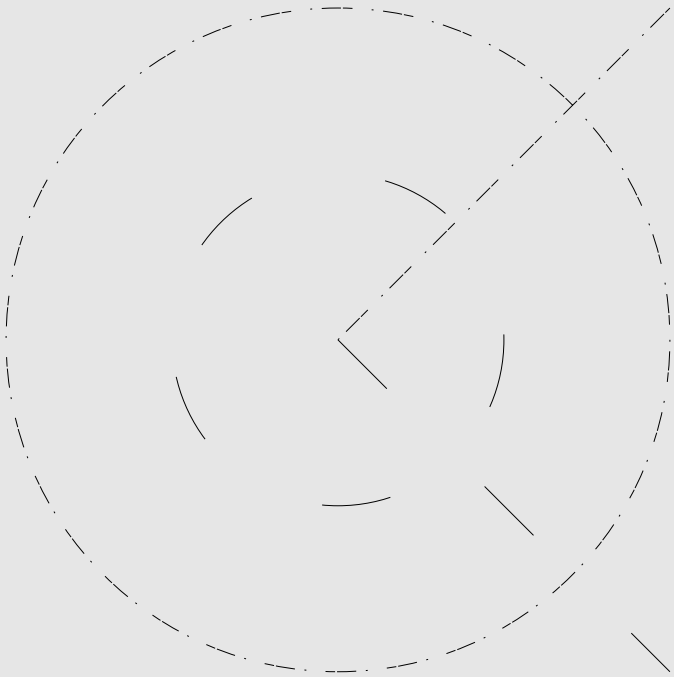
Сплошная или пунктирная линия, размер черточек и пробелов

AbsoluteDashing[{d1, d2, ..}] (*последующие линии рисуются пункциром с циклически повторяющимися интервалами, абсолютной длины d1, d2, .., видимо, в пикселях*)
Dashing[{d1, d2, ..}] (*то же, но размеры в долях ширины всей картинки*)

Чтобы почувствовать разницу между AbsoluteDashing и просто Dashing, полезно поменять мышкой размер рисунка. При этом не меняется длина абсолютных Dash-ей (большая окружность и отрезок, идущий вправо вверх) и количество относительных Dash-ей (все остальные линии).

In[117]:=
```
Graphics[{AbsoluteDashing[{1, 5, 10}], Line[{{0, 0}, {1, 1}}], Circle[],
  Dashing[{.1, .2}], Line[{{0, 0}, {1, -1}}], Circle[{0, 0}, 1 / 2]}]
```

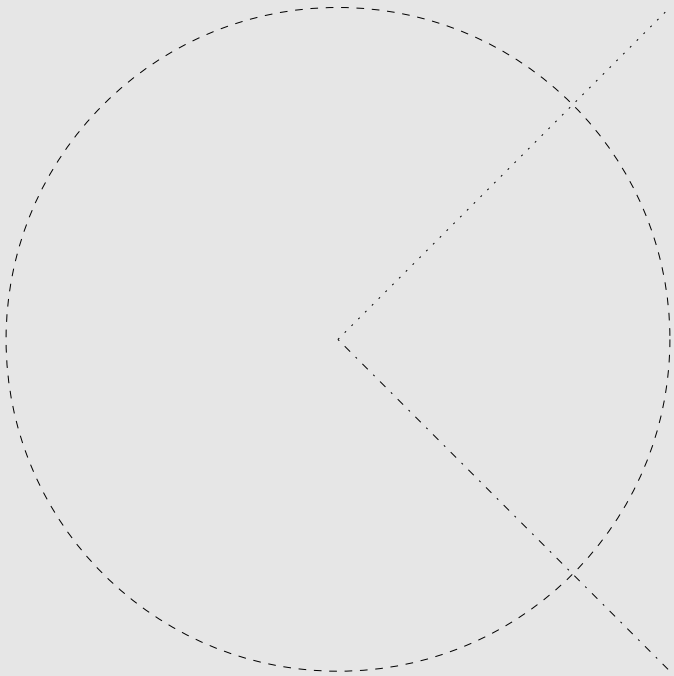
Out[117]=



Возможны сокращенные версии этих абсолютных директив **Dotted**, **Dashed**, **DotDashed**:

In[118]:=
```
Graphics[{Dotted, Line[{{0, 0}, {1, 1}}],
  Dashed, Circle[], DotDashed, Line[{{0, 0}, {1, -1}}]}]
```

Out[118]=



Толщина линии

Аналогично, имеются команды, управляющие толщиной линии

In[119]:=
```
AbsoluteThickness[r] (* абсолютная толщина *)
Thickness[r] (* относительная толщина *)
Thin
Thick
```

Out[119]=
```
AbsoluteThickness[1]
```

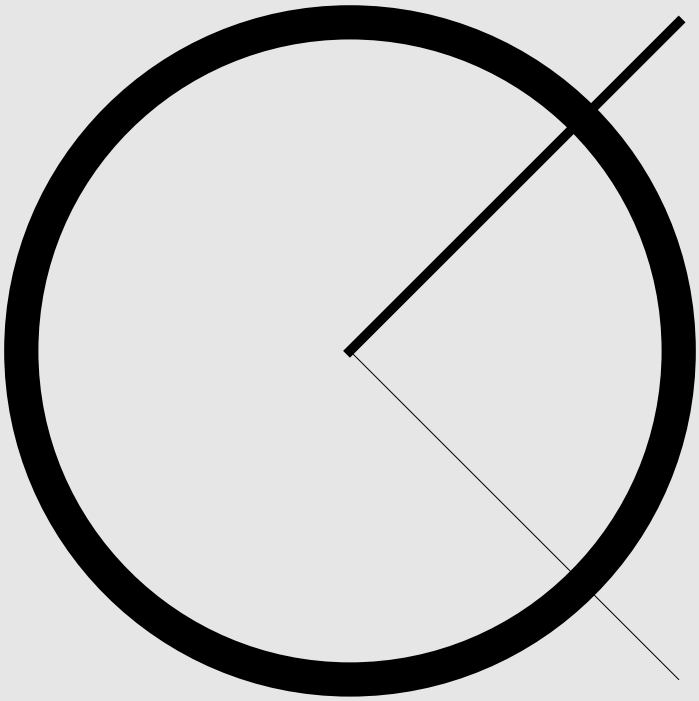
Out[120]=
```
Thickness[1]
```

Out[121]=
```
Thickness[Tiny]
```

Out[122]=
```
Thickness[Large]
```

In[123]:=
```
Graphics[{AbsoluteThickness[5], Line[{{0, 0}, {1, 1}}],
  Thickness[0.05], Circle[], Thin, Line[{{0, 0}, {1, -1}}]}]
```

Out[123]=



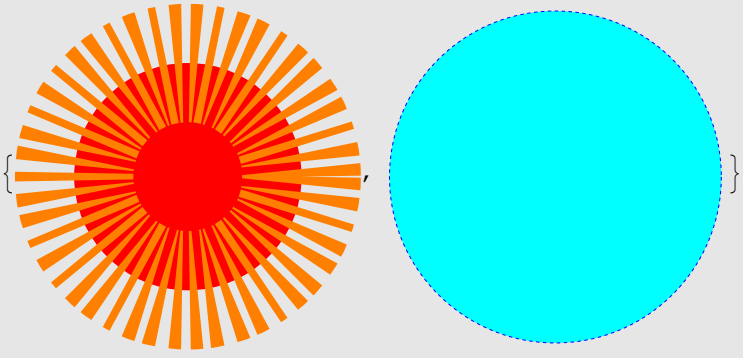
Снова поменяйте мышкой размер рисунка. Не меняется толщина абсолютных линий. Относительные линии меняют абсолютную толщину вместе с рисунком.

Вид многоугольника, диска, прямоугольника

Видом "заполненных объектов" управляют директивы **EdgeForm[список директив]** и **FaceForm[список директив]**. Они определяют цвет, толщину, прозрачность и стиль заполнения. Пустой **FaceForm[]** дает прозрачный многоугольник. В трёхмерной графике (см. ниже) с помощью FaceForm можно определить разные цвета для двух сторон многоугольника (тогда аргумент - список из двух списков директив).

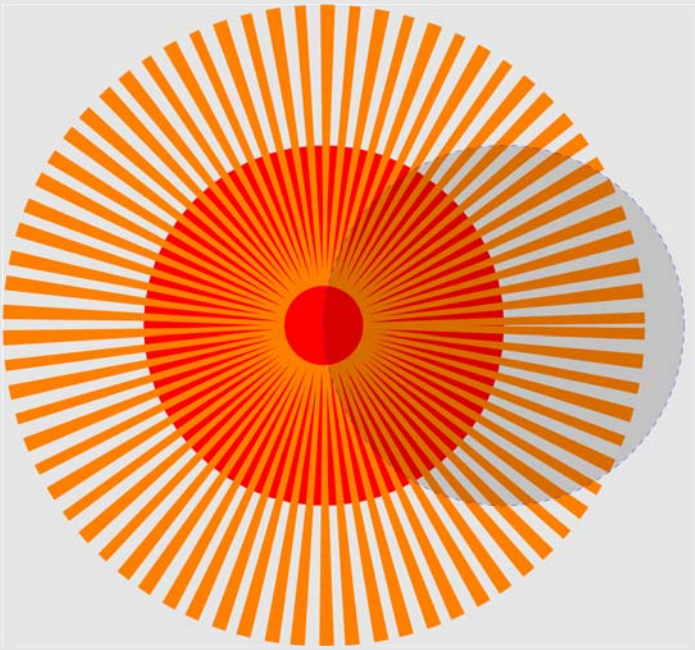
In[124]:=
```
{Graphics[{EdgeForm[{Thickness[0.5], Dashed, Orange}], Red, Disk[]}], Graphics[
  {EdgeForm[{AbsoluteThickness[0.2], Blue, Dashing[{0.01, .01}]}], Cyan, Disk[]}]}
```

Out[124]=



In[125]:=
```
Graphics[{{EdgeForm[{Thickness[0.5], Dashed, Orange}], Red, Disk[{0, 0}, 1]},
  {EdgeForm[{AbsoluteThickness[0.2], Blue, Dashing[{0.01, .01}]}],
   FaceForm[Opacity[0.15], Cyan], Disk[{1, 0}, 1]}}]
```

Out[125]=



- **Директива Directive**

Эта директива служит для формирования единой директивы, являющейся композицией любого набора директив. Удобна при создании сложных комбинаций директив.

Опции

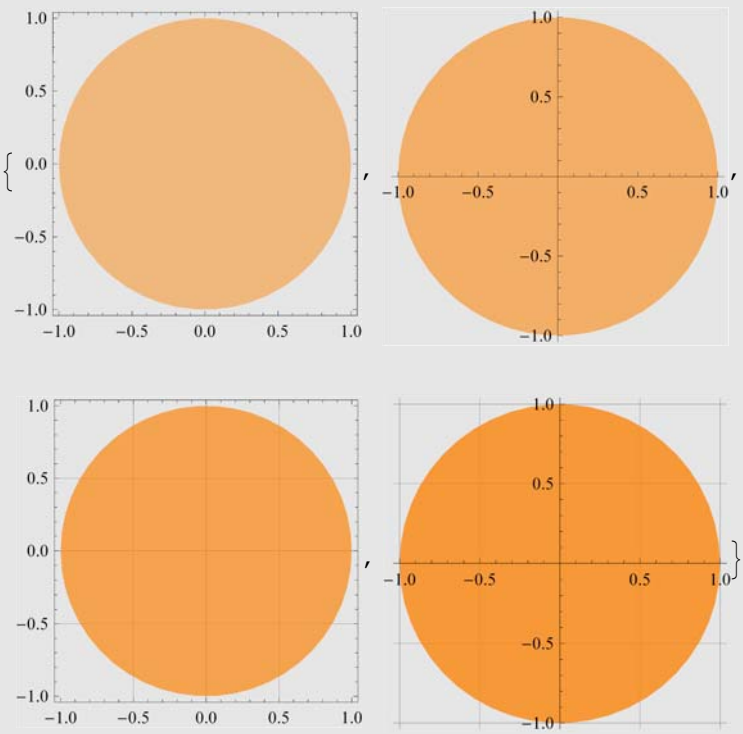
Mathematica также предусматривает множество опций, помогающих придать рисунку удобный, понятный и изящный вид. Общий вид опций следующий: **Имя_опции -> значение**

- **Рамки и координатные оси**

Отношения размеров, взаимное расположение и т.п. удобнее наблюдать при наличиии той или иной системы отсчета. Для этого предусмотрены по умолчанию выключенные **Axes** (координатные оси), и **Frame** (рамка). Чтобы их включить, следует установить их в **True**. Кроме того, можно вызвать координатную сетку **GridLines**, для чего проще всего установить ее в автоматический режим **Automatic**. Всё это, разумеется можно настраивать и украшать.

In[126]:=
```
{Graphics[{FaceForm[{Opacity[0.45], Orange}], Disk[]}, Frame → True],
 Graphics[{FaceForm[{Opacity[0.55], Orange}], Disk[]}, Axes → True],
 Graphics[{FaceForm[{Opacity[0.65], Orange}], Disk[]},
  Frame → True, GridLines → Automatic], Graphics[
  {FaceForm[{Opacity[0.75], Orange}], Disk[]}, Axes → True, GridLines → Automatic]}
```

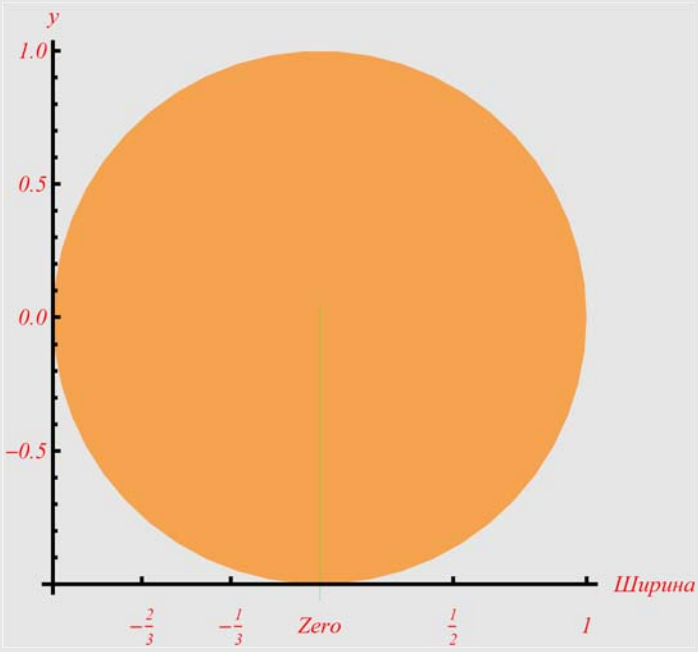
Out[126]=



Так, например, оси можно подписать с помощью **AxesLabel** (приписывается список подписей) и **LabelStyle** (директива как подписывать, или список директив для разных осей), выбрать начало координат **AxesOrigin** (приписываются координаты точки), стиль осей **AxesStyle** (директива или список директив) и деления на них с помощью **Ticks** (дает возможность описать каждое деление на оси максимально подробно: где, что написать, как деление отступает вниз и вверх от оси, как оно выглядит (в примере нуль по горизонтали)) и **TicksStyle** (аналогично)

In[127]:=
```
Graphics[{FaceForm[{Opacity[0.65], Orange}], Disk[]},
 Axes → True, AxesOrigin → {-1, -1}, AxesLabel → {"Ширина", y},
 LabelStyle → Directive[Red, Medium, Italic],
 AxesStyle → Thick, Ticks → {{-1 / 3, -2 / 3,
    {0, "Zero", {0.5, 0.03}, Directive[Green, Thin]}, 1 / 2, 1}, Automatic}]
```

Out[127]=



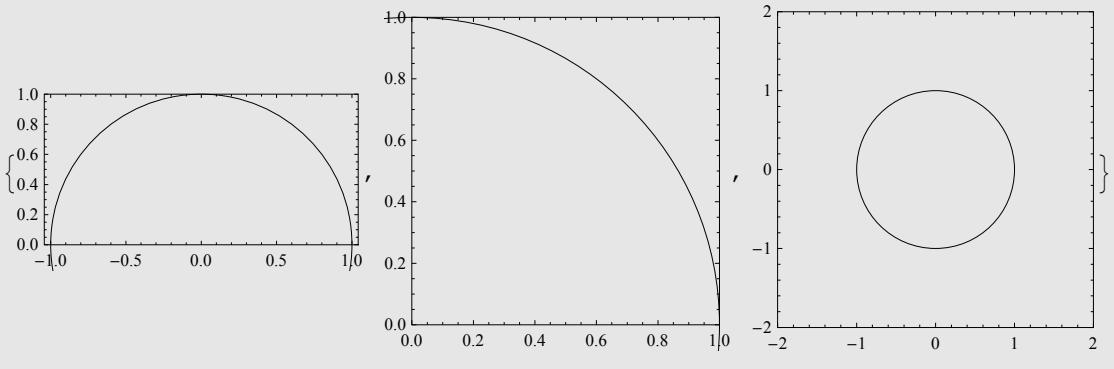
Имеются аналогичные опции для работы с **Frame**.

Регулировка размеров

PlotRange явно указывает какую область рисунка рисовать (в координатах).

In[128]:=
```
{Graphics[Circle[], PlotRange → {0, 1}, Frame → True],
 Graphics[Circle[], PlotRange → {{0, 1}, {0, 1}}, Frame → True],
 Graphics[Circle[], PlotRange → 2, Frame → True]}
```

Out[128]=



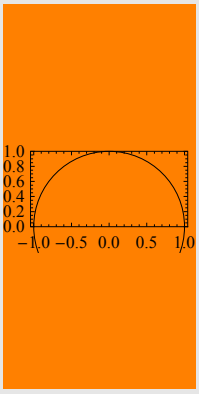
Одна пара значений задает пределы изменения **y** координаты, две пары - обеих координат. Может также стоять одно число, определяющее максимальное значение обеих координат. Также может принимать значения **All** (все точки рисунка) или **Full** (все данные). По умолчанию равен **Automatic**. Последнее означает, что *Mathematica* сама решает, что рисовать, а что нет.

ImageSize в аналогичных терминах задает размер рисунка, как целого. Чтобы лучше видеть

происходящее, удобно воспользоваться опцией **Background**, которая позволяет закрасить весь прямоугольник образа т.е. весь **ImageSize**.

In[129]:=
```
Graphics[Circle[], PlotRange → {0, 1},
 ImageSize → {100, 200}, Frame → True, Background → Orange]
```

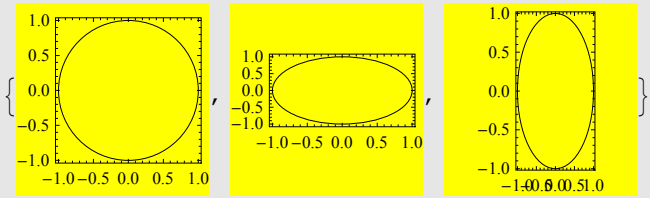
Out[129]=



AspectRatio регулирует отношение высоты рисунка (а не образа) к его ширине.

In[130]:=
```
{Graphics[Circle[], AspectRatio → Automatic,
   ImageSize → {100, 100}, Frame → True, Background → Yellow],
 Graphics[Circle[], AspectRatio → 1 / 2, ImageSize → {100, 100},
   Frame → True, Background → Yellow], Graphics[Circle[], AspectRatio → 2,
   ImageSize → {100, 100}, Frame → True, Background → Yellow]}
```

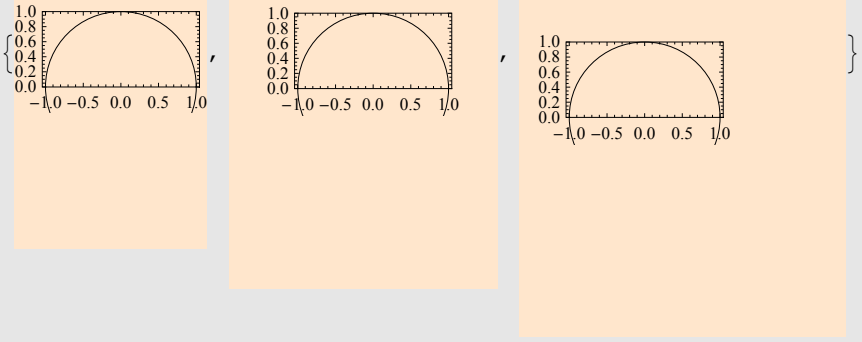
Out[130]=



Отступ от краев прямоугольника образа (наружу) может быть также явно задан с помошью опции **ImageMargins**

In[131]:=
```
{Graphics[Circle[], PlotRange → {0, 1},
  ImageSize → {100, 200}, Frame → True, Background → LightOrange],
 Graphics[Circle[], PlotRange → {0, 1}, ImageSize → {100, 200},
  Frame → True, Background → LightOrange, ImageMargins → 20],
 Graphics[Circle[], PlotRange → {0, 1}, ImageSize → {100, 200}, Frame → True,
  Background → LightOrange, ImageMargins → {{10, 60}, {30, 60}}]}
```

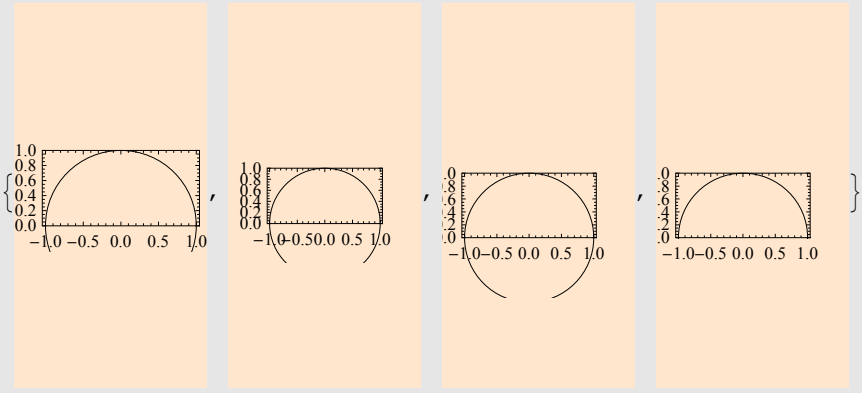
Out[131]=



Отступ от краев прямоугольника образа (внутрь) может быть также явно задан с помошью опции **ImagePadding.** При этом, можно обрезать вылезающие за **Frame (**заданный **PlotRange**) фрагменты рисунка с помощью переключателя **PlotRangeClipping**.

In[132]:=
```
{Graphics[Circle[], PlotRange → {0, 1},
  ImageSize → {100, 200}, Frame → True, Background → LightOrange],
 Graphics[Circle[], PlotRange → {0, 1}, ImageSize → {100, 200},
  Frame → True, Background → LightOrange, ImagePadding → 20],
 Graphics[Circle[], PlotRange → {0, 1}, ImageSize → {100, 200}, Frame → True,
  Background → LightOrange, ImagePadding → {{10, 20}, {30, 40}}], Graphics[Circle[],
  PlotRange → {0, 1}, ImageSize → {100, 200}, Frame → True, Background → LightOrange,
  ImagePadding → {{10, 20}, {30, 40}}, PlotRangeClipping → True]}
```

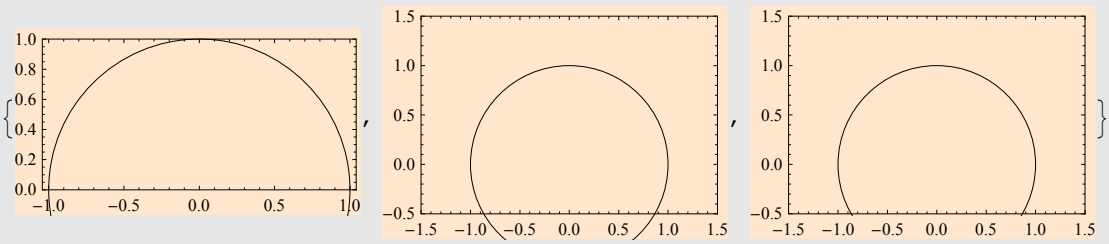
Out[132]=



Расширение Frame от границы, заданной PlotRange, определяется **PlotRangePadding** (в размерных единицах чертежа). При этом, фактически, меняется PlotRange.

In[133]:=
```
{Graphics[Circle[], PlotRange → {0, 1}, Frame → True, Background → LightOrange],
 Graphics[Circle[], PlotRange → {0, 1}, Frame → True, Background → LightOrange,
  PlotRangePadding → 1/2], Graphics[Circle[], PlotRange → {0, 1}, Frame → True,
  Background → LightOrange, PlotRangePadding → 1 / 2, PlotRangeClipping → True]}
```

Out[133]=



Украшательства

Комбинируя опции этого типа, такие как **Background**, **Prolog**, **Epilog** и.т.п. можно регулировать вид фона, взаимное перекрытие объектов и т.п. Имеются также опции, управляющие выводом текста.

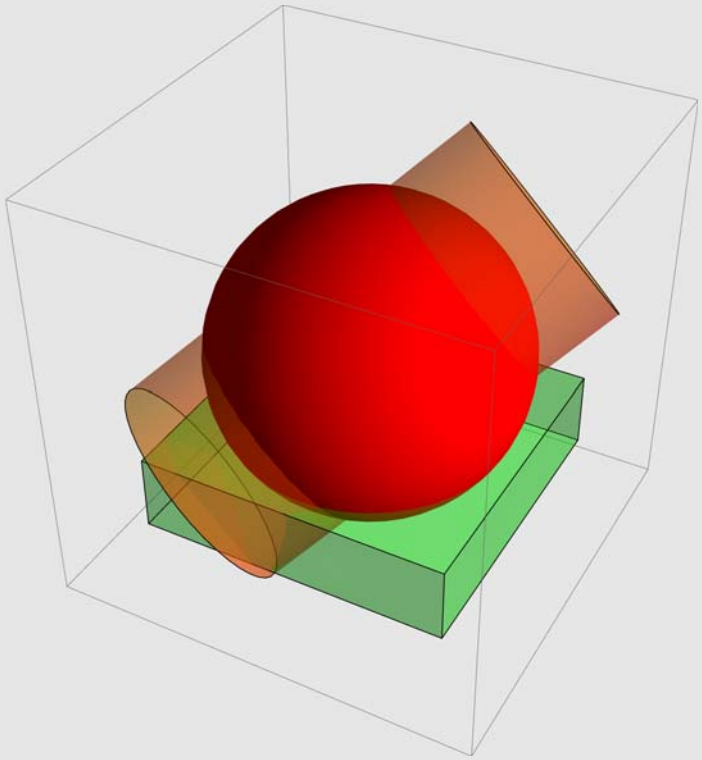
Трехмерный случай

Трёхмерный случай принципиально от двумерного не отличается. Перечислим основные отличия. Прежде всего, добавляется несколько **специфических трехмерных примитивов**.

In[134]:=

```
Graphics3D[{
  {Orange, Opacity[0.4], Cylinder[{{-1.5, -1.5, -1.5}, {1.5, 1.5, 1.5}}, √2]},
  (* Прямой цилиндр, заданный осью-отрезком и радиусом*)
  {Red, Sphere[{0, 0, 0}, 2]}, (* Сфера: центр, радиус*)
  {Green, Opacity[.3], Cuboid[{-2, -2, -2}, {2, 2, -1}]}}]
(*Прямоугольный параллелепипед, заданный координатами противоположных вершин*)
```

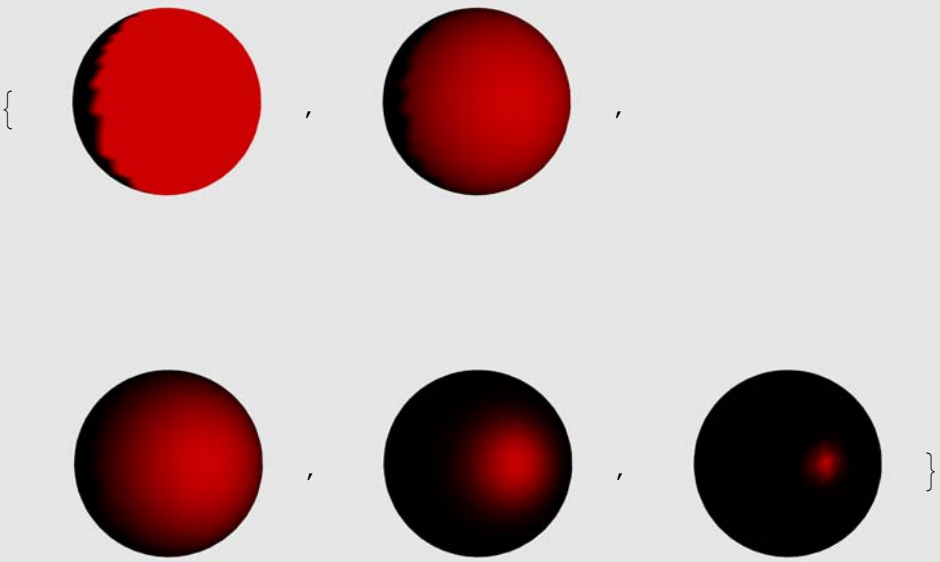
Out[134]=



К графическим директивам, определяющим вид поверхности добавляются две: **Glow[]** (характеризует цвет, которым поверхность светится) и **Specularity []** (характеризует зеркальные свойства поверхности.)

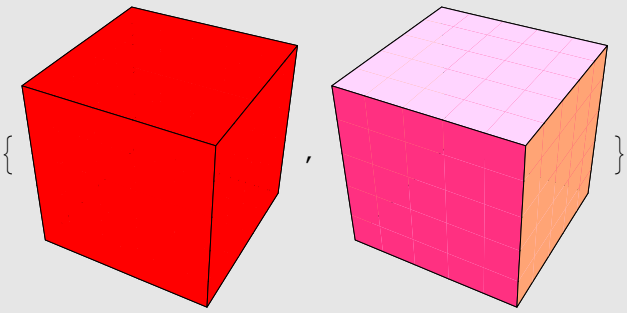
In[135]:=
```
Table[Graphics3D[{Specularity[Red, s], Black, Sphere[]},
  Boxed → False, ImageSize → 150], {s, {0, 1, 2, 10, 100}}]
```

Out[135]=



In[136]:=
```
{Graphics3D[{Glow[Red], Black, Cuboid[]}, Boxed → False, ImageSize → 150],
 Graphics3D[{Glow[Red], White, Cuboid[]}, Boxed → False, ImageSize → 150]}
```

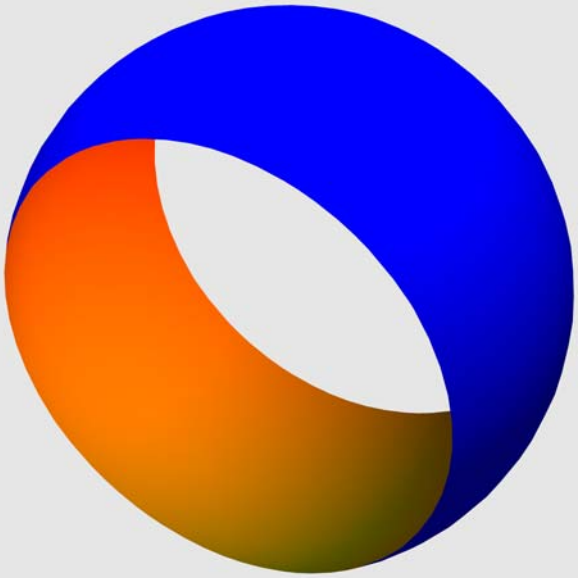
Out[136]=



Кроме того, с помощью **FaceForm** можно теперь задавать разные цвета для разных сторон поверхности.

In[137]:=
```
Graphics3D[{FaceForm[Blue, Orange], Sphere[]},
 PlotRange → {{-1, 1}, {-1 / 2, 1 / 2}, {-1, 1}}, Boxed → False]
```

Out[137]=

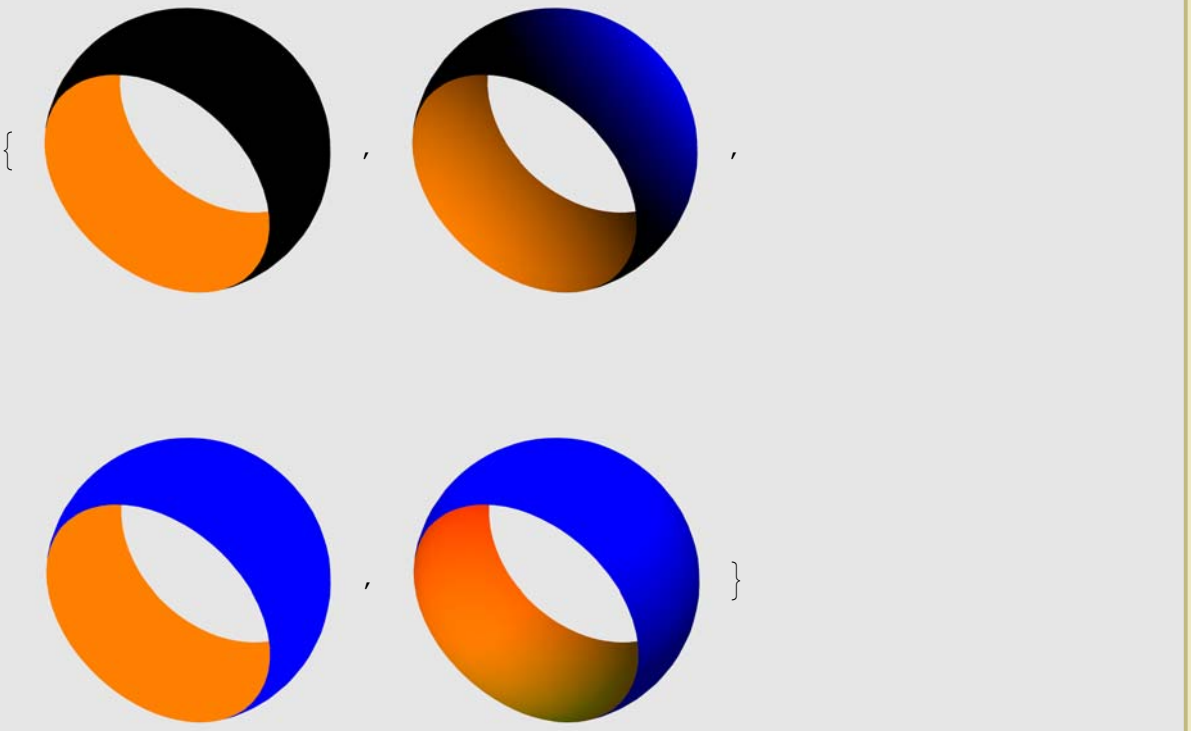


В опциях **Frame** заменяется на **Boxed**, добавляется источник света **Lighting** и опции, характеризующие "точку зрения" наблюдателя и др. характеристики образа.

In[138]:=

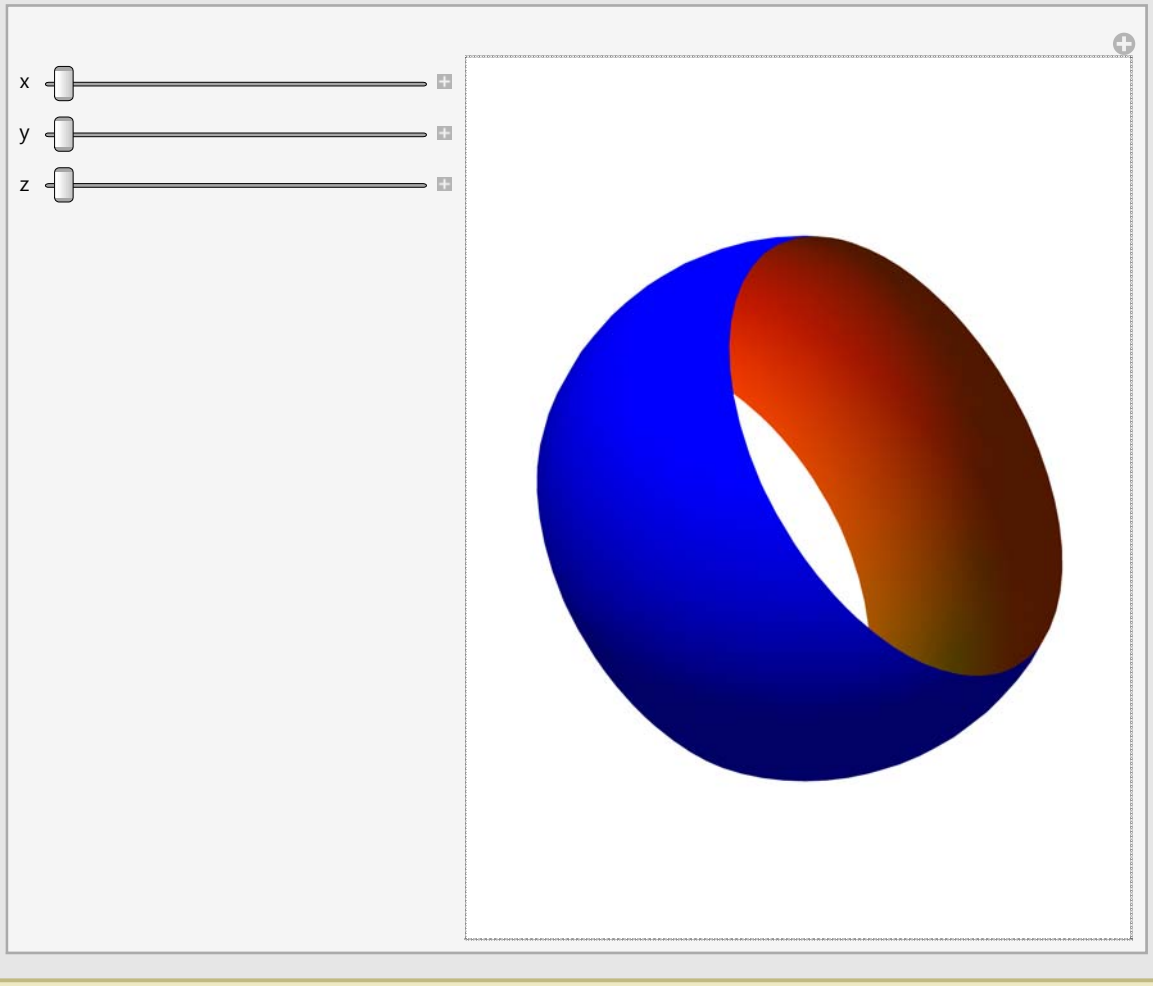
```
{Graphics3D[{FaceForm[Blue, Orange], Sphere[]},
  PlotRange → {{-1, 1}, {-1 / 2, 1 / 2}, {-1, 1}},
  Boxed → False, Lighting → {{"Point", White, {0, 0, 0}}}],
 Graphics3D[{FaceForm[Blue, Orange], Sphere[]},
  PlotRange → {{-1, 1}, {-1 / 2, 1 / 2}, {-1, 1}},
  Boxed → False, Lighting → {{"Point", White, {2, 2, 2}}}],
 Graphics3D[{FaceForm[Blue, Orange], Sphere[]},
  PlotRange → {{-1, 1}, {-1 / 2, 1 / 2}, {-1, 1}}, Boxed → False,
  Lighting → {{"Ambient", White}}], Graphics3D[{FaceForm[Blue, Orange], Sphere[]},
  PlotRange → {{-1, 1}, {-1 / 2, 1 / 2}, {-1, 1}}, Boxed → False, Lighting → Automatic]}
```

Out[138]=



In[139]:=
```
Manipulate[Graphics3D[{FaceForm[Blue, Orange], Sphere[]},
  PlotRange → {{-1, 1}, {-1 / 2, 1 / 2}, {-1, 1}}, Boxed → False,
  ViewPoint → {x, y, z}], {x, -3, 3}, {y, -3, 3}, {z, -3, 3}]
```

Out[139]=



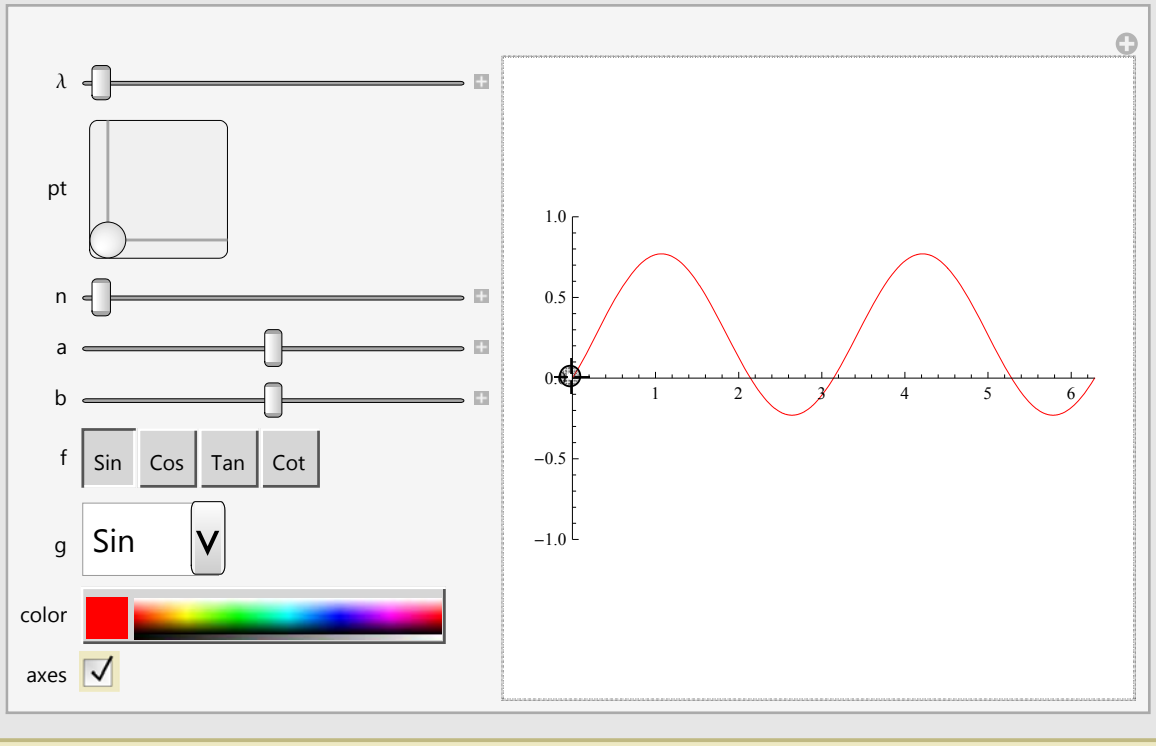
Визуализация зависимости от параметра

Вот пример, который использует почти все возможности Manipulate

In[140]:=

```
Manipulate[Plot[λ f[a (x - loc[[1]]) + b]^n g[x - loc[[1]] + loc[[2]], {x, 0, pt[[1]]},
   PlotRange → {{0, pt[[1]]}, {-pt[[2]], pt[[2]]}}, PlotStyle → {color}, Axes → axes],
 {λ, 1, 2},
 {pt, {2 π, 1}, {4 π, 2}},
 {n, 1, 5, 1},
 {{a, 1}, 0, 2},
 {{b, 1}, 0, 2, 0.2},
 {f, {Sin, Cos, Tan, Cot}},
 {g, {Sin, Cos, Tan, Cot, Sqrt, Log}},
 {color, Red},
 {axes, {True, False}},
 {{loc, {0, 0}}, Locator}
]
```

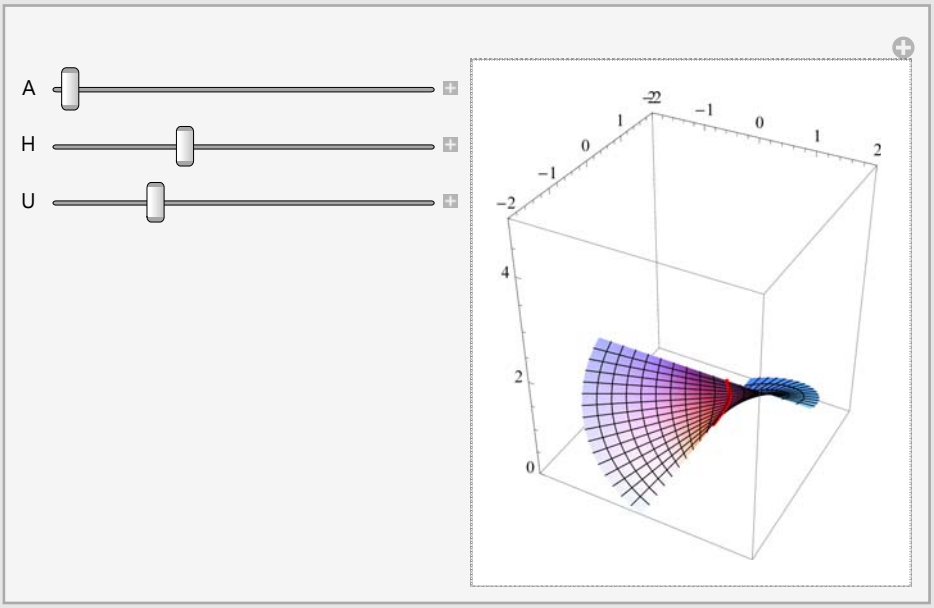
Out[140]=



Геликоид (минимальная линейчатая поверхность) вместе с винтовой линией на нем.
Смысл параметров A, H, U ясен из работы кода.

In[141]:=
```
Manipulate[
 Show[{ParametricPlot3D[{u Sin[v], u Cos[v], H v}, {u, -2, 2}, {v, 0, A},
    PlotRange → {{-2, 2}, {-2, 2}, {0, 5}}, ImageSize → 200(*,Mesh→None*)],
   ParametricPlot3D[{U Sin[v], U Cos[v], H v},
    {v, 0, A} , PlotStyle → {Thick, Red}]} ],
 {A, Pi / 2, 7 Pi}, {{H, 1}, 0, 3}, {{U, 0.5}, 0, 2}]
```

Out[141]=



Движение точки в гравитационном потенциале. ВАЖНО : Использование численного интегрирования дифференциального уравнения (команда NDSolve и следующая за ней Parametric-Plot)

Обратите внимание на то, что происходит с чисенным решением между TT=1200 и TT=1300 (Нажмите на + около линейки со слайдером)

Ваше мнение по поводу системного сообщения, которое выводится коричневым шрифтом после кода, но до картинки?

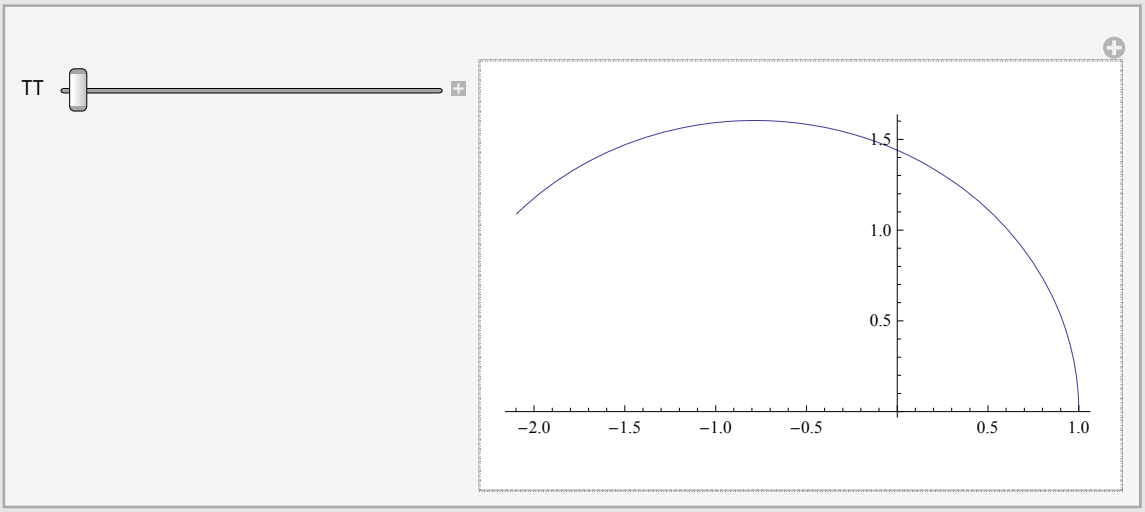
In[142]:=

```
ClearAll;
GG = 1;
s0 = NDSolve[{
     x''[t] == - GG x[t] / ((x[t]^2 + y[t]^2))^(3 / 2),
     y''[t] == - GG y[t] / (x[t]^2 + y[t]^2)^(3 / 2), x'[0] == 0,
     x[0] == 1, y'[0] == 1.2, y[0] == 0}, {x, y}, {t, 0, 1500}];

Manipulate[ParametricPlot[Evaluate[{x[t], y[t]} /. s0], {t, 0, TT}],
 {{TT, 5}, 5, 1500, 100, Appearance → Labeled}]
```

NDSolve::mxst : Maximum number of 10000 steps reached at the point t == 1217.1134894793197`. ≫

Out[145]=



КОНЕЦ ВТОРОЙ ЛЕКЦИИ